On improving analysis and testing of open- and closed-source Android Apps

Camilo Escobar Velásquez

Research Advisor

Prof. Dr. Mario Linares Vásquez

Research Co-Advisor

Prof. Dr. Gabriele Bavota

Dissertation Committee

Prof. Dr. Denys Poshyvanyk	William & Mary, VA, USA
Prof. Dr. Massimiliano Di Penta	University of Sannio, Italy
Prof. Dr. Nicolás Cardozo	Universidad de los Andes, Colombia

Dissertation accepted on 15 June 2023

Research Advisor	Research Co-Advisor
Prof. Dr. Mario Linares Vásquez	**Prof. Dr. Gabriele Bavota**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Camilo Escobar Velásquez
Bogotá, Colombia, 15 June 2023

Abstract

The amount of available Android applications on marketplaces is having an increasing trend, leading to a highly competitive environment between similar apps. To stand out, practitioners and researchers need to ensure that the development process is supported on approaches and tools that help developers to release high quality applications frequently. This quality assurance process can be done in three main phases by preventing, identifing and fixing issues/bugs. To help in this process, a plethora of automated approaches have been proposed, exploiting the capabilities and information provided by source-code. However, practitioners and researchers work on heterogenous environments in which the access to the source code might be restricted or its not needed due to the capabilities available at working at APK level. Therefore, proposing approaches that work at both level of access (*i.e.,* source-code and APK) is required to improve the state of the art and enhance the development process.

In this research, we aim on enabling the automation of software engineering tasks by (i) exploiting the knowledge available from approaches at source code level, and (ii) extending it with new approaches for closed-source apps. We focused on 4 aspects towards enhancing the quality of analysis and testing of Android apps: (i) mutation testing, (ii) internationalization testing, (iii) interaction-based testing, and (iv) connectivity testing. In order to do this, we conducted a set of empirical studies, and designed and implemented approaches to enhance the aforementioned aspects at APK and source-code level.

Acknowledgements

<PLACEHOLDER>

Contents

List of Figures

List of Tables

Part I

Prologue

1

Introduction

Mobile markets have pushed and promoted the rise of an interesting phenomenon that has permeated not only developers' culture but also human beings' daily activities. Mobile devices, apps, and services are helping companies and organizations make "digital transformation" possible through services and capabilities offered close to the users. Nowadays, mobile apps and devices are the most common way to access those services and capabilities; in addition, apps and devices are indispensable tools for allowing humans to have in their phones computational capabilities that make life better and more accessible.

The mobile apps phenomenon has also changed how practitioners design, code, and test apps. Mobile developers and testers face critical challenges in their daily life activities, such as (i) continuous pressure from the market for frequent releases of high-quality apps, (ii) platform fragmentation at device and OS levels, (iii) rapid platform/library evolution and API instability, and (iv) a developing market with millions of apps available for being downloaded by ends users [2, 3]. Tight release schedules, limited developer and hardware resources, and cross-platform delivery of apps are common scenarios when developing mobile apps [2]. Therefore, reducing the time and effort devoted to software engineering tasks while producing high-quality mobile software is a "precious" goal.

The development cycle of an Android application can be described by 4 main steps: (i) development, (ii) testing, (iii) publication on applications markets, and (iv) Review/ratings analysis. To enhance this development process, practitioners and researchers have contributed by proposing approaches, mechanisms, best practices, and tools that improve different steps of the development cycle. For instance, cross-platform languages and frameworks (e.g., Flutter, Ionic, Xamarin, React Native) reduce the development time by providing developers with a mechanism for building Android and iOS versions of apps in a write-one-run-anywhere way [2, 4]. Automated testing approaches help testers increase the apps' quality and reduce the detection/reporting time [5, 6, 7]. Automated categorization of reviews also allows developers to select relevant information, issues, features, and sentiments from a large volume of reviews posted by users [3, 8, 9].

Although, identifying an issue after the publication of an app impacts the application's reputation and can lead users to replace the app with one of the existing competitors in the application marketplaces. Therefore, despite the previous work done in these areas, two

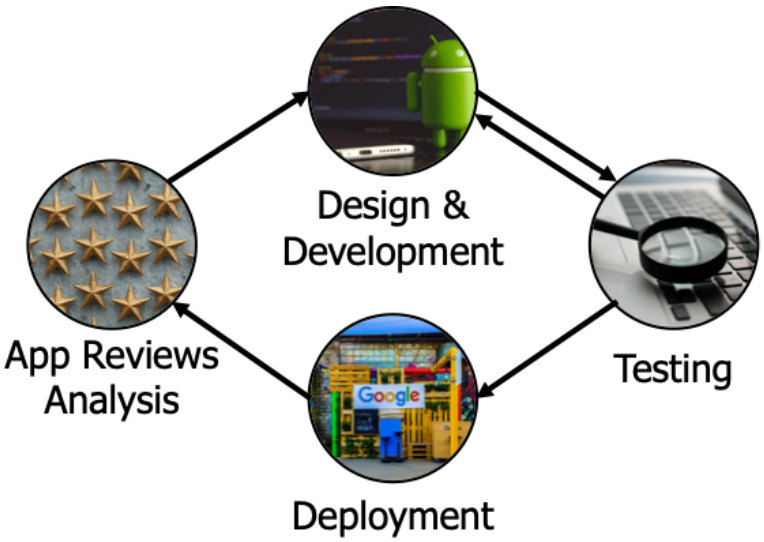


Figure 1.1. Android App Development Cycle

main steps of the development cycle (*i.e.,*, development and testing) might be enhanced to increase the productivity and quality of the generated applications. On the one hand, for the *development* step, there is previous work towards the static analysis of Android apps to help developers early detect different bugs and issues that, without automated support, could be time-consuming processes — when doing the analysis manually [10] —. Therefore, it is valuable to increase and improve the available approaches, techniques, and mechanisms to mitigate the injection of bugs in the developing phase. Specifically, in the case of static analysis of source code to propose an approach to identify connectivity issues in the early stages of the development process.

On the other hand, researchers and practitioners have addressed the *testing* step by proposing a set of approaches to ease and improve the execution of testing tasks for developers. However, most of these approaches depend on the usage of source code, generating a limitation to the use of crowd-sourced services that could fasten the testing process based on the capabilities that these services can provide, such as device farms and access to final users for beta testing. Therefore, there is a gap on the available approaches to test Android apps by using only APK files. Example of this missing techniques are internationalization analysis and testing, and quality assurance for app features based on the interaction between two devices. Therefore, working towards enabling the execution of software engineering tasks at the APK level will provide crowd-sourced and third-party services with additional capabilities that developers and practitioners can exploit to improve their development cycle.

Besides this, as it is shown in Fig. 1.1, after performing the corresponding tasks at the testing phase, the practitioners might go back to the development phase to fix identified issues before publishing a new version of an app. As part of the process of ensuring the quality of an android app, the repairment of issues before providing access to users can improve the reputation of the app and avoid its replacement with a competitor.

1.1 Problem Statement

The reason behind working towards improving analysis and testing at both source code and APK level relies on the existing benefits and limitations of both approaches. For example, the closeness of source code to humans (*i.e.,* practitioners and researchers) provides a more prosperous environment to understand practices and development techniques. However, it has a dependency on the development environment specifics. In contrast, APK level approaches are less prone to fail due to configuration settings, but their modification can quickly lead to corruption of the final app. Likewise, working at source code allows approaches to avoid injecting bugs and takes advantage of the optimization process made by compilers, unfortunately, at the expense of longer execution times. In comparison, working at APK level avoids the compilation process, reducing the execution times of the processes but affecting the understandability of the generated resources and code representations. At the same time, working at APK level enables the modification of the app after its compilation, providing access for third parties to offer new services without requiring access to source-code. For example, mobile markets might offer extended analysis of the apps metrics by adding instrumentation or custom library usages.

Additionally, after analyzing the state-of-the-art, we identified a set of tasks that practitioners did not address previously.The generation of approaches and studies will positively affect the development and quality assurance processes for mobile apps. First, we identified that proposed approaches for mutation testing of android apps rely on the usage of source code to evaluate the coverage and scope of the implemented test. Even knowing that mutation testing allows the evolution of apps by the analysis of not recognized mutants, providing an approach at APK level might reduce the time required to execute/explore/identificate generated mutants, since faults are being injected directly at APK level.

Likewise, we identified that there was no previous approach towards mitigating connectivity bugs for android apps. This research gap has claimed importance due to the increase in access to the internet worldwide and users' reliance on their mobile devices to perform critical transactions even in unstable conditions.

Moreover, the global availability of application markets has enabled the usage of applications by users around the world, generating new requirements regarding the automatic adaptation of GUI to different languages and locations (*i.e.,* internationalization), and also regarding the enabling of mechanisms to communicate or interact between users regardless of the platform used (*i.e.,* mobile or web) . This created a need for approaches that work toward ensuring these capabilities within apps. However, for the case of internationalization, working at source code level is limited due to the possible usage of graphical component, such as fragments, that are only loaded within activities during the execution of the app. Therefore, static analysis approaches require more efforts to recognize implicit relations within GUI components that are loaded together during the execution. An alternative to this additional effort is the dinamyc analysis of the app, that by working at source-code level would require for constant compilation and decompilation tasks to modify the app and then test the impact of changes. This limitation can be mitigated while working at APK level, since changes are included directly in the APK, reducing the time required to have an executable

version of the app with new changes. Additionally, since changes are done directly to the APK, developers and mobile markets might create temporal releases, enabling localization and internationalization features while the complete protocol at source-code is triggered to evaluate and include the corresponding changes.

Similarly, in the case of automated testing of android apps, specifically for interaction capabilities, events are triggered directly to the GUI, therefore working at APK level could provide a faster route to include changes in the app and test its impact on the interaction between users. Nevertheless, practitioners and researchers might analyze the source code to identify a set of features that are based on interaction, such as communication to a specific backend or service.

1.2 Research Goals

Consequently, the goal of this thesis is to *devise and assess a set of techniques to address four identified research opportunities in the analysis and testing of open- and closed-source Android apps*. In particular, the specific goals of this thesis are:

Goal 1: Evaluate the feasibility and impact of performing mutation testing at APK level. To achieve this, we will design a framework to support the automation of software engineering tasks at APK level and evaluate the feasibility of its implementation by using mutation testing as a case study. As a requirement for this goal, it is necessary to study and analyze the existing approaches that focus on generating intermediate representations and models that enable the static analysis of APKs. As a result, we will compare the designed framework and its usage for mutation testing with the existing approaches for mutation testing at source-code level. The comparison will allow us to identify the improvements and shortfalls of using APK files as input in terms of execution time, quality of the generated mutants, and capabilities enabled by executing this technique at APK level.

Goal 2: Understand and address the underlying reasons behind common *internationalization* issues within Android Apps. To achieve our goal, we will design a technique that identifies collateral changes that might be presented in GUIs due to the misconfiguration of graphical components and textual content to adapt to the different available languages user might use in their devices. Based on this technique, we will conduct an empirical study in which we plan to understand and categorize the issues identified during the automatic exploration of android apps using different languages. As a result of this characterization, we will then be able to study the possible solutions to the identified issues. The previously mentioned solutions might be generated from the official guidelines for the internationalization of Android Apps or the learned lessons of the approaches proposed for the internationalization of other software environments such as web and iOS.

Goal 3: Understand and address the underlying reasons behind common *connectivity* issues within Android App. To achieve this goal, we will conduct an empirical study to manually identify connectivity issues associated with core functionalities within android apps. We will enhance the issue identification by analyzing the application's source code to identify the underlying reasons for the issue. The result of this study will be a taxonomy

characterizing the common issues identified and an analysis of the underlying causes identified during the study. This result will allow us to work on building an automated approach to mitigate the injection of connectivity bugs to Android apps.

Goal 4: Propose solutions to existing limitations on automated testing of android apps. To achieve our goal, we will study the existing approaches to perform testing of Android apps in the search for insights to enable interaction-based testing and contribute to the state of art on the usage of Deep Reinforcement Learning for automated testing. As a result of this search, we will design an approach that exploits the capabilities of the Android environment to enable the interaction of multiple devices, to execute synchronously test scenarios via information and notification exchange between the devices. After the design phase, we will direct our efforts on implementing the proposed approach and evaluate its functionality over a set of Android apps having features based on interaction. For example, multiplayer games, social networks, or email apps.

In parallel, we will design and implement a tool based on DRL techniques using composite reward functions based on the number of identified failures, the partially achieved coverage and the rate of change of the GUI. The use of these variables as parameters of the reward function will be evaluated by comparing our approach with baseline tools to identify improvements on the amount of identified issues and the achieved method coverage. Additionally, we will analyze our results to identify orthogonality relations between approaches.

1.3 List of contributions

In the following section we enlist the main contributions and accomplishments. A similar list of accomplishments can also be found in the beginning of each chapter. We use 🗐🔍 icons for publications presenting empirical studies, 🖧 icon for catalogs/taxonomies output of such studies. We use instead 🗐⚙ icons for publications presenting and/or experimenting automated tools created to support the research and thesis objectives.

Mutation Testing → Part II

Mutation Testing is a technique designed to assess the quality of test suites generated for different types of apps and software. It consist on the generation of "copies" of the Application Under Test (AUT) with minimal changes that represent possible faults that could be made by developers. The objective of this technique is to create a set of "mutants" that helps practitioners check if the test suite is capable of recognizing the modified versions of the AUT. In order to perform Mutation Testing, three main phases need to be done: (i) mutant generation/selection, (ii) mutant execution and (iii) result analysis. In Part II we present our efforts towards mutant generation and selection.

Mutant Generation → *Chapter 3*

Mutant generation process consist of identifying locations within the application code in order to modify certain code statement with a change that inject faults frequently done by developers. Nevertheless, this process requires the prior identification of common faults and

its location. Currently proposed approaches rely on the usage of source code to create the copies of the AUT. In order to solve these issues, we built and evaluated an approach that generates mutants using APK as input. The usage of a compile representation of the app reduces the efforts required to create copies, mutate it (insert fault in code), and recompile. Our results in this matter was published in the following articles:

Enabling mutant generation for open-and closed-source android apps. [11]
Escobar-Velásquez, C., Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Cárdenas, C., and Poshyvanyk, D. *IEEE Transactions on Software Engineering, 2020.*

MutAPK: Source-codeless mutant generation for android apps. [12]
Escobar-Velásquez, C., Osorio-Riaño, M., and Linares-Vásquez, M. In *Proceeding of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019.*

Mutant Selection → *Chapter 4*

Mutation testing (i) generates large sets of fault-injected-versions of an original app and (ii) execute these with the purpose of evaluating the quality of a given test suite. In the case of Android apps, as it is shown in Chapter 3, mutant generation and testing effort could be enhanced when the mutants are generated at the APK level. To reduce that effort, useless (e.g., equivalent) mutants should be avoided and mutant selection techniques could be used to reduce the set of mutants used with mutation testing. Taking this into account, we extended our open source mutant generation tool (MutAPK) to provide selection techniques in its execution. Our results in this matter was published in the following article:

> **MutAPK 2.0: A tool for reducing mutation testing effort of Android apps. [13]**
> **Escobar-Velásquez, C.**, Riveros, D., and Linares-Vásquez, M. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020.*

Internationalization of Android Apps → Part III

Mobile markets allow developers to easily distribute mobile apps worldwide and collect complaints and feature requests in the form of user reviews and star ratings. Therefore, internationalization (i18n) of apps is a highly desired feature, which is currently supported in mobile platforms by using resources files with strings that can be internationalized manually. This manual translation can be a time consuming and error-prone task when the app is targeted for different languages and the amount of strings to be internationalized is large. Moreover, the lack of consideration of the impact of internationalized, strings can drive to collateral (i.e., unexpected) changes and bugs in the GUI layout of apps.

Prevention and Detection of i18n Issues → *Chapter 6*

In this chapter, we present an empirical study on how i18n can impact the GUIs of Android apps. In particular, we investigated the changes, bugs and bad practices related to GUIs when strings of a given default language (i.e., English in this case) are translated to 7 different languages. To this, we created a source- codeless approach, ITDroid, for automatically (i) translating strings, and (ii) detecting bad practices and collateral changes introduced in the GUIs of Android apps after translation. ITDroid was used on a set of 31 Android apps and their translated versions. Then, we manually validated the i18n changes that introduced bugs into the GUIs of the translated apps. Our results in this matter was published in the following articles:

> **An empirical study of i18n collateral changes and bugs in guis of android apps. [14]**
> **Escobar-Velásquez, C.**, Osorio-Riaño, M., Dominguez-Osorio, J., Arevalo, M., and Linares-Vásquez, M. In *Proceedings of the 36th IEEE international conference on software maintenance and evolution (ICSME), 2020.*

Taxonomy of i18n collateral changes and bugs → Fig. 6.4
Based on our previous study we built a taxonomy in which we distinguish i18n changes from i18n bugs. Concerning the later, there are bugs introduced by developers due to the lack of i18n practices (e.g., hard-coded strings), and collateral bugs induced by the i18n collateral changes

ITDroid: A tool for automated detection of i18n issues on android apps. [15]
Escobar-Velásquez, C., Donoso-Diaz, A., and Linares-Vásquez, M. In *Proceedings of the 8th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft), 2021.*

Reporting and Repairing i18n Issues → *Chapter 7*

Continuing with the developement process of ITDroid, in this chapter we present our efforts towards enabling visualization and repairment of i18n bugs automatically identified and localized by ITDroid. We enhanced the final report to provide a more comprehensive guide of the issues along with the improvement of the tool to enable the capability to automatically repair a subset of the identified issues. The main result of this process is the extension of our open source tool.

Connectivity Management for Android Apps → Part IV

Mobile apps have become indispensable for daily life, not only for individuals but also for companies/organizations that offer their services digitally. Inherited by the mobility of devices, there are no limitations regarding the locations or conditions in which apps are being used. For example, apps can be used where no internet connection is available. Therefore, offline-first is a highly desired quality of mobile apps. Accordingly, inappropriate handling of connectivity issues and miss-implementation of good practices lead to bugs and crashes occurrences that reduce the confidence of users on the apps' quality.

Identification of Connectivity Issues on Android Apps → *Chapter 9*

We conducted the first study on Eventual Connectivity (ECn) issues exhibited by Android apps, by manually inspecting 971 scenarios related to 50 open-source apps. We found 304 instances of ECn issues (6 issues per app, on average) that we organized in a taxonomy of 10 categories. Based on our findings, we distill a list of lessons learned for both practitioners and researchers, indicating directions for future work. Our results in this matter was published in the following article:

Studying eventual connectivity issues in Android apps[16]
Escobar-Velásquez, C., Mazuera-Rozo, A., Bedoya, C., Osorio-Riaño, M., Linares-Vásquez, M., and Bavota, G. In *Empirical Software Engineering, 2022.*

> **Taxonomy of eventual connectivity issues → Fig. 9.5**
> This taxonomy includes high level categories (represented by rounded squares) and low level categories (represented by rectangles). We identified 21 low-level types of issues, grouped into 10 high-level categories. This taxonomy includes a total of 304 issues identified during our analysis&testing process.

Automated- Prevention, Detection and Repairment of Connectivity Issues → *Chapter 10*

Taking into account the identified errors on the previous chapter, we identified that connectivity issues (e.g., mishandling of zero/unreliable Internet connection) can result in bugs and/or crashes, negatively affecting the app's user experience. While these issues have been studied in the literature, there are no techniques able to automatically detect and report them to developers. In order to tackle this, we built CONAN, a tool able to detect statically 16 types of connectivity issues affecting Android apps. We assessed the ability of CONAN to precisely identify these issues in a set of 44 open source apps, observing an average precision of 80%. Our results in this matter was published in the following article:

> **Detecting connectivity issues in android apps [17]**
> Mazuera-Rozo, A., **Escobar-Velásquez, C.**, Espitia-Acero, J., Linares-Vásquez, M., and Bavota, G. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.*

Automated Testing for Android Apps → Part V

Mobile devices and apps have a primordial role in daily life, and both have supported daily activities that solve daily human taks, such as calling, messaging or performing transactions. Due to the criticality of the tasks performed by users, the quality assurance process of apps required the usage of different techniques to provide better results. For example, testing the interaction between devices and platforms help identify possible issues inherited by the multi-device environment. Because of this, developers are required to test combinations of heterogeneous interactions to ensure a correct behavior of multi-device and multi-platform apps. Unfortunately, to the best of our knowledge, there is no existing open source tool that enables testing for those cases. Likewise, commonly used testing approaches rely on constant interaction with testers to configure, execute or analyze the tests suite/results. Nevertheless, new approaches have been generated based on the usage of Machine Learning and more expert techniques such as Deep Reinforcement Learning. Our interest is to study and extend the state of the art regarding the usage of such DRL-based techniques for the automated testing of Android Apps.

Enabling Automated Platform-Agnostic Multi-Device Interaction-based Testing for Web and Android Apps → *Chapter 12*

In order to mitigate the problems regarding multi-device interaction-based testing techniques, we built an open source tool that enables the execution of interactive End-2-End

tests between Android devices and Web applications. This tool, Kraken 2.0, provides an interface for practitioners and researchers to define and execute End-2-End tests based on the interaction of two users while being agnostic of the platform used. Our results in this matter were published in the following articles:

> **Kraken-mobile: Cross-device interaction-based testing of android apps [18]**
> Ravelo-Mendéz, W., **Escobar-Velásquez, C.**, and Linares-Vásquez, M. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019.*

> **Kraken: A framework for enabling multi-device interaction-based testing of Android apps. [19]**
> Ravelo-Mendéz, W., **Escobar-Velásquez, C.**, and Linares-Vásquez, M. In *Science of Computer Programming, 2021.*

> **Kraken 2.0: A platform-agnostic and cross-device interaction testing tool [20]**
> Ravelo-Mendéz, W., **Escobar-Velásquez, C.**, and Linares-Vásquez, M. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.*

> **Kraken 2.0: A platform-agnostic and cross-device interaction testing tool [21]**
> Ravelo-Mendéz, W., **Escobar-Velásquez, C.**, and Linares-Vásquez, M. In *Science of Computer Programming, 2023.*

Using DRL for Automated Testing of Android Apps → *Chapter 13*

Reinforcement Learning (RL), is a machine learning technique that uses positive and negative rewards to find an optimal solution. Specifically for mobile apps, RL approaches can use different aspects of the mobile environment, such as fault-triggering, code coverage, and GUI changes, to guide the learning process towards enhanced exploration and testing of apps. However, state-of-the-art public approaches have focused on testing the individual impact of the aforementioned aspects on the learning process. In this chapter, we present, first a tool, called AgentDroid, that uses a reward function based on a combination of identified-faults and achieved coverage. Second, we present a study focused on comparing the performance of AgentDroid with baseline approaches (*i.e.,* ARES and Monkey).

1.4 Thesis Structure

This document is composed of 15 chapters, organized into eight parts as follows:

Part I: Prologue presents the introductorial chapters.

- **Chapter 1** presents the introduction to the thesis, the problem statement, the main and specific goals of the thesis, the list of contributions of this thesis to state of the art, and the structure of the remaining document.
- **Chapter 2** introduces the thesis context concerning analysis, testing and fixing of open- and closed-source Android Apps. We walkthrough the main concepts and applications of the different mechanisms available to ensure quality, and we discuss the downfalls and missing techniques that could be valuable for developers to continue improving the development process.

Part II: Mutation Testing

- The first section of this part outlines the main concepts and previous work related to Mutation Testing.
- **Chapter 3** presents our study on mutant generation using APKs as input.
- **Chapter 4** depicts our efforts towards enabling mutant selection at APK level.
- **Chapter 5** presents a summary of findings, contributions and open research opportunities created as result of our work regarding mutation testing of Android apps.

Part III: Internationalization of Android Apps

- The first section of this part outlines the main concepts and previous work related to Internationalization of Android Apps.
- **Chapter 6** presents our study on automated detection of i18n issues in Android apps.
- **Chapter 7** depicts our efforts towards enhancing our tool to provide capabilities for reporting and reparing i18n issues in Android app.
- **Chapter 8** presents a summary of findings, contributions and open research opportunities created as result of our work regarding i18n of Android apps.

Part IV: Connectivity Management for Android Apps

- The first section of this part outlines the main concepts and previous work related to Connectivity Management for Android Apps.
- **Chapter 9** presents our empirical study on manual identification of connectivity issues present on Android apps.
- **Chapter 10** depicts our efforts towards generating a tool capable of automatically identify connectivity issues. Additionally, it presents our study validating the capabilities of the aforementioned tool.
- **Chapter 11** presents a summary of findings, contributions and open research opportunities created as result of our work regarding Connectivity Management for Android Apps.

Part V: Automated Testing of Android Apps

- The first section of this part outlines the main concepts and previous work related to Automated Testing of Android Apps.
- **Chapter 12** presents the details of Kraken-Mobile our platform-agnostic multi-device Interaction-based testing tool for Android and Web apps.
- **Chapter 13** presents our first approach towards applying DRL techniques towards the generation of a systematic exploration approach for android Apps.
- **Chapter 14** presents a summary of findings, contributions and open research opportunities created as result of our work regarding Automated Testing for Android Apps.

Part VI: Epilogue concludes this dissertation with the final chapter of this thesis.

- **Chapter 15** presents conclusion remarks, and directions for futurework.

Part VII: Appendix presents complementary information.

- **Chapter A** briefly present a set of studies conducted during the PhD as collaboration with other researchers.
- **Chapter ??** identifies in a standarized manner the contributions performed by the author of the present thesis for each of the chapters composing it.

2

Thesis Background

There is continuous pressure on the developers to prioritize quality and continuous delivery of mobile apps. To achieve this, developers focus their efforts on (i) preventing, (ii) detecting, and (iii) evolving from issues present in the applications. **Prevention** can be done by following a set of good development practices for mobile development, for example: (i) the definition and implementation of architectural/design patterns [22], (ii) correct management of activities/fragments lifecycle [23], (iii) correct usage of libraries and external packages [24], (iv) definition of mechanisms to ensure the security of data [25] and many others. As a mechanism to enable the prevention of issues, manual and automated code reviews can be performed by statically analyzing the source code to identify patterns and code statements breaking the rules of previously mentioned development practices. Additionally, developers can use formal methods to define logical rules (preconditions, postconditions, invariants, and others) that help ensure the correctness of the algorithms.

In contrast, **detection** of issues can be done by following a set of dynamic analysis mechanisms to identify different issues. Examples of these mechanisms are (i) testing, (ii) debugging, and (iii) profiling/monitoring. As mentioned, these processes require the app's execution on emulated or real devices. Specifically, *testing* is focused on validating and verifying the app in terms of its functionalities, while *debugging* and *profiling/monitoring* are used to ensure the quality of internal procedures. Finally, **evolution** of apps is mainly focused on reacting to the identification of an issue. For example, by generating fixes to be applied manually or automatically to reduce the presence of issues within the apps.

As the reader might notice, each of these mechanism provide benefits to the developer and combining them can bring benefits to the development process. This chapter introduces the related literature concerning analysis and testing of open- and closed-source Android Apps. We walkthrough the main concepts and applications of the different mechanisms available to ensure quality, and we discuss the downfalls and missing techniques that could be valuable for developers to continue improving the development process.

Structure of the Chapter

- Section 2.1 presents some context regarding the prevention of issues via the static analysis of android apps. We contextualize the reader with relevant concepts for this mechanism, and summarize the approaches and focuses that have been proposed.
- Section 2.2 summarizes the existing approaches, techniques and mechanisms available for detecting issues in Android Apps. We present a brief definition of the different mechanisms and discuss the missing approaches and new opportunities for research available.
- Section 2.3 briefly discusses the approaches that have been proposed for the evolution/reparing of Android apps. Despite we do not investigate on the improvement of this process, the contributions made in this thesis can be extended to improve the evolving process of android apps.

2.1 Preventing issues in android apps: Static Analysis

2.1.1 Context

According to Boehm and Basili[26] developers can use code reviews and inspections to prevent around 60% of the defects inserted in software during the development phase. These reviews/inspections are done by *statically analyzing* the code. Fortunately, due to the efforts of developers and researchers, this procedure can now be done automatically, and its results are presented to developers directly in IDE or as reports in more thorough analysis processes. Automated *static analysis* of software commonly relies on the usage of Intermediate Representations (IR) and also models generated from the processing of the source code.

Intermediate Representations (IR)

An intermediate representation is a representation of the source code aimed at being more expressive and easier to interpret for a machine. An intermediate representation has the property that, without losing app behavior, it presents the source code in a less complex format or in a less cumbersome context from the point of view of a machine. In the specific case of Android apps, there are several intermediate representations; we will briefly describe four of them, that have been widely used in research: Java Bytecode, Dex, Smali, and Jimple. The first two are the closer representations to the starting and ending point of the Android application building procss, while the other two (even being closer to the endpoint) have been created to enable analysis tasks on the source code.

One of the benefits of using intermediate representations (in the context of JAR and APK analysis) is avoiding the decompilation of the app to reach the original source code; in the specific case of Android apps, by using an intermediate representation, we get rid of reversing the Android building process.

JAVA Bytecode This intermediate representation is used by the Java Virtual Machine (JVM). It is worth noting that JVM follows a stack-based architecture [27]. The most common way to get into this representation is through JAVA itself. However, there are compilers for other languages and frameworks that generate Java Bytecode, such as SCALA, Clojure, Object Pascal, Kotlin and others [28]. As presented previously, Java Bytecode is the first resulting representation when compiling an Android application source code.

Dalvik Bytecode (DEX) Originally designed to work with Dalvik VM (DVM) and the Android Runtime (ART), the dalvik bytecode was designed for systems as mobile devices that have restrictions in their computational power [29, 30]. The Dalvik VM has been replaced by the Android Runtime (ART) keeping the same dalvik bytecode as representation for its execution. It is worth noting that both DVM and ART follow a register-based architecture that normally requires fewer, but typically more complex VM instructions.

Java Source Code

```
startActivity( new Intent(main.this, ImportActivity.class));
```

SMALI representation

```
new-instance v1, Landroid/content/Intent;
iget-object v2, p0, Lio/github/hidroh/materialistic/accounts/AccountAuthenticator;->mContext:Landroid/content/Context;
const-class v3, Lio/github/hidroh/materialistic/ImportActivity;
invoke-direct {v1, v2, v3}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V
invoke-virtual {p0, v3}, Lio/github/hidroh/materialistic/accounts/AccountAuthenticator;->startActivity(Landroid/content/Intent;)V
```

Figure 2.1. SMALI representation example

JIMPLE Jimple is an intermediate representation used by Soot [31]. Jimple is based on 15 different operations that compared with the over 200 operations defined for Java bytecode make it easier to optimize, however, a lot of information can be lost when translated from Java bytecode to Jimple.

SMALI SMALI is an intermediate representation created by Ben Gruver [32]. It offers a readable version of the Dalvik bytecode, due to the similar amount of operations supported for both representations. As it is closer to dalvik bytecode rather than Java bytecode it represents the code following a register-based architecture. This enhances our possibilities to analyze applications because all Java instructions tend to be separated from recursive calls. For example, if a new object is created inside the parameter field of a method, SMALI translates the object creation into one line and the method call into another as it can be seen in Figure 2.1. Because of this, as it will be seen in the next sections, we were able to recognize more potential fault injection points that are more complex to be recognized when analyzed in Java.

2.1.2 Static Analysis of Android Apps: Related Literature

As part of the problem definition, we reviewed articles focused on static analysis of Android apps, with the purpose of identifying the intermediate representations used by researchers. Therefore, we queried publications with the keywords: "static analysis", "android", "source code". Most of the retrieved works focused mainly on security. As part of the results we found a paper called "Static Analysis of Android Apps A Systematic Literature Review" [33] wrote by Li *et. al.* Therefore, instead of conducting a new mapping study or literature review by ourselves, we relied on the work by Li *et. al.*.

The paper by Li *et. al.*, in the authors words, *"provide a clear view of the state-of-the-art works that statically analyze Android apps, from which we [the authors] highlight the trends of static analysis approaches, pinpoint where the focus has been put, and enumerate the key aspects where future researchers are still needed"*. In particular, Li *et. al.,* conducted a systematic

Table 2.1. Keywords used by Li *et. al.*

Line	Keywords
1	Analisis; Analyz*; Analys*;
2	Data-Flow; "Data Flow*"; Control-Flow; "Control Flow"; "Information-Flow*"; "Information Flow*"; Static*; Taint;
3	Android; Mobile; Smartphone*; "Smart Phone";

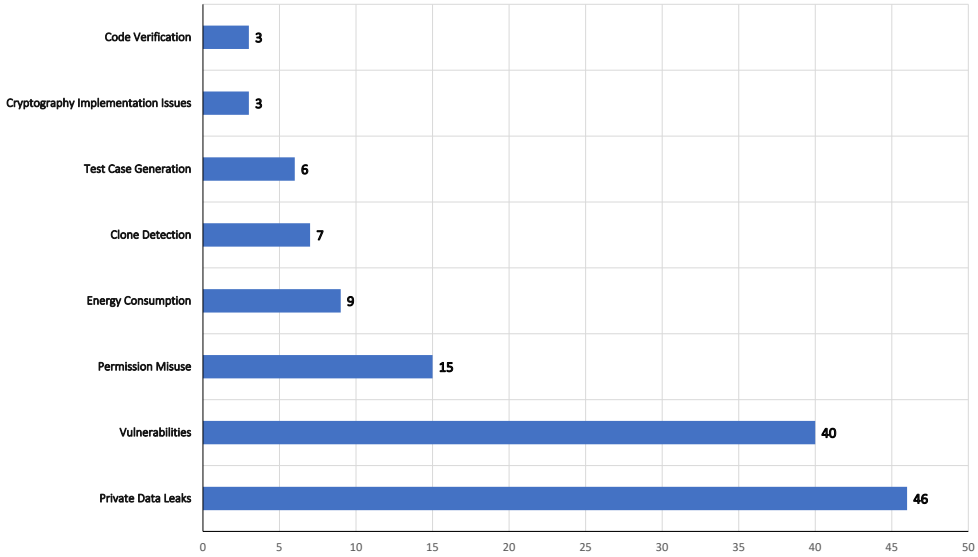


Figure 2.2. Statistics of main concerns addressed by the publications presented in Li *et. al.* publication

literature review using the search string shown in Table 2.1, which reports 124 papers and classifies them in 8 categories depicted in Figure 2.2: (i) Private Data Leaks (46 papers), (ii) Vulnerabilities (40 papers), (iii) Permission Misuse (15 papers), (iv) Energy Consumption (9 papers), (v) Clone Detection (7 papers), (vi) Test Case Generation (6 papers), (vii) Cryptography Implementation Issues (3 papers) and (vii) Code verification (3 papers).

Note that the systematic literature review by Li *et. al.*, considers only papers up to 2015, and we recognize that since then several works could have been published. Our purpose with reviewing previous papers was to identify the approaches proposed towards static analysis of android applications. From the identified research gaps, we selected a set of techniques that were not addressed until that moment, and analyze the related work generated from 2015 until the design process started. Therefore, in the following sections we will review the most recent works for each one of the selected testing techniques.

In Fig. 2.2, we depict the task-related groups used by Li *et. al.*, and their distribution in terms of the goal addresses within the articles.

Private Data Leaks. This is the most frequent purpose reported in the papers categorized

by Li *et. al.,*. In this group, FlowDroid is a representative example [34]; FlowDroid performs static taint analysis on Android apps using flow-, context-, field-, object-sensitive and implicit flow-, lifecycle-, static-, alias-aware analysis. Therefore, FlowDroid has became a defacto tool used by researchers interested on finding privacy leaks in Android apps. For example, PCLeaks [35] goes one step further by performing sensitive data-flow analysis on top of component vulnerabilities, enabling not only issue identification but also data endangered. The most used intermediate representation for this category of papers is JIMPLE with 18 out of 46 papers, followed by SMALI with 8 papers.

Vulnerabilities. This category groups papers aiming at detecting vulnerabilities in Android apps. For instance, CHEX[36] detects potential component hijacking-based flows through reachability analysis on customized system dependence graphs and, Epicc [37] and IC3 [38] that implement static analysis techniques for implementing detection scenarios of inter-component vulnerabilities. The most used intermediate representation for this category of papers is SMALI with 15 out of 40 papers, followed by JIMPLE with 6 papers.

Permission Misuse. Permissions are one of the core elements of the Android security model. Malware applications try to use the permissions granted by the user to perform actions that do not correspond to the app features. Lin *et al.* [39] conducted an study of permissions that users are most comfortable to grant, creating a set of privacy profiles, and in which way applications use those permissions. The most used intermediate representation for this category of papers is JIMPLE with 6 out of 15 papers, followed by SMALI with 4.

Energy Consumption. APIs and some hardware components have been demostrated as energy greedy elements in Android apps [40, 41], thus, analysis of energy consumption of mobile apps is becoming a hot topic. For instance, Li *et al.* [42] present a tool to calculate source line level energy consumption through combining program analysis and statical modeling. The output of these analyses can then be leveraged to perform quantitative and qualitative empirical investigations into the categories of API calls and usage patterns that exhibit energy consumptions profiles. The most used IR for this category of papers is JAVA_CLASS with 6 out of 9 papers, followed by JIMPLE with 2.

Clone Detection. It is also well known that there are some circumstances that lead app users to use APK repositories different from Google Play, which generates a concern about the origin and provenance of Android apps in general. Therefore, approaches such as DNADroid[43] uses neural networks and dinamic-, static analysis to propose detection of ransomware before infection happens. At the same time, Crusell *et. al.,* [44] propose a scalable to detecting similar Android Apps based on their semantic information. The most used intermediate representation for this category of papers is SMALI with 3 out of 7 papers, followed by DEX_ASSEMBLER with 2.

Test Case Generation. A common way to perform analysis of an application is using systematic exploration, nevertheless running real world applications in real devices is cumbersome due to several problems like non-determinism, non-standard control flow, etc. Because of this, A3E[45] uses static, taint-style, dataflow analysis on the app bytecode to construct a higher level flow-graph that captures legal transition among activities, and can be used to explore the app in a user-like behavior. At the same time, Jensen *et al.*[46] propose a two-phase technique that uses concolic execution to build summaries of the event handlers

of the application and builds event sequences backward from the target, enabling the testing of parts that require more complex event sequences. The most used intermediate representation for this category of papers is JIMPLE with 3 out of 6 papers, followed by WALA_IR and SMALI with 1 paper each one.

Code Verification. Code verification intends to ensure the correctness of a given app but without testing (*i.e.,* app execution). For instance, Cassandra [47] is proposed to check whether Android apps comply with their personal privacy requirements before installing an app. As another example, researchers have also extended the Julia [48] static analyzer to perform code verification through formal analyses of Android programs. The most used intermediate representation for this category is JAVA_CLASS with 2 papers.

Cryptography Implementation Issues. In addition to the aforementioned concerns, state-of-the-art works have also targeted cryptography implementation issues. As an example, CMA [49] performs static analysis on Android apps and select the branches that invoke the cryptographic API. Then it runs the app following the target branch and records the cryptographic API calls. At last, the CMA identifies the cryptographic API misuse vulnerabilities from the records based on the pre-defined model. It is worth noticing that the top representations used by the reported papers are JIMPLE (38 papers), SMALI (26 papers) and JAVA_CLASS (22 papers), and the main concern covered is security with around 101 papers.

As it can be seen, static analysis of Android apps is mostly focused on security and code quality assurance. However, there were no approaches toward the assurance of quality attributes such as eventual connectivity.

Additionally, it is important to highlight that static analysis of android apps rely on the usage of intermediate representations (IR), for example, Java bytecode, which is the closest IR to source code. Since we are interested in evaluating the feasibility of using APKs as input for the execution of SE tasks, we decided to extend our search to identify an IR that preserves the app's functionalities when decoded and whose structure provides us with mechanisms to generate models and analyze it.

Therefore, as we wanted to use an intermediate representation that is closer to the compiled code, we have to choose between JIMPLE and SMALI. Because of this, we extended our research to find existing studies aimed at comparing these two intermediate representations. After searching in google scholar, we found a paper by Arnatovich *et. al.,*[50] called *Empirical Comparison of Intermediate Representations for Android Applications* in which they study the capabilities of the different IRs to preserve the program behavior, by *disassembling, assembling, signing, aligning and installing* 520 applications selected from the Google Play Store. The way they studied this was by running a random GUI-based input generation program (*i.e., monkey runner*) over each app before and after the designed process[1] and collecting the number of apps that crashed. Using this result, they were able to identify the number of apps that do not crash after this process. The results of the study are summarized in Table 2.2

Consequently, SMALI appears to be the IR that preserves most of the times the programs behavior after decoding process and it is close enough to APK level to avoid decompilation of input files. Based on this, and recalling the research gap on static analysis of android

[1]The monkey runner generates a seed that when given as parameter replicates the same events.

Table 2.2. Comparision of program behavior preserveness for SMALI, JIMPLE and JASMIN

Intermediate Representation	Preserved Program Behaviors of Original Applications (%)
SMALI	97.68
JIMPLE	85.58
JASMIN	81.92

apps, several techniques and mechanisms should be proposed to evaluate the feasibility of implementing and executing SE tasks at APK level. Specifically, we focus on mutation testing, internationalization and interaction-based testing.

2.2 Detecting issues in android apps: Dynamic Analysis

2.2.1 Context

Currently, all proposed approaches and efforts towards quality assurance of android apps can be categorized as static or dynamic analysis of the Application Under Test (AUT). We have studied and analyzed the static approaches in the previous section. Now, when considering dynamic approaches, there are two main categories: (i) testing and (ii) debugging that cover the proposed approaches to dynamically analyzed an AUT. Testing relies on the execution of a set of cases/scenarios that excercise a complete interaction flow with the app or its components, while debugging focus on providing practitioners and developers with the capability of executing the AUT step by step at source-code level in order to analyze the values and behaviour of the AUT on each step of an execution scenario. For this thesis we focus on studying and analyzing the efforts provided to **test** AUT since our focus is the automatization of the process, reducing then the interaction of practitioners and developers along the process.

2.2.2 Android App Testing

Regarding testing techniques, there are several categories of approaches that can be group based on the type of interaction and mechanism used to explore/execute the aplication. Linares-Vásquez *et al.* [7] presents a categorization of this techniques in 7 categories: (i) Automation Frameworks and APIs, (ii) Record and Replay tools, (iii) Automated GUI-Input Generation tools, (iv) Bug and Error Reporting/monitoring tools, (v) Testing Services, (vi) Cloud Testing Services and (vii) Device Streaming tools.

Automation Frameworks and APIs provide a mechanism for developers to define via code statements interactions with visual and logical components, due to this, developers are required to understand the capabilities of the approaches and must identify the best way to define each interaction. Examples of this APIs are UIAutomator[51], Espresso[52], Appium[53], Robotium[54] among others[55, 56]. *Record and Replay tools* provide practitioners with an additional capabily since these tools/approaches record a usage scenario executed by a

practitioner and translates the interaction into scripts (*i.e.,* code representation) to then be replayed in future opportunities. This type of approaches ease the testing process but can generate layout dependent scripts that rely on the position of elements when the different components do not have ids defined during the development process. Examples of this approaches are Espresso Test Recorder [57], RERAN [58], VALERA [59], MOSAIC [60], among others.

Automated GUI-Input Generation tools consist of approaches that extract representations of the AUT and device in order to automatically generate inputs over the app. Some of this approaches generate random GUI-events to test the behaviour of the app in such conditions, this type of event generation migth generate a large amount of invalid events but leads to the evaluation of extreme cases that are not commonly taken into account such as double pressing a button with a critical task. An example of this is the Monkey Exerciser[61] that belongs to Android SDK. Based on this approches, Systematic explorers, extract representations of the UI and select random events over the identified components, this reduces the rate of invalid events but increases the time required to executed this approaches, an example of these approaches is Google Robotest[62]. Remaining categories mentioned by Linares-Vásquez *et al.* are not mentioned in this section since none of these impact or is used in the development of this thesis.

Based on the mentioned techniques different practitioners, researchers and developers have created tools that tackle different quality attributes and behaviours. Taking into account the results published by Kong *et al.* [63], after analyzing 103 publications, the main concerns addressed by existing tools/approaches are *Concurrency* [64, 65, 66], *Security* [67, 68, 69], *Performance* [70, 71, 72], *Energy* [73, 74, 75], *Compatibility* [76, 77, 78], and *Bug/Defects Identification* [79, 80, 81]

2.3 Fixing issues in android apps

Automated Program Repair (APR) is one of the procedures that have benefited from the several approaches and tools created for static and dynamic AUT analysis. The main objective of this task is to provide solutions to known issues either within the app being developed or based on benchmarks and studies targeting Android apps. In order to achieve this, approaches must generate modifications to the current state of the source code to remove or prevent the injection of faults. The fundament of these fixes comes from the generation of a knowledge base extracted from empirical studies; for example, Linares-Vásquez *et al.* [82] identified standard practices to detect and fix performance bottlenecks in Android apps; they surveyed 485 developers and analyzed the GitHub repositories in which the interviewees contributed to study the process followed to report and fix performance issues. Based on this knowledge base, new approaches could be generated to identify code statements that generate performance issues automatically.

Based on our contributions, we performed an initial search regarding automated fixing to set the ground for our proposed approach. This thesis mentions fixing issues but does not focus on studying their impact and evaluating the capabilities of such approaches.

After an initial search for approaches focused on fixing, we found that the existing ap-

proaches tackled different topics. First, automated bug fixing, in which, based on the result of test suites, approaches generate patches that can be applied to the source code of the apps to fix the issue. [83, 84, 85, 86, 87, 88]. Other topics tackled by researchers is fixing compatibility issues, in which approaches focus on solving issues identified due to particular characteristics of OS versions. The proposed approaches use combinations of static and dynamic analysis to identify the issues, based on these, propose a set of patches that enhance the quality of the app to support a more extensive set of devices and OS versions [89, 90, 91]. Researchers have also worked on repairing issues related to user-generated data during execution; Guo *et al.* [92] presents an approach that generates patches to help ensure data is correctly handled during the app execution; to identify activities and fragments that allow the user to generate data and propose the usage of the correct life-cycle methods to save and retrieve data.

Part II

Mutation Testing of Android Apps

Structure of Part II: Mutation Testing

- First section outlines the main concepts and previous work related to Mutation Testing.
- **Chapter 3** presents our study on mutant generation using APKs as input.
- **Chapter 4** depicts our efforts towards enabling mutant selection at APK level.
- **Chapter 5** presents a summary of findings, contributions and open research opportunities created as result of our work regarding mutation testing of Android apps.

Mutation testing is a testing technique that consists of modifying an application (by injecting bugs) in order to enhance and evaluate the quality of the test suite that accompanies it. Each injected bug generates a new version of the application, and it's called **mutant**. Each mutant differs from the original version in a simple modification, called **mutation**. As an example, if there is a compound logical operation in the original version, a valid mutation consists in replacing one and only one of the logical operators in the compound operation (i.e. if there is an "AND", the mutation changes it with "OR"). It is worth noticing, that the replacement of each operator in the compound operation generates one mutant.

Now, to determine how many mutants can be generated, mutation testing uses a set of rules called **mutation operators** that define common errors and practices that belong to programming rules (e.g., replace a math operator), or to the specific programming language context (e.g., assign a null value to an Intent parameter). Therefore, depending on the tool used to generate the mutants, the application to be mutated is analyzed to derive a Potential Fault Profile [93, 94], i.e., a set of possible locations where a mutant operator can be applied. For each of these locations, the mutation tool creates a copy of the original app, and this copy is a modification produced by applying a mutation operator. Several mutation operators have been proposed for different types of applications such as web apps [95], data-centric apps [96], NodeJS packages [97] and Android apps [93]. For more information about mutation testing you can refer to Chapter 5 of Paul Ammann and Jeff Offutt book called Introduction to Software Testing [98], or the survey by Jia and Harman [99].

Regarding mutation testing for Android Apps, previous work can be divided in two main groups: (i) approaches available for Java programs, and (ii) Android specific mutation testing approaches. In the case of Java-based approaches, there proposed approaches generate mutants at source-code and bytecode level. The most representative tools are PIT[100], Jester[101], μJava [102], Major[103], Jumble [104] and Javalanche [105]. However, since these approaches are designed for java applications, they lack of mutation operators that take into account android specifics, then the generated mutants are not as valuable as expected.

In the case of android specific mutation testing approaches, there are approaches that work at source-code and bytecode level. First, Linares-Vásquez *et. al.,*[93, 94] implemented a mutant generation tool at source code level, MDroid+. They empirically extracted a taxonomy of crashes/bugs in Android apps, and based on that, then proposed a set of 38 mutation operators. At the same time, Deng *et al.* [106], presented a set of eight mutant operators oriented to mutate core components of Android (*e.g., intents, event handlers, XML files and activity lifecycle*). Later, Deng *et al.*[107], presented the implementation of the proposed mutation operators. Another existing approach is called muDroid[108] a mutation testing

tool that works at APK level, but it implements standard mutation operators (mutant operators whose rule can be applied to software applications in general, for example, replace a logical operator by its negation). However, MDroid+ authors [93, 94] found that muDroid generates around 53% of non-compilable mutants, that can be translated into a lost of half of the time invested on executing muDroid. Paiva *et al.* [109] propose 3 mutation operators aimed at validating the preservation of user's data and UI state after an application has changed from background to foreground. Finally, Liu *et al.* [110] present DroidMutator, a mutant generation tool aimed to reduce the generation of non-compilable mutants by using Type checking, their tool was compared wit MDroid+ and it reports the reduction of non-compilable mutants generation from 1.7% of MDroid+ to 0.1%. Additionally, it claims to be 5 times faster in the generation process.

3

Enabling mutant generation for open-and closed-source android apps

Mutation testing has been widely used to assess the fault-detection effectiveness of a test suite, as well as to guide test case generation or prioritization. Empirical studies have shown that, while mutants are generally representative of real faults, an effective application of mutation testing requires "traditional" operators designed for programming languages to be augmented with operators specific to an application domain and/or technology.

The case for Android apps is not an exception. Therefore, in this chapter we describe the process we followed to create (i) a taxonomy of mutation operations and, (ii) two tools, `MDroid+` and `MutAPK` for mutant generation of Android apps. To this end, we used the taxonomy of 262 types of Android faults devised by Linares-Vásquez [93] and the set of mutant operators proposed to implement its corresponding version at APK level within a tool called `MutAPK`.

The rationale for having a second approach at APK level is based on the fact that source code is not always available when conducting mutation testing. Thus, mutation testing for APKs enables new scenarios in which researchers/practitioners only have access to APK files. `MutAPK` have been evaluated in terms of the number of non-compilable, trivial, equivalent, and duplicate mutants generated and their capacity to represent real faults in Android apps as compared to other well-known mutation tools.

Structure of the Chapter

- Section 3.1 provides motivation for this chapter.
- Section 3.2 contextualizes the reader with instrumental background concepts.
- Section 3.3 presents the architecture of the two tools used for the study
- Section 3.4 presents the design of our study, as well as the data extraction procedure and analysis methodology.
- Section 3.5 discusses our results and findings.
- Section 3.6 presents the threats that affect the validity of our work.

Supplementary Material

All the data used in this chapter as well as our tool are publicly available. More specifically, we provide the following items:

Online Appendix [111] The online appendix includes the following material:

- **Source Code.** `MDroid+` and `MutAPK` source code is available as Gitlab and GitHub repositories.
- **Taxonomy.** Taxonomy of faults identified on Android apps.
- **List of Mutant Operators.** Complete list of mutant operators with examples of java and SMALI code snipepts.
- **List of Applications.** List of Android applications assessed using `MDroid+` and `MutAPK`
- **Empirical Study.** Additional information supporting the results presented in Section 3.4

Publications and Contributions

Enabling mutant generation for open-and closed-source android apps. [11] **Escobar-Velásquez, C.**, Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Cárdenas, C., and Poshyvanyk, D. *IEEE Transactions on Software Engineering, 2020.*

MutAPK: Source-codeless mutant generation for android apps. [12] **Escobar-Velásquez, C.**, Osorio-Riaño, M., and Linares-Vásquez, M. In *Proceeding of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019.*

MutAPK → Section 3.3
We present `MutAPK`, an open source mutation testing tool that enables the usage of Android Application Packages (APKs) as input for this task. `MutAPK` generates mutants without the need of having access to source code, because the mutations are done in an intermediate representation of the code (*i.e.,* SMALI) that does not require compilation.

3.1 Introduction

This chapter aims to deal with the lack of (i) an analysis of the applicability of state-of-the-art mutation tools on Android apps, and (ii) a characterization of the pros and cons of conducting mutant generation at source code or APK level.

Based on the proposed android-specific operators and the implemented tool for android mutation at source-code level, `MDroid+`, we conceived and implemented a framework for mutant generation of Android apps, `MutAPK`, at APK-level. Our rationale for having two different tools is based on the fact that source code is not always available (*e.g.,* as in the case of outsourced testing services). In addition, we wanted to identify the pros and cons of generating mutants of Android apps at source-code and APK levels. Both tools are publicly available [112, 113].

In addition, we conducted a study comparing `MDroid+` and `MutAPK` with other Java and Android-specific mutation tools. The study results indicate that both `MDroid+` and `MutAPK`, as compared to existing competitive tools, (i) can cover a larger number of bug types/instances present in Android apps; (ii) are highly complementary to the existing tools in terms of covered bug types; and (iii) generate fewer trivial, non-compilable, equivalent and duplicate[1] mutants. When comparing source-code vs. APK level mutation, we found that both mutation and compilation/assembling are performed quicker at APK level than at source code level, but with a lower quality of generated mutants. Our experiments show an improvement of 93.83% (*i.e.,* 4.32 from 4.61 seconds) in the mutation time and 87.05% (*i.e.,* 174.73 from 195 seconds) for compilation/assembling times. However, the source code-based mutation approach generates only 2.97% (*i.e.,* 263 of 8847 mutants) of non-compilable or trivial mutants as compared to the 6.8% (*i.e.,* 5105 of 75053 mutants) of the APK-based mutation approach. It is worth noting that our study does not conduct mutation testing (*i.e.,* executing test suites), since no test suite is available for the selected app dataset.

3.2 Context

This chapter presents an extension of a paper published at the 11*th* Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17) [93]. Linares-Vásquez *et al.* produced a taxonomy of Android faults by analyzing a statistically significant sample of 2,023 candidate faults documented in (i) bug reports from open source apps; (ii) bug-fixing commits of open source apps; (iii) Stack Overflow discussions; (iv) the Android exception hierarchy and APIs potentially triggering such exceptions; and (v) crashes/bugs described in previous studies on Android. Fig. 3.1 presents the taxonomy that contains 262 types of faults grouped in 14 categories, four of which relate to Android-specific faults, five to Java-related faults, and five mixed categories[2]. Then, leveraging this fault taxonomy and focusing on Android-specific faults, Linares-Vásquez *et al.* devised a set of 38 Android mutation operators and created their corresponding mutation rules for Java code.

[1] We use the definition provided by Papadakis*et al.* [114]: two mutants that are equivalent to each other are called *duplicate mutants*

[2] The details of the process done to built the taxonomy can be found in the original article[93]

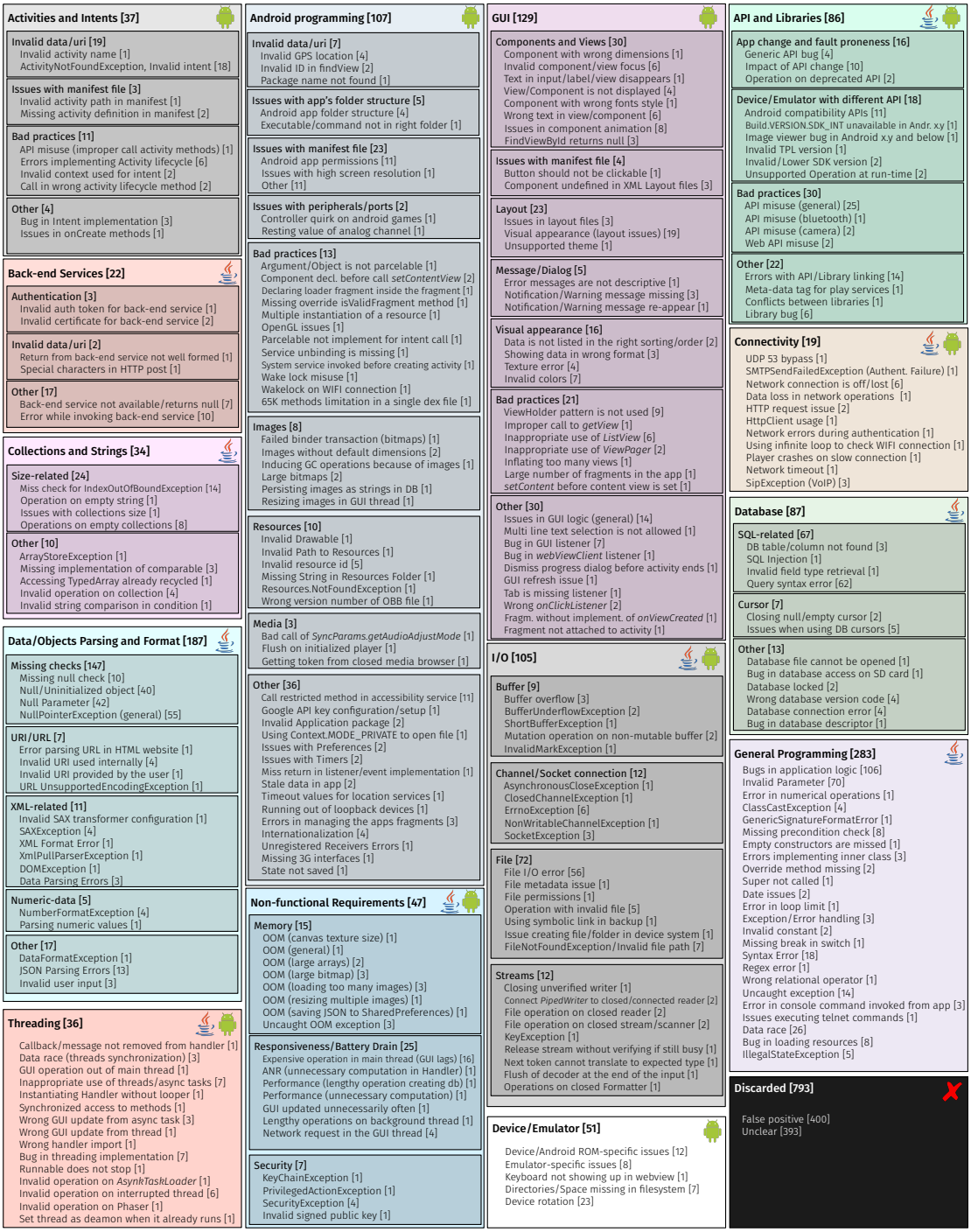


Figure 3.1. The defined taxonomy of Android bugs.

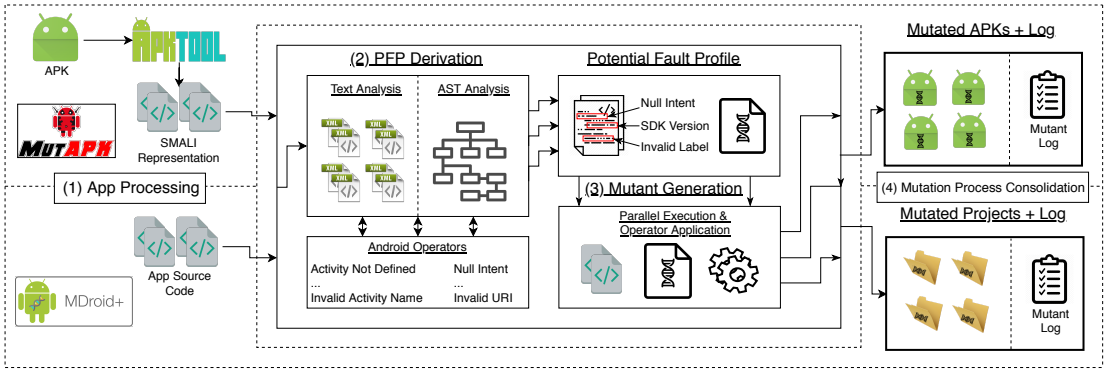


Figure 3.2. Overview of `MDroid+` and `MutAPK` workflows.

3.3 Tools Architecture

To ensure that `MDroid+` and `MutAPK` are effective, practical, and flexible/extensible tools for mutation testing, both tools take into account the following design considerations:
(i) an empirically derived set of mutation operators; (ii) a design embracing the open/closed principle (*i.e.,* open to extension, closed to modification); (iii) visitor and factory design patterns for deriving the Potential Failure Profile (PFP) and applying operators; (iv) parallel computation for efficient mutant seeding. Both tools are written in Java and are available as open source projects [112, 113].

Fig. 3.2 presents an overview of the workflow for both tools. There are four main stages for both tools: First, **App Processing** where `MutAPK` requires the *APKTool* [115] library to decode an app to get an intermediate representation of the compiled code and resources. In this stage, `MDroid+` requires the source code and resources folder of a given app (*e.g.,* the `/res/` folder).

Second, **PFP Derivation** consisting in two processes: (i) resource files processing: by taking advantage of the structure provided in given files (*i.e.,* XML files, resource files), to identify XML tags that match the different operators either by its tag name (*e.g.,* WrongStringResource that search for `<string>` tags) or tag attributes (*i.e.,* InvalidLabel that search for `android :label` in Manifest's tags), and (ii) Code-related files: by pattern matching API calls. In `MDroid+` case, the JDT Core DOM Library is used to generate an AST representation from JAVA code and matches AST's nodes with API call templates defined in a file called *target-apis.properties*. In contrast, `MutAPK` uses *Antlr*, *JFlex* and *GAP* libraries, to generate the AST given it works with SMALI code. Nevertheless, since SMALI code uses a larger set of instructions to represent a JAVA instruction, a larger set of API calls must be matched for each operator.

Third, **Mutant Generation** is performed based on the previously-generated *PFP* and the catalog of implemented operators (explained in Section 3.3.1). Therefore, using the mutation rules, for each location in the *PFP* a copy of the app processing result is generated and modified. To provide the most efficient process, both tools allow users to parallelize the generation process, utilizing the multi-core architecture of most modern hardware.

Finally, the fourth stage is the **Mutation Process Consolidation**: `MutAPK` generates for

each mutant an APK using the *APKTool* [115] and signs it with the *Uber APK Signer* [116]; MDroid+ stores the mutated source code folder. Finally, both tools generate a log file for the mutation process result.

3.3.1 Implemented Mutation Operators

One of the main components in the mutation process is the set of mutation operators that define the correct way to represent a naturally occurring fault in an Android project. Specifically, in this study we define 38 mutation operators that belong to 10 of the 14 categories extracted in the previous taxonomy (*i.e.,* Fig. 3.1). As it can be seen in Table 3.1, MDroid+ implements 35 operators while MutAPK implements 34. It worth noticing that 32 of these implemented operators are shared between tools.

For example, a *Missing Permission on Manifest file* could be a fault likely to be found in an Android Project. Therefore, both MDroid+ and MutAPK have an operator called *MissingPermissionManifest* that, given a permission on the manifest file, remove it from the file by replacing the complete line with a blank space.

Listing 3.1. Permission on Manifest file

```
1 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Another mutation operator we defined is *DifferentActivityIntentDefinition* where, given an intent declaration (Listing 3.2), MDroid+ replaces the Activity.class argument in the intent instantiation with the Activity.class value of another class belonging to the project (Listing 3.3).

Listing 3.2. Intent instantiation Java

```
1 Intent intent = new Intent(main.this, ImportActivity.class);
```

Listing 3.3. MDroid+ operator result

```
1 Intent intent = new Intent(main.this, ExportActivity.class);
```

MutAPK also provides this operator. However, since it works at the APK level, the definition of the mutation rule is in terms of SMALI representation. Therefore, the intent instantiation seen in Listing 3.2 is represented in SMALI as it can be seen in Listing 3.5. Moreover, the mutation result seen in Listing 3.3 is represented as it is shown in Listing 3.5.

Listing 3.4. SMALI Intent instantiation Java

```
1 const-class v3, Lcom/fsck/k9/activity/ImportActivity;
2 invoke-direct {v1, v2, v3}, Landroid/content/Intent;-> <init>(Landroid/content/Context;Ljava/lang
      /Class;)V
```

Listing 3.5. MutAPK operator result

```
1 const-class v1, Lcom/fsck/k9/activity/ExportActivity;
2 invoke-direct {v1, v2, v3}, Landroid/content/Intent;-> <init>(Landroid/content/Context;Ljava/lang
      /Class;)V
```

Table 3.1. Proposed mutation operators. The table lists the operator names, detection strategy (AST or TEXTual), the fault category (Activity/Intents, Android Programming, Back-End Services, Connectivity, Data, DataBase, General Programming, GUI, I/O, Non-Functional Requirements), a brief operator descriptions, and if it is implemented in `MDroid+` and `MutAPK`.

Mutation Operator	Det.	Cat.	Description	MDroid+	MutAPK
ActivityNotDefined	Text	A/I	Delete an activity <android:name="Activity"/> entry in the Manifest file	✓	✓
DifferentActivityIntentDefinition	AST	A/I	Replace the Activity.class argument in an Intent instantiation	✓	✓
InvalidActivityName	Text	A/I	Randomly insert typos in the path of an activity defined in the Manifest file	✓	✓
InvalidKeyIntentPutExtra	AST	A/I	Randomly generate a different key in an Intent.putExtra(key, value) call	✓	✓
InvalidLabel	Text	A/I	Replace the attribute "android:label" in the Manifest file with a random string	✓	✓
NullIntent	AST	A/I	Replace an Intent instantiation with null	✓	✓
NullValueIntentPutExtra	AST	A/I	Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable[0]	✓	✓
WrongMainActivity	Text	A/I	Randomly replace the main activity definition with a different activity	✓	✓
MissingPermissionManifest	Text	AP	Select and remove an <uses-permission /> entry in the Manifest file	✓	✓
NotParcelable	AST	AP	Select a parcelable class, remove"implements Parcelable" and the @override annotations	✓	x
NullGPSLocation	AST	AP	Inject a Null GPS location in the location services	✓	✓
SDKVersion	Text	AP	Randomly mutate the integer values in the SdkVersion-related attributes	✓	✓
WrongStringResource	Text	AP	Select a <string /> entry in /res/values/strings.xml file and mutate the string value	✓	✓
NullBackEndServiceReturn	AST	BES	Assign null to a response variable from a back-end service	✓	✓
BluetoothAdapterAlwaysEnabled	AST	C	Replace a BluetoothAdapter.isEnabled() call with"true"	✓	✓
NullBluetoothAdapter	AST	C	Replace a BluetoothAdapter instance with null	✓	✓
InvalidURI	AST	D	If URIs are used internally, randomly mutate the URIs	✓	✓
ClosingNullCursor	AST	DB	Assign a cursor to null before it is closed	✓	✓
InvalidIndexQueryParameter	AST	DB	Randomly modify indexes/order of query parameters	✓	✓
InvalidSQLQuery	AST	DB	Randomly mutate a SQL query	✓	✓
InvalidDate	AST	GP	Set a random Date to a date object	✓	✓
InvalidMethodCallArgument	AST	GP	Randomly mutate a method call argument of a basic type	x	x
NotSerializable	AST	GP	Select a serializable class, remove "implements Serializable"	✓	x
NullMethodCallArgument	AST	GP	Randomly set null to a method call argument	x	✓
BuggyGUIListener	AST	GUI	Delete action implemented in a GUI listener	✓	x
FindViewByIdReturnsNull	AST	GUI	Assign a variable (returned by Activity.findViewById) to null	✓	✓
InvalidColor	Text	GUI	Randomly change colors in layout files	✓	✓
InvalidIDFindView	AST	GUI	Replace the id argument in an Activitity.findViewById call	✓	✓
InvalidViewFocus	AST	GU	Randomly focus a GUI component	x	✓
ViewComponentNotVisible	AST	GUI	Set visible attribute (from a View) to false	✓	✓
InvalidFilePath	AST	I/O	Randomly mutate paths to files	✓	✓
NullInputStream	AST	I/O	Assign an input stream (*e.g.,* reader) to null before it is closed	✓	✓
NullOutputStream	AST	I/O	Assign an output stream (*e.g.,* writer) to null before it is closed	✓	✓
LengthyBackEndService	AST	NFR	Inject large delay right-after a call to a back-end service	✓	✓
LengthyGUICreation	AST	NFR	Insert a long delay (*i.e.,* Thread.sleep(..)) in the GUI creation thread	✓	✓
LengthyGUIListener	AST	NFR	Insert a long delay (*i.e.,* Thread.sleep(..)) in the GUI listener thread	✓	✓
LongConnectionTimeOut	AST	NFR	Increase the time-out of connections to back-end services	✓	✓
OOMLargeImage	AST	NFR	Increase the size of bitmaps by explicitly setting large dimensions	✓	✓

3.4 Study Design

The *goal* of this study was to: (i) understand and compare the ***applicability*** of MDroid+, MutAPK, and other currently available mutation testing tools; (ii) understand the ***underlying reasons*** for mutants generated by these tools that cannot be considered useful, *i.e.,* non-compilable mutants, mutants that cannot be launched, mutants that are equivalent to the original app, and mutants that are duplicate; and (iii) understand the pros and cons of conducting mutation testing at source code and APK level.

The study addressed the following research questions[3]:

- **RQ_2:** *What is the rate of non-compilable (*e.g., *those leading to failed compilations), trivial (*e.g., *those leading to crashes on app launch), equivalent, and duplicate mutants produced by the studied tools when used with Android apps?*
- **RQ_3:** *What are the major causes for non-compilable, trivial, equivalent, and duplicate mutants produced by the mutation testing tools when applied to Android apps?*
- **RQ_4:** *What are the benefits and trade-offs of performing mutation testing at APK level vs. source code level?*

To answer **RQ_2, RQ_3** and **RQ_4**, we compared the tools based on different mutation testing metrics. In particular, we compared Major, PIT, muDroid, `MDroid+`, and `MutAPK`. Major and PIT are popular open source mutation testing tools for Java, that can be tailored for Android apps. The tool by Deng *et al.* [117] is a context-specific mutation testing tool for Android available at GitHub. We chose these tools because of their diversity (in terms of functionality and mutation operators), their compatibility with Java, and their representativeness of tools working at different representation levels: source code, Java bytecode, and SMALI (*i.e.,* Android-specific bytecode representation). Jabbarvand and Malek[118] present a tool called μDroid that generates mutants to validate the energy usage of apps. However, this tool is only compatible with Eclipse, precluding it from being used with the large set of apps collected for our empirical evaluation.

To compare the applicability of each mutation tool, we need a set of Android apps that meet certain constraints: (i) the source code of the apps must be available, (ii), the apps should be representative of different categories, and (iii) the apps should be compilable (*e.g.,* including proper versions of the external libraries they depend upon). For these reasons, we use the Androtest suite of apps [119], which includes 68 Android apps from 18 Google Play categories. These apps have been previously used to study the design and implementation of automated testing tools for Android and met the three above listed constraints. The mutation testing tools exhibited issues in 13 of the considered 68 apps, *i.e.,* the 13 apps did not compile after injecting the faults. Thus, in the end, we considered 55 subject apps in our study. The list of considered apps as well as their source code and APKS is available in our replication package [111].

[3]RQ1 is not mentioned in this section since it was proposed in the original FSE publication and our research did not extend this. The information regarding this questions is presented in Section 3.2

Note that while Major and PIT are compatible with Java applications, they cannot be directly applied to Android apps. Thus, we wrote specific wrapper programs to perform the mutation, the assembly of files, and the compilation of the mutated apps into runnable Android application packages (*i.e., APKs*). While the procedure used to generate and compile mutants varies for each tool, the following general workflow was used in our study: (i) generate mutants by operating on the original source/byte/SMALI code using all possible mutation operators; (ii) compile or assemble the APKs either using the `ant`, `dex2jar`, or `baksmali` tools; (iii) run all of the apps in a parallel-testing architecture that utilizes Android Virtual Devices (AVDs); (iv) collect data about the number of apps that crash on launch and the corresponding exceptions of these crashes, which will be utilized for a manual qualitative analysis; and (v) compute the number of equivalent and duplicate mutants. We refer readers to our replication package for the complete technical methodology used for each mutation tool [111].

To quantitatively assess the applicability and effectiveness of the considered mutation tools to Android apps, we used five metrics: **Total Number of Generated Mutants (TNGM)**, **Non-Compilable Mutants (NCM)**, **Trivial Mutants (TM)**, **Equivalent Mutants (EM)**, and **duplicate Mutants (DM)**. Additionally, we analyzed the time required by both `MutAPK` and `MDroid+` to: (i) generate a mutated copy of the app and to (ii) compile/build the copy.

In this paper, we consider *Non-Compilable Mutants* as those that are syntactically incorrect to the point that the APK file cannot be compiled/assembled, and *trivial mutants* as those that are exhibited when launching the app. If a mutant crashes upon launch, we consider it as a *trivial mutant* because it could be detected by any test case that starts the app. Note that we use the term “Non-Compilable Mutants (NCM)” as a synonym of still-born mutants.

Two other metrics that one might consider to evaluate the effectiveness of a mutation testing tool is the number of *equivalent* and *duplicate mutants* the tool produces. However, in past work, the identification of equivalent mutants has been proven to be an undecidable problem [120, 121], and both equivalent and duplicate mutants require the existence of test suites (not always available and sufficiently complete for this purpose in the case of the Androtest apps).

Papadakis *et al.* [114] proposed a method to overcome the lack of test suites and to reduce the computational time required to detect equivalent and duplicate mutants by relying on proxies computed at the machine code level. Note that this idea has also been explored previously by Offutt *et al.* [122] and Kintis *et al.* [123].

In particular, Papadakis *et al.* [114] propose using “Trivial Compiler Equivalence (TCE)", which relies on comparing compiled machine code to detect equivalence between (i) mutants and original programs, and (ii) among mutants to detect duplicated ones. TCE has been shown to detect, on average, 30% of equivalent mutants [114] on a benchmark of 18 small/medium C/C+ programs [124].

We used TCE to compute equivalent and duplicate mutants at APK level. Instead of doing binary comparisons, we computed hashes, which is also proposed by Papadakis *et al.* [114] as an alternative for the comparisons. Because mutations of Android apps can be applied on source code, manifest files, or resource files (*i.e.,* XMLs), for each original APK and generated mutants, we computed four different hashes.

Given and *APK* file under analysis, we computed:

1. $H(APK)$: hash of the whole apk file;
2. $H(APK_{resources})$: hash of the resource files in the *APK* file, which is computed as the concatenation of hash for each resource file;
3. $H(APK_{manifest})$: hash of the manifest file in the *APK* file;
4. $H(APK_{SMALI})$: hash of the SMALI files extracted from the *APK* file, which is computed as the concatenation of hash values for each SMALI file.

To detect equivalent mutants, we compared the four hashes of an original APK, with the four hashes of each of the generated mutants. In cases where all of the mutant hashes are equal to the original ones, the corresponding mutant is declared as equivalent. Similarly, to detect duplicate mutants, we compared the four hashes, but among the mutants. Note that in the case of duplicate mutants we report only the number of mutants that should be discarded.

3.5 Results

RQ$_2$: What is the rate of non-compilable (*e.g.,* those leading to failed compilations), trivial (*e.g.,* those leading to crashes on app launch), equivalent, and duplicate mutants produced by the studied tools when used with Android apps?

Fig. 3.3 depict the total number of mutants generated by each tool on each analyzed app, while Fig. 3.4 and Fig. 3.5 show the percentage of (a) Non-Compilable Mutants (NCM) and (b) Trivial Mutants (TM) respectively. As stated in Section 3.3, `MutAPK` was based on `MDroid+`, therefore, in the following comparisons we will show the results for both `MutAPK` and `MutAPK`-Shared[4] to study the benefits of applying the `MDroid+` operators at APK level.

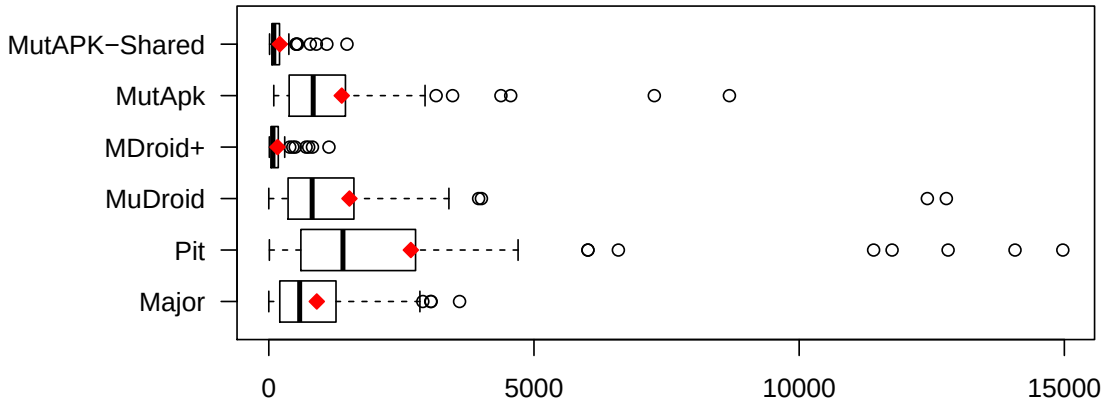


Figure 3.3. Distribution of number of mutants generated per app.

[4]MutAPK-Shared means `MutAPK` using only the operators shared with `MDroid+`

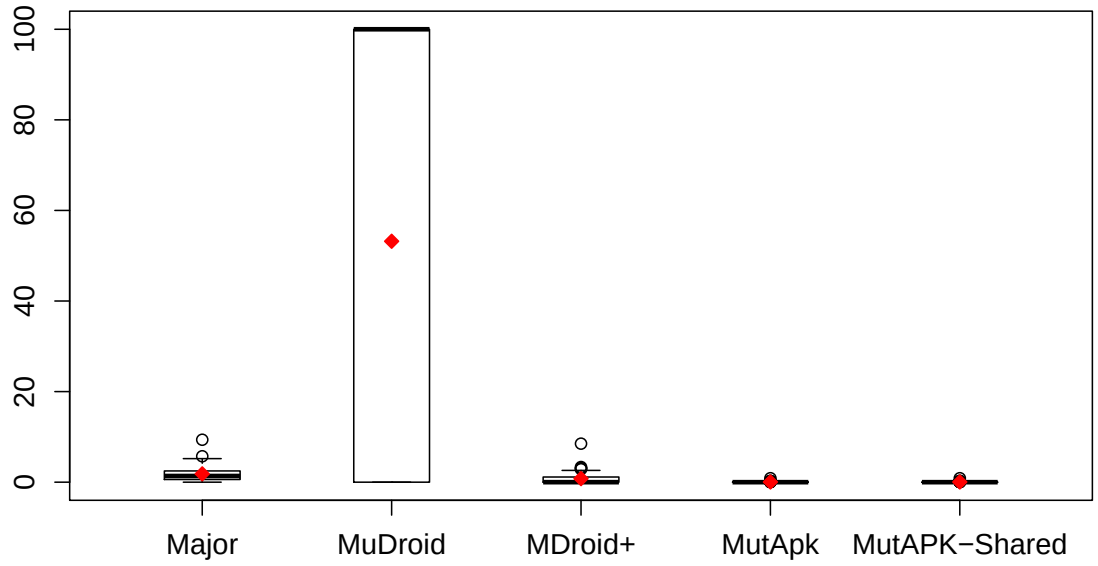


Figure 3.4. Distribution (%) of non-compilable mutants.

On average, 167, 207, 1.3k+, 904, 2.6k+, and 1.5k+ mutants were generated by `MDroid+`, MutAPK-Shared, `MutAPK`, Major, PIT, and muDroid, respectively for each app. The larger number of mutants generated by PIT is due, in part, to the larger number of mutation operators available for the tool; note that PIT uses object oriented-based mutators for Java source code. muDroid tends to generate a larger number of mutants due to its more generic mutation operators, meaning that there are more potential instances in the source code for mutants to be seeded.

`MutAPK` generates significantly more mutants than Major (Wilcoxon paired signed-rank test adjusted p-value< 0.001 with Holm's correction [125]) and significantly fewer mutants than PIT (Wilcoxon paired signed-rank test adjusted p-value < 0.001 with Holm's correction). However, `MDroid+` and `MutAPK`-Shared generate significantly fewer mutants than Major, muDroid, and PIT (Wilcoxon paired signed-rank test adjusted p-value < 0.001).

The average percentage of **Non-Compilable Mutants** (NCM) generated by `MutAPK`, MutAPK-shared, `MDroid+`, Major and muDroid over all the apps is 0.04%, 0.31%, 0.56%, 1.8%, and 53.9%, respectively, while no NCM were generated by PIT (Fig. 3.4). `MDroid+` produces significantly fewer NCM than Major (Wilcoxon paired signed rank test adjusted p-value < 0.001, and large Cliff's d=0.59) and than muDroid (adjusted p-value < 0.001, and medium Cliff's d=0.35). `MutAPK` and `MutAPK`-Shared produces significantly fewer NCM than Major and muDroid (Wilcoxon paired signed rank test adjusted p-value< 0.001)

These differences across the tools are mainly due to the compilation/assembly process they adopt during the mutation process. PIT works at Java bytecode level and thus can avoid the NCM problem, at the risk of creating a larger number of TM. However, PIT is the tool that required the highest effort to build a wrapper to make it compatible with Android apps. Major and `MDroid+` work at the source code level and compile the app in a "traditional"

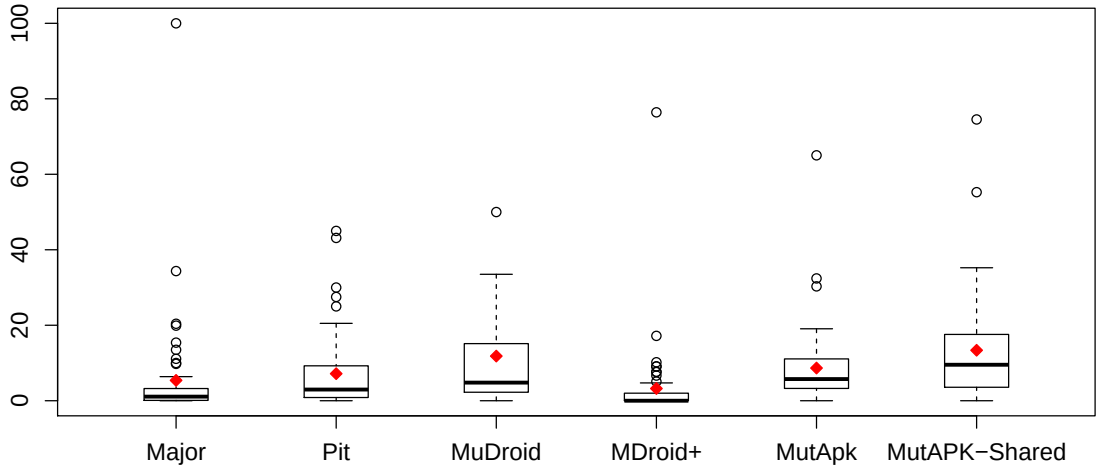


Figure 3.5. Distribution (%) of trivial mutants.

manner. Thus, it is more prone to NCM and requires an overhead in terms of memory and CPU resources needed for compiling/building the mutants. Finally, muDroid and `MutAPK` operate on `APKs` and `smali` code, reducing the computational cost of mutant generation, but significantly increasing the chances of NCM; muDroid is the top-one generator of NCM with an average of 53.9% of NCMs per app. However, `MutAPK` and `MutAPK`-Shared are the ones generating the least amount of NCM with averages of 0.04% and 0.31% respectively. This due to the process designed to create the mutation rules, as explained in Section 3.3.

All the analyzed tools generated **trivial mutants** (TM) (*i.e.,* mutants that crashed simply upon launching the app). These instances place an unnecessary burden on the developer, particularly in the context of mobile apps, as they must be discarded from the analysis. The average of the distribution of the percentage of TM over all apps for `MDroid+`, Major, PIT, `MutAPK`, muDroid and `MutAPK`-Shared is 2,42%, 5.4%, 7.2%, 9%, 11.8% and 13.62%, respectively (Fig. 3.5). `MDroid+` generates significantly less TM than muDroid (Wilcoxon paired signed rank test adjusted p-value=0.04, Cliff's d=0.61 - large) and than PIT (adjusted p-value=0.004, Cliff's d=0.49 - large), while there is no statistically significant difference with Major (adjusted p-value=0.11). `MutAPK` generates significantly more TM than Major (Wilcoxon paired signed rank test adjusted p-value < 0.001) and `MutAPK`-Shared generates significantly more TM than PIT (Wilcoxon paired signed rank test adjusted p-value < 0.001)

While these percentages may appear small, the raw values show that the TM can comprise a large set of instances for tools that can generate thousands of mutants per app. For example, for the Translate app, 518 out of the 1,877 mutants generated by PIT were TM. For the same app, muDroid creates 348 TM out of the 1,038 it generates. For the Blokish app, 340 out of the 3,479 GM by Major were TM. At the same time, for HNDroid app `MutAPK` generates 673 trivial mutants out of 1038 generated mutants but also for Anycut app generates only 2 TM out of 380 GM. Finally, `MDroid+` generates also for HNDroid 94 trivial mutants from 123 generated. Both `MutAPK` and `MDroid+` generate the smallest number of NCM with

55 and 37 respectively. However, `MDroid+` generates less TM, only 213 in total across apps due to being also the one generating less mutants, around 167 per app. At the same time, `MutAPK` belongs to the top generators of TM, with around 9% of TM per app.

Following the approach proposed by Papadakis *et al.* [114], we found that none of the tools generated equivalent mutants when comparing the hash values between the original APKs and the mutated APKs. As previously mentioned, the four hash values calculated for both original and mutated APK should be equal to identify a mutant as an equivalent mutant. Nevertheless, we used the same approach to find duplicate mutants by performing a pairwise comparison of all mutants. As a result, we found that `MutAPK`, MutAPK-Shared, PIT, and muDroid generate duplicate mutants. Specifically, 211, 43, 8, and 2,031 duplicate mutants were generated, respectively. Note that muDroid generates more duplicate mutants that other tools, with a percentage of 6.55% of the generated mutants; the second one is MutAPK-Shared with 0.38%; the third is `MutAPK` with 0.28%; and finally PIT with 0.006%.

Summary of RQ_2 findings: *As for the generation of mutants, all the analyzed tools (Major, Pit, muDroid, `MDroid+`, `MutAPK`) generated a relatively low rate of trivial mutants, with muDroid being the worst with a 11.8% average rate of trivial mutants. Additionally, no equivalent mutants were found for any tool, according to a hash-based comparison between the original APKs and the corresponding mutants. Nevertheless, 4 tools (`MutAPK`, MutAPK-Shared, PIT and muDroid) generated duplicate mutants, with muDroid being the worst with a 6.55% of total duplicate mutants.*

RQ_3: What are the major causes for non-compilable, trivial, equivalent, and duplicate mutants produced by the mutation testing tools when applied to Android apps?

We found that for Major, the Literal Value Replacement (LVR) operator had the highest number of TM, whereas the Relational Operator Replacement (ROR) had the highest number of NCM. It may seem surprising that ROR generated many NCM, however, we discovered that the reason was due to improper modifications of loop conditions. For instance, in the A2dp.Vol app, one mutant changed this loop: `for (int i = 0; i < cols; i++)` and replaced the condition "$i < cols$" with "`false`", causing the compiler to throw an unreachable code error. For PIT, the Member Variable Mutator (MVM) is the one causing most of the TM; for muDroid, the Unary Operator Insertion (UOI) operator has the highest number of NCM (although all the operators have relatively high failure rates), and the Relational Value Replacement (RVR) has the highest number of TM. For `MutAPK`, the FindViewByIdReturnsNull and NullValueIntentPutExtra operators had the highest number of NCM, while the NullMethodCallArgument operator generates the highest number of TM.

The details of the mutation operators being the source of duplicate mutants are depicted in Tables 3.2, 3.3, and 3.4. Table 3.2 presents the results for muDroid. As previously mentioned, muDroid generates the largest number of duplicate mutants. One example is *"Relational Operator Replacement"* mutant operator with 1135 duplicate mutants of 28,560 generated ones, which account for a total of 4% of duplicate mutants generated with this operator.

PIT's results are shown in Table 3.3; in this case, there are only 8 duplicate mutants where 6 of them belong to a relation between mutants having *"NegateConditional*" and "*Re-*

Table 3.2. Number of duplicate mutants created by muDroid grouped by operator.

Mutation Operators	**Amount**
Relational Operator Replacement	849
Inline Constant Replacement	228
Arithmetic Operator Replacement	486
Return Value Replacement	34
Logical Connector Replacement	6
Negative Operator Inversion	1
Inline Constant Replacement & Relational Operator Replacement	180
Inline Constant Replacement & Arithmetic Operator Replacement	77
Arithmetic Operator Replacement & Relational Operator Replacement	57
Return Value Replacement & Inline Constant Replacement	57
Return Value Replacement & Relational Operator Replacement	24
Return Value Replacement & Arithmetic Operator Replacement	14
Logical Connector Replacement & Inline Constant Replacement	4
Relational Operator Replacement & Logical Connector Replacement	3
Negative Operator Inversion & Inline Constant Replacement	3
Arithmetic Operator Replacement & Logical Connector Replacement	2
Negative Operator Inversion & Relational Operator Replacement	2
Negative Operator Inversion & Arithmetic Operator Replacement	2
Logical Connector Replacement & Return Value Replacement	2
Total (MuDroid)	2,031

moveConditional" operators. Finally, for `MutAPK` (Table 3.4) the *"NullMethodCallArgument*" is the operator generating more duplicate mutants. Additionally, it is worth noticing that there are 12 duplicate mutants in `MutAPK` and MutAPK-shared that are between different pairs of operators; this behavior is further analyzed later.

To qualitatively investigate the causes behind the crashes and duplicate generation, four authors manually analyzed a randomly selected sample of 15 crashed mutants and 10 duplicate mutants per tool. In this analysis, the authors relied on information about the mutation (*i.e.,* applied mutation operator and location), and the generated stack trace.

Major. The reasons behind the crashing mutants generated by Major mainly fall into two categories. First, mutants generated with the LVR operator that changes the value of a literal causing an app to crash. This was the case for the *wikipedia* app when changing the "1" in the invocation `setCacheMode(params.getString(1))` to "0". This passed a wrong asset URL to the method `setCacheMode`, thus crashing the app. Second, the Statement Deletion (STD) operator was responsible for app crashes especially when it deleted needed methods' invocations. A representative example is the deletion of invocations to methods of the superclass when overriding methods, *e.g.,* when removing the `super.onDestroy()` invocation from the `onDestroy()` method of an `Activity`. This results in throwing of an `android.util.-`

Table 3.3. Number of duplicate mutants created by PIT grouped by operator.

Mutation Operators	**Amount**
NegateConditional	1
RemoveSwitch	1
NegateConditional & RemoveConditional_ORDER_ELSE	6
Total (PIT)	8

Table 3.4. Number of duplicate mutants created by `MutAPK` and `MutAPK`-Shared grouped by operator.

Mutation Operators	**Amount**
DifferentActivityIntentDefinition	10
NullValueIntentPutExtra	6
NullIntent	6
WrongStringResource	3
InvalidIDFindView	3
LengthyGUICreation	2
InvalidActivityPATH	1
ActivityNotDefined	1
InvalidFilePath	1
NullValueIntentPutExtra & NullBackEndServiceReturn	3
NullValueIntentPutExtra & MissingPermissionManifest	2
WrongStringResource & NullIntent	2
MissingPermissionManifest & ViewComponentNotVisible	2
LengthyGUICreation & LengthyGUIListener	1
Subtotal (`MutAPK`-Shared - Common operators)	43
NullMethodCallArgument	165
InvalidViewFocus	1
InvalidActivityPATH & InvalidViewFocus	2
Total (`MutAPK`)	211

`SuperNotCalledException`. Other STD mutations causing crashes involved the deletion of a statement initializing the main `Activity` leading to a `NullPointerException`. No duplicate mutants were identified among the mutants generated by Major.

muDroid. This tool is the one exhibiting the highest percentage of NCM and TM. The most interesting finding of our qualitative analysis is that 75% of the crashing mutants lead to the throwing of a `java.lang.VerifyError`. A `VerifyError` occurs when Android tries to load a class that, while being syntactically correct, refers to resources that are not available (*e.g.,* wrong classpaths). In the remaining 25% of the cases, several of the crashes were due to the Inline Constant Replacement (ICR) operator. An example is the crash observed in the `photostream` app where the "100" value has been replaced with "101" in `bitmap.compress(Bitmap.CompressFormat.PNG, 100, out)`. Since "100" represents the quality of the compression, its value must be bounded between 0 and 100.

In terms of duplicate mutants, muDroid[5] is also the tool generating the highest amount of DM. As listed in Table 3.2, the 6 mutants operators generate duplicate mutants, and there are 13 combinations of operators that also generate duplicate mutants. The mutation operator that generates more duplicate mutants is *Relational Operator Replacement (ROR)*. This operator generates around 850 mutants that are duplicate with other ROR mutants. Additionally, there are around 266 duplicate mutant pairs with one of the mutants being a result of ROR operator being applied. After manually analyzing the duplicate mutants, we found that there are implementation errors in muDroid, since several mutants generated with the ROR, ICR and AOR operators have identical mutations in the same app, despite being reported in the log files as different mutants.

PIT. In this tool, several of the manually analyzed crashes were due to (i) the RVR operator changing the return value of a method to null, causing a `NullPointerException`, and (ii) removed method invocations causing issues similar to the ones described for Major. In terms of the duplicate mutants, PIT generated the lowest rate (8 out of 103k mutants). The most common duplicate mutant case is between *NegateConditional* and *RemoveConditional_ORDER_ELSE*. From its definition, the *RemoveConditional_ORDER_ELSE* operator is a specialization of the base mutant operator *RemoveConditional*, whose objective is to change "a==b" to "true". The specialized operator negates the condition to ensure the ELSE block is executed; however, in some cases, the effect of both operators is the same. Consider, for example, the following source code snippet:

```
1  if(a < b){
2      // Do something
3  } else {
4      // Do something else
5  }
```

If we apply *NegateConditional* operator, PIT will look for the conditional operator used in the if statement (*i.e.,* $<$) and it would negate it, replacing it with a $>=$. However, if we apply *RemoveConditional_ORDER_ELSE*, PIT would make the necessary change in the if condition so the else statement is executed. This is represented in replacing the $<$ condition, with a $>=$ condition. Therefore, both operators will give a final result where the $<$ condition was

[5]For the time of the writing of this document last update of MuDroid's source code was done in May 3, 2016

replaced with a >= condition.

MDroid+. Table 3.5 lists the mutants generated by MDroid+ across all the systems (information for the other tools is provided with our replication package). In MDroid+, the overall rate of NCM was quite low with the ClosingNullCursor operator having the highest total number of NCM (across all the apps) with 13. These instances stem from edge case that trigger compilation errors involving cursors that have been declared Final, thus causing the reassignment to trigger the compilation error. The small number of other NCMs are generally other edge cases, and current limitations of MDroid+ can be found in our replication package with detailed documentation. No duplicate mutants were identified among the mutants generated by MDroid+.

MutAPK. Table 3.5 lists the mutants generated by MutAPK across all the systems (information for the other tools is provided with our replication package). The overall rate of NCM is very low in MutAPK, and most failed compilations pertain to specialized cases that would require a more robust static analysis approach to inject the mutations. However, it is worth noting that MutAPK works at APK level, and the mutation rules require more modifications (when compared to source code level mutations) to generate a valid mutation. For example, the *FindViewByIdReturnsNull* mutation rule consists of replacing the statements used to find a specific component with a null value assignment to the register where the corresponding result is stored. This storage process could store the value in a register higher than 16. This value is the maximum register index normally accepted by DALVIK instructions. However MutAPK uses the *const/4 v#, 0x0* instruction to set the null value. Therefore, if the register used for the assignment is above 16 the instruction will not compile. To correctly mutate the app, an extensive search for an empty register must be done to store the null value. However, if there is no empty register, MutAPK would need to save the value from one of the below-16-registers into a temporal above-16-register, use the first selected register to storage the null value while the process uses it and then reassign its original value. This problem also applies to *NullValueIntentPutExtra* mutation operator.

The operator generating the highest number of TM is *NullMethodCallArgument* (84.43%, *i.e.,* 4,264 out of 5,050). The main reason for this behavior is due to the nature of the mutation rule; the *NullMethodCallArgument* operator replaces one parameter value in a method call with a null value. Therefore, all method invocations with the null value as an argument will throw an exception when the method does not handle the null value. It is worth noting that MutAPK generates 63,441 mutants using this operator, therefore only 6.72% of generated mutants are trivial under our definition. Future work must be focused on avoiding mutations of this type in the main activities to avoid the TM case. There are also 3 other operators that increase the amount of TM generated by MutAPK. First, *FindViewByIDReturnsNull*, modifies a *findViewByID* call to return null. Second, InvalidIDFindView replaces the parameter that represents the view Id required with a generated randomly, therefore, the result of the *findViewByID* call will be a null value. Third, the *NullValueIntentPutExtra* operator replaces the value sent as extra in an intent with a null value. Therefore, just like what happens with *NullMethodCallArgument*, all the method invocations and statements that do not correctly handle the null values will generate an exception breaking the app.

In terms of duplicate mutants, MutAPK generates the largest amount with the *NullMethod-*

CallArgument operator. By definition, this operator changes the value of a parameter in a method call to null. In source code, it is possible to find method calls that use the same value more than once in a call. For example, in the a2dp.Vol app, the method *deleteAll* calls the method *delete* by providing twice a null value as parameter: `this.db.delete(TABLE\_NAME, null, null)`. This instruction at APK level also makes the call using two times the same parameter:

```
1 invoke-virtual {v0, v1, v2, v2}, Landroid/database/sqlite/SQLiteDatabase;->delete(Ljava/lang/String;
      Ljava/lang/String;[Ljava/lang/String;)I
```

However, MutAPK does not validate the current value of the parameter before changing it to null; therefore, the value for *v2* was already null before MutAPK injected the null assignment. This is also an example of an equivalent mutant that cannot be found by following the TCE approach, since the SMALI bytecode was modified and the hash values were not affected by the change.

It is also important to see that there are some mutant operators shared with MDroid+ that are generating duplicate mutants only at APK level, such as *DifferentActivityIntentDefinition* that has different implementations in both tools. This is a good example since this operator requires finding a new Activity name for making the change. In the MutAPK case, the operator is not validating that the activity should not be replaced with the same name (picked randomly from the list of activities in the APK).

Summary of RQ_3 Findings: *The performed analysis indicate that the PIT tool outperforms others in terms of ratio between non-compilable and generated mutants, because it does not generate any non-compilable mutant. However, MDroid+ and MutAPK provide Android-specific mutations, which make the tools (*i.e., *Pit, MDroid+, MutAPK) complementary for mutation testing of Android apps. MDroid+ and MutAPK generated the lowest rate of both non-compilable and trivial mutants (when compared to Major and muDroid), illustrating its immediate applicability to Android apps. Major and muDroid generate non-compilable mutants, with the latter having a critical average rate of 58.7% non-compilable mutants per app. Also, even when PIT generates duplicate mutants, the number is insignificant when compared to the number of mutants generated; at the same time MutAPK and MutAPK-Shared also generate a low number of duplicate mutants; some of them can be fixed by improving the current implementation.*

RQ_4: What are the benefits and trade-offs of performing mutation testing at APK level vs. source code level?

Table 3.5 presents the results from both MutAPK and MDroid+ in terms of Generated Mutants (**GM**), Non-Compilable Mutants (**NCM**) and Trivial Mutants (**TM**) per mutation operator defined in this study. Note that each tool has implemented some operators that the other does not. Therefore, we first study the times required by both tools to (i) generate a mutated copy of the app and to (ii) compile/assemble a given mutant into an APK. Then, we analyze the mutation results taking into account only the operators that are common in both tools, and finally, we study the impact of tool-specific operators in the results.

As it was mentioned previously, we ran MutAPK and MDroid+ over 55 apps. MDroid+'s default behavior generates mutated copies of the original source code; therefore, we implemented a wrapper that based on the MDroid+ results builds the corresponding APKs. MutAPK

Table 3.5. Number of Generated (GM), Non-Compilable (NCM), and Trivial Mutants (TM) created by MDroid+ and MutAPK

	MDroid+			MutAPK		
Mutation Operators	**GM**	**NCM**	**TM**	**GM**	**NCM**	**TM**
WrongStringResource	3,394	0	14	3,432	0	10
NullIntent	559	3	41	482	0	37
InvalidKeyIntentPutExtra	459	3	11	477	0	9
NullValueIntentPutExtra	459	0	14	477	22	103
InvalidIDFindView	456	4	30	1,313	0	193
FindViewByIdReturnsNull	413	0	40	1,313	28	190
ActivityNotDefined	384	1	8	385	0	11
InvalidActivityName	382	0	10	383	0	50
DifferentActivityIntentDefinition	358	2	8	482	0	7
ViewComponentNotVisible	347	5	7	398	0	58
MissingPermissionManifest	229	0	8	227	0	7
InvalidFilePath	220	0	1	228	0	36
InvalidLabel	214	0	3	214	0	5
ClosingNullCursor	179	13	5	222	0	14
LengthyGUICreation	129	0	1	336	0	15
LengthyGUIListener	122	0	0	339	0	5
NullInputStream	61	0	4	90	0	4
WrongMainActivity	56	0	0	56	0	8
InvalidColor	52	0	0	47	0	0
NullOuptutStream	45	0	2	59	0	2
InvalidDate	40	0	0	20	0	0
InvalidSQLQuery	33	0	2	82	0	7
NullBluetoothAdapter	9	0	0	9	0	0
LengthyBackEndService	8	0	0	15	15	0
NullBackEndServiceReturn	8	1	0	34	5	2
InvalidIndexQueryParameter	7	1	0	82	0	2
OOMLargeImage	7	4	0	7	0	4
BluetoothAdapterAlwaysEnabled	4	0	0	1	0	0
InvalidURI	2	0	0	2	0	0
NullGPSLocation	1	0	0	2	0	0
LongConnectionTimeOut	0	0	0	0	0	0
SDKVersion	66	0	2	0	0	0
Subtotal (Common operators)	8,703	37	211	11,214	55	779
NotParcelable	7	6	0	-	-	-
NotSerializable	15	7	0	-	-	-
BuggyGUIListener	122	0	2	-	-	-
NullMethodCallArgument	-	-	-	63,441	0	4,264
InvalidViewFocus	-	-	-	398	0	7
Total	8,847	50	213	75,053	55	5,050

works directly on APKs; therefore, the APK generation is embedded in the mutation process.

As it can be seen in Table 3.6, MutAPK takes 6.17% of the time required by MDroid+ to mutate a copy of the app and 12.95% of the time required to compile/assemble the mutant into an APK. Therefore, MutAPK executes the complete mutation process (*i.e.,* mutation of app copy plus compilation/assembling) 87.2% faster than MDroid+.

Table 3.6. Summary of time results: MutAPK vs. MDroid+.

Metric Name	MutAPK	MDroid+
Avg. Mutation Time (secs.)	284.67×10^{-3}	4.61
Avg. Compilation Time (secs.)	25.265	195
Avg. Full Mutant Creation Time (secs.)	25.549	199.61

Common operators. Concerning the mutation type metrics (Table 3.5), when considering only the common operators, MDroid+ and MutAPK generated 8,703 and 11,214 mutants respectively. This shows that at the APK level, the *PFP* (Section 3.3) detects more locations for implementing mutations. MutAPK generates on average 78 more mutants per operator with a median of 7.5 more than MDroid+. However, the *SDKVersion* operator does not find instances in any app due to latest modifications of the Android building process, where all the details for SDK Version must be defined in the *build.gradle* file instead of the Manifest file.

The number of NCM is very similar in both cases. However, the percentage of TMs is larger with MutAPK (6.94%) than with MDroid+ (2.24%), and this happens because of the *FindViewByIdReturnsNull*, *InvalidIDFindView*, and *NullValueIntentPutExtra* operators that account for 62,39% of the trivial mutants in MutAPK. Note that MutAPK generates more mutants than MDroid+ for those operators, because SMALI representation of code statements must express each instruction in a line. Therefore, as it can be seen in Listing 3.6 a Java statement can contain several instructions that are solved from inside to outside. However, a given Java statement in SMALI uses a line for each instruction as it can be seen in Listing 3.7. Knowing that, MDroid+'s search power is reduced by the Java capability of chaining instructions. For example, for *FindViewByIDReturnsNull* operator MDroid+ search for statements where the view is stored in a variable (see *Listing 3.8*). However, if the result of *findViewById* method is used directly as parameter (see *Listing 3.6*), MDroid+ does not recognize that statement as part of the PFP.

Listing 3.6. Java chained instructions

```
1 highlight(findViewById(R.id.load_data_button));
```

Listing 3.7. SMALI representation of JAVA chained instructions

```
1 invoke-virtual {p0, p1}, Lio/github/hidroh/materialistic/AboutActivity;->findViewById(I)Landroid/view/
      View;
2 move-result-object v0
3 check-cast v0, Landroid/widget/TextView;
4 invoke-virtual {v5, v0}, Lio/github/hidroh/materialistic/AboutActivity;->highlight(I)Ljava.awt.String;
```

Listing 3.8. MDroid+ mutation rule

```
1 ImageButton loadButton = (ImageButton) findViewById(R.id.load_data_button);
```

If NCM, TM and DM are removed, we can see that MDroid+ generated 8,455 mutants versus 10,337 mutants generated by MutAPK. Therefore, the results suggest that MDroid+ takes 9.79 more hours to generate 21.9% less functional mutants (*i.e.,* mutants that compile and are not trivial nor equivalent nor duplicate) than MutAPK.

Whole set of operators. Considering all the operators available with each tool, MutAPK generates per operator on average 63% more mutants than MDroid+ with a median of 3.9% and a mode of 0%. We found that MutAPK generates a significant amount of extra mutants because of the *NullMethodCallArgument* operator. This operator is capable of generating 63,441 additional mutants for the 55 apps, which is around 6 times the amount of mutants generated by the rest of the operators. However, 6.72% of those are trivial and 0.26% are duplicate. Additionally, on the one hand MutAPK also implements *InvalidViewFocus*, that generated 398 mutants (only 7 were trivial and 1 duplicate). On the other hand, MDroid+ has 3 additional operators that add 144 mutants to the list; 13 of them are non-compilable and only 2 are trivial. Even in this case, MutAPK is able to generate more functional mutants: 69,742 vs 8,579 generated by MDroid+.

Summary of RQ_4 Findings: *The clear benefit of performing APK-level mutation analysis (MutAPK) as opposed to source code-level mutation analysis (MDroid+) relates primarily to the ease of use of the system, as it only requires a single file (*i.e., *an APK file instead of several source files), and generates mutants with higher compilability ratio in less time. Moreover, this makes the mutation tool applicable to apps written with various languages,* i.e., *Java, Kotlin, and Dart. However, this ease of use comes with a slight trade-off in terms of generating a higher number of mutants (which leads to an extensive execution effort from developers), and a higher number of trivial and duplicate mutants for certain operators that are likely to be discarded during mutant analysis.*

3.6 Threats to validity

This section discusses the threats to validity of the work related to devising the fault taxonomy, and carrying out the study reported in Section 3.4.

Threats to *construct validity* concern the relationship between theory and observation. The main threat is related to how we assess and compare the performance of mutation tools, *i.e.,* by covering the types, and by their capability to limit non-compilable, trivial, equivalent and duplicate mutants. In particular, for detecting equivalent and duplicate, we relied on the TCE approach [114], which was able to detect (on average) 30% of equivalent mutants on a benchmark of 18 small/medium C/C+ programs [124]. Our choice for TCE is justified by the fact that the analyzed apps do not have test suites that can be used for mutation analysis. However, we can not claim that we are not detecting all the equivalent and duplicate mutants generated by the analyzed tools.

Threats to *internal validity* concern factors internal to our settings that could have influenced our results. This is, in particular, related to the possible subjectiveness of mistakes in

the tagging and for **RQ**$_1$. As mentioned by the authors[93], they employed multiple taggers to mitigate such a threat

Threats to *external validity* concern the generalization of our findings. To maximize the generalizability of the fault taxonomy, we have considered six different data sources. However, it is still possible that we could have missed some fault types available in sources we did not consider, or due to our sampling methodology. Also, we are aware that in our study results of **RQ**$_1$ are based on the new sample of data sources, and results of **RQ**$_2$**, RQ**$_3$**, and RQ**$_4$**,** on the set of 68 apps considered [119]. Also, although we compared the proposed tools with several state-of-the-art mutation tools (six tools in **RQ**$_1$ and three tools in **RQ**$_2$ **and RQ**$_3$), our results may not generalize to tools not included in the study.

4

Enabling mutant selection for open-and closed-source android apps

Mutation testing is a time consuming process because large sets of fault-injected-versions of an original app are generated and executed with the purpose of evaluating the quality of a given test suite. In the case of Android apps, recent studies even suggest that mutant generation and mutation testing effort could be greater when the mutants are generated at the APK level. To reduce that effort, useless (*e.g.,* equivalent) mutants should be avoided and mutant selection techniques could be used to reduce the set of mutants used with mutation testing. However, despite the existence of mutation testing tools, none of those tools provides features for removing useless mutants and sampling mutant sets. In this chapter, we present `MutAPK 2.0`, an improved version of our open source mutant generation tool (`MutAPK`) for Android apps at APK level. To the best of our knowledge, `MutAPK 2.0` is the first tool that enables the removal of dead-code mutants, provides a set of mutant selection strategies, and removes automatically equivalent and duplicate mutants.

Structure of the Chapter

- Section 4.1 provides motivation for this chapter.
- Section 4.2 presents the architecture of the two tools used for the study
- Section 4.3 discusses our results and findings.

Supplementary Material

All the data used in this chapter as well as our tool are publicly available. More specifically,we provide the following items:

Online Appendix [REF] The online appendix includes the following material:

- **Source Code.** `MutAPK 2.0` source code is available as a GitHub repository.
- **Installation.** Installation instructions and prerequisites are provided.
- **Usage.** Usage and parametrization instructions for mutant generation and selection
- **List of Applications.** List of Android applications assessed using `MutAPK 2.0`

Publications and Contributions

MutAPK: Source-codeless mutant generation for android apps.[13] **Escobar-Velásquez, C.**, Osorio-Riaño, M., and Linares-Vásquez, M. In *Proceeding of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019*.

`MutAPK 2.0` → Chapter 4
We present `MutAPK 2.0`, an improved version of our open source mutant generation tool (`MutAPK`) for Android apps at APK level. To the best of our knowledge, `MutAPK 2.0` is the first tool that enables the removal of dead-code mutants, provides a set of mutant selection strategies, and removes automatically equivalent and duplicate mutants.

4.1 Introduction

Mutation testing is a cumbersome and time consuming technique aimed at assessing test suites quality [126]. Therefore, a plethora of previous works have investigated different ways to reduce the effort required by mutation testing [127]. In addition, a significant number of approaches have been proposed to support mutant generation and mutation testing for different languages and types of apps [95, 96, 97, 128]. The case of Android apps is not the exception; developers and researchers can use a diverse set of tools for mutation testing of Android apps [93, 106, 107, 108, 109, 118], and there are also specific challenges that impact the cost of mutation testing. For instance, enabling mutation at APK level [12] increases signifcantly the amount of Android-specific mutants that can be generated, when compared to mutation at source-code level.

In general, removing useless mutants (*i.e.,* equivalent, duplicate, and dead-code mutants) is a widely used technique for reducing testing efforts. The latter type, dead-code mutants[1], are less common because compiler optimizations take care of removing dead-code when building executable files/packages. However, this is not the case for Android native apps written in Java and Kotlin, because dead code is not removed (by default) when building APK files. In addition, mutant selection techniques [129] are a desirable feature during mutation testing to reduce the amount of analyzed mutants while preserving the quality of the mutant population.

Unfortunately, despite the availability of a diverse set of mutation tools for both source-code and bytecode levels, none of the existing tools provide developers with capabilities for removing dead-code, equivalent and duplicate mutants. Moreover, practitioners and researchers lack publicly available tools for mutant selection and sampling. Therefore, in this paper we present `MutAPK 2.0`, an improved version of our original mutant generation tool (`MutAPK`) for android apps at APK level. `MutAPK 2.0` enhances mutant generation process by removing dead-code, equivalent and duplicated mutants. Additionally, it provides users with three mutant selection techniques based on random and representative subset selection.

4.2 MutAPK 2.0 Tool

In this section, we describe the main features implemented in `MutAPK 2.0` to support (i) mutant selection, (ii) identification and removal of duplicated and equivalent mutants, and (iii) dead code analysis. Fig. 4.1 presents the workflow `MutAPK 2.0` follows to generate and reduce APK mutants for Android apps. Note that our first release of `MutAPK` [12] focused on mutants generation but without having mechanisms for reducing the generated mutants. In this paper we describe the components added and modified on top of `MutAPK` (*i.e.,* sections framed in red in Fig. 4.1). In order to understand the original `MutAPK` architecture we refer the interested reader to our preliminary publication [12] and online appendix [130]. As the reader can note, we have improved three stages of the mutant generation process: (i) prior to the Potential Faults Profile (PFP) definition, (ii) after the PFP definition and before mutant generation, and (iii) during mutant building process.

[1]By dead-code mutants we mean mutants that are generated by applying mutation operators on dead-code.

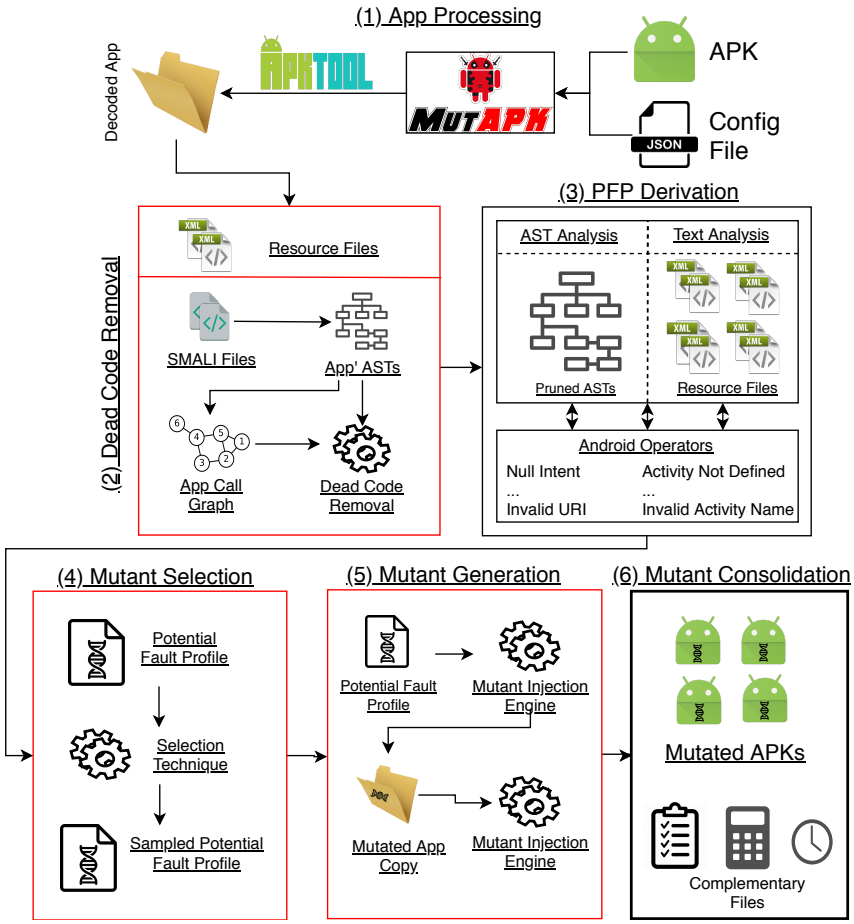


Figure 4.1. MutAPK 2.0 architecture and workflow

4.2.1 Avoiding Mutants from Dead Code

The Android compilers for Java and Kotlin languages do not remove dead code automatically, which means that by default, APKs of native apps include dead code. Thus, `MutAPK` allows users to reduce the mutants set by not injecting mutation operations on dead code. In order to identify the dead code within the app' SMALI representation[32], our tool builds the call graph of an app under analysis and identifies the methods that are not called by others.

Before explaining the dead code detection details, we provide some context about how a method call is represented in SMALI. As it can be seen in Fig. 4.2, a method call has 4 sections that are relevant for our purpose: (i) ***unitName***, (*i.e.,* the class a method belongs to), (ii) ***methodName***, (iii) ***parameters***, represented as a concatenation of the types of the method arguments, and (iv) ***returnType***. Having this in mind we define the `methodId` of a method as the concatenation of ***methodName***, ***parameters*** and ***returnType***; and the `methodCompoundId` as the concatenation of ***unitName*** and ***methodId***. This decision is based on the fact that it might exist more than one method with the same name but that differs by the parameters count and types.

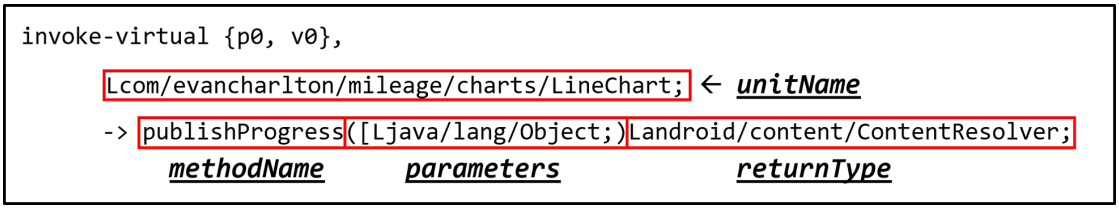


Figure 4.2. SMALI representation of method call

Concerning the dead code detection, `MutAPK` splits the AST computation and PFP derivation [93] into two steps. First, it creates a dictionary that maps the ***unitName*** of a SMALI file to an object containing the AST and the qualified path. Second, using the previously generated ASTs, `MutAPK` visits all the methods within the different ASTs to identify the method calls.

Afterwards, `MutAPK` extracts the *Call Graph* of the app using a two levels HashMap, being the second level a mapping from the `methodId` to a *CallGraphNode (cGN)* and the first level a mapping from the ***unitName*** to the aforementioned second level HashMap of methods. Now, regarding the content of a *CallGraph node*, we store the `methodId`, the ***unitName***, the AST node that represents the method, and two sets containing (i) the *cGNs* of the methods that call the current method (*callers*), and the *cGNs* of the methods that are called by the current method (*callees*). As the reader might know, there are some methods that are called through the code that do not belong to the app's developed code (*i.e.,* libraries' methods), thus, we excluded API calls from the call graph creation[2].

Once `MutAPK` has built the call graph, the methods that do not have a caller are removed from the original AST, therefore, during the mutant generation stage, no mutation is applied on those methods. Note that there are some callback and lifecycle methods in the Android programming model that are automatically invoked by the framework, so, those methods usually do not have callers in the call graph; therefore, we have manually created a *greenList* of those methods that should not be removed from the AST [3]. Finally, when all ASTs were pruned conserving the methods in the *greenList*, the PFP derivation and mutant generations process are executed.

4.2.2 Mutant Selection Techniques

After the PFP derivation is completed, a mapping between the mutation operators and the locations within the code identified as mutable is obtained. Using this map as input, along with the user selected technique, `MutAPK` generates a subset of the PFP to be used during the mutant generation process. If no selection technique is defined by the user, then, the whole set of mutants is generated. We describe each selection technique as follows.

[2]A file containing a list of ***methodCompoundId*** for API calls is generated at the end of mutant generation process in case a user wants to analyze them.

[3]The list of methods can be found in our online appendix[130]

Random selection (randSel).

In this case, a user must provide a desired amount of mutants to be selected. First, `MutAPK` checks that the user's required amount of mutants is less than the available PFP locations. Second, in order to ensure all mutant operators that have PFP locations, are being represented in the subset, `MutAPK` uses a *round robin* exploration technique to go through all possible key values in the PFP map. Once the algorithm has retrieved the list of locations associated to a mutant operator, it randomly selects one element, adds it to the final set of PFP locations and remove it from the mutation operator's list. If there is at least one location per identified mutation operator, `MutAPK` creates a list with all the available locations within the PFP and randomly selects the remaining amount of mutants.

Representative Subset.

The second available technique is based on the selection of an amount of mutants that fits the size of a representative sample of the PFP locations set. In order to use this technique, a user is required to provide three values: (i) ***confidenceLevel (c)***, (ii) ***marginError (e)***, and (iii) ***selectionScope***. `MutAPK` uses the equation shown in Fig. 4.3 to calculate the size of the subset. Until this point there are two values that are not being provided by the user, the *z-score* and the base' population size. Nevertheless, *z-score* is associated to the confidence level and the size of the population size is calculated based on the value of the latter parameter.

$$sampleSize(N, e, z) = \frac{\frac{{z_c}^2 p(1-p)}{e^2}}{1 + \left(\frac{{z_c}^2 p(1-p)}{e^2 N}\right)}$$

Figure 4.3. Sample Size equation. N = population Size, e = margin of error, z_c = z-score, and p = expected proportion

The population size (N) is computed based on the value of the selectionScope argument provided by the user:

Per Mutation operator (rSPerOperator). Represented as a *positive* boolean value in the config file, this option generates the resulting subset as a concatenation of the subsets of the locations belonging to each mutant operator. Which means, if a user selects this option, `MutAPK` would go through all mutant operators in the PFP, calculating and randomly selecting a sample of the locations associated to the mutant operator. Therefore, the resulting set would be a concatenation of the subsets for each mutant operator. For example, let us say we have a PFP with 1000 locations that are distributed between two mutant operators: *InvalidURI (N=600)* and *NullInputStream (N=400)*, and let us use a ***confidenceLevel*** of 95% and ***marginError*** of 5%. Once `MutAPK` is executed it would calculate the sample size for *InvalidURI* using the previous equation and will randomly select 235 mutants from the 600 mutants. Afterwards, it will continue by calculating the amount for *NullInputStream* resulting in a set of size 197. Finally, it would return a set of 432 *(235+197)* mutants.

Whole PFP set (rSWholePFPSet). On the other hand, if a user provides a *negative* boolean value, MutAPK uses the whole PFP set size as the N parameter of the equation. Nevertheless, it would first select at least one mutant per mutation operator to ensure the representativeness of the sample. Using the same example previously defined, MutAPK would use 1000 as N, however, it would first randomly select a mutant from each mutant operator.

4.2.3 Duplicate and Equivalent Mutants

To enhance the quality of the generated mutants set, we have included a step for removing duplicate and equivalent mutants. This process relies on the *Trivial Compiler Equivalence (TCE)* proposed by Papadakis *et al.* [114]. Given two apps, TCE calculates their similarity via the hashcode of its files. Specifically, in order to identify equivalent mutants (*i.e.,* mutants that are syntactically equivalent to the original app) we compare each mutant to the base app, and for duplicate mutants (*i.e.,* mutants that are syntactically equivalent to other mutants) we compare each pair of mutants by using TCE.

Since one of the most time consuming processes in the generation of mutants is the compilation/building of the mutants, we decided to compute TCE using three hashcodes for each mutant [11]: (i) *smali folder hash*, containing all the source code representations, (ii) *res folder hash*, containing all the resources of the app, and (iii) *AndroidManifest.xml hash*. By relying on the three hashes, we compute a ***compoundHashcode*** for each mutant, with the procedure depicted in Listing 4.1. This approach allows us to generate one Hashcode that represents the mutant and to use a hashmap structure to map ***compoundHashcode*** values to mutants . Afterwards, if there is no difference between the ***compoundHashcode*** of original app and a mutant or between two mutants, then we mark the mutants as equivalent or duplicate (respectively).

Listing 4.1. Compound Hash code computation algorithm

```
int hash = 7;
hash = 31 * hash + hashManifest.hashCode();
hash = 31 * hash + hashSmali.hashCode();
hash = 31 * hash + hashResource.hashCode();
return hash;
```

The preference for a hashmap to map ***compoundHashcodes*** and mutants, is based on the fact that its search complexity is $O(1)$. Therefore, it allows MutAPK to easily identify if a mutant ***compoundHashcodes*** already exists. MutAPK computes the ***compoundHashcode*** of the base application and stores it in the hashmap. By this, MutAPK is able to identify, in case a hashcode is duplicated, which mutant is colliding to. MutAPK also computes the original app's ***compoundHashcode*** and adds it to the hashmap. Therefore, if there is a collision with the original app, then the mutant is identified as equivalent; if there is a collision but between mutants, the mutant is then tagged as a duplicate.

4.3 Evaluation

In order to evaluate the `MutAPK 2.0` features, we used our tool with 10 open source android apps belonging to the androtest [131] dataset. In particular, we aimed at measuring the reduction of the dataset when removing dead-code mutants, as well as equivalent and duplicate ones. In addition, we evaluated whether the implementation of the selection techniques produced balanced sets (in terms of the mutation operators). Since our approach analyzes dead code within the SMALI representation, we selected the top 10 apps in terms of the smali folder size of their corresponding APK files. Note that (i) the original version of our tool [12] does not detect dead code before applying the mutation operators, and (ii) both (1.0 and 2.0) versions implement the same list of mutation operators. The list of analyzed apps is with our online appendix [130].

4.3.1 Dead-code Mutants

To measure the number of dead-code mutants we executed both versions of our tool (**`MutAPK`** [12] and `MutAPK 2.0`) on the ten selected apps and without using the equivalent/duplicate removal feature from `MutAPK 2.0`. By computing the difference in the number of mutants generated by each tool, we were able to identify the number of mutants created by mutations on dead code.

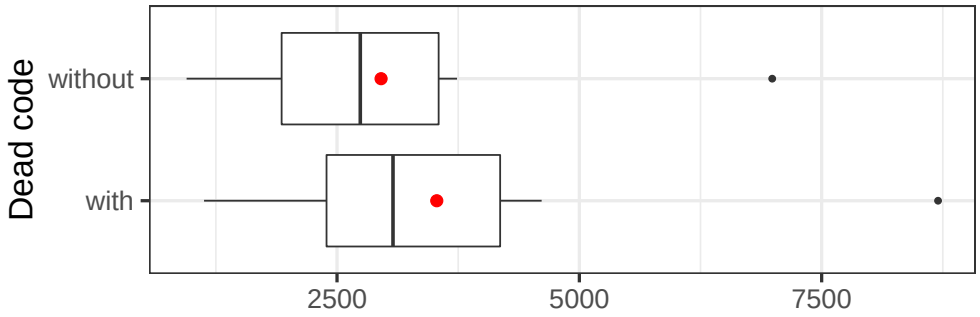


Figure 4.4. Amount of mutants generated with and without dead code on the 10 analyzed apps.

Fig. 4.4 presents the amount of mutants generated on the 10 apps. As expected, the set of mutants is larger when dead code is not removed. Specifically, there is a difference (on average) of 15.5% when removing dead-code mutants. Since mutation testing requires to execute a test suite on the generated mutants, by avoiding mutant generation over dead code, a user might reduce its execution time by 15%. In our dataset, *com.eleybourn.bookcatalogue* is the app with the largest number of dead-code mutants, 1711 cases (19.9%). Nevertheless, the *com.bwx.bequick* app is the one having the largest percentage of dead code mutants: 20.9% (*i.e.,* 462 out of 2210).

4.3.2 Mutant Selection Techniques

In order to evaluate the behavior of the selection techniques, we calculated the difference between the estimated amount of mutants[4] and the number of selected mutants by operator and app. Therefore, we extracted from the `MutAPK 2.0` logs the number of mutants generated for each mutant operator per app after applying a given selection strategy. Additionally, we estimated the expected amount of mutants that should be generated per mutant operator in order to preserve the distribution of the original mutant dataset (*i.e.,* without applying any selection technique). Once we had those values, we computed the average difference for the three selection techniques available in `MutAPK 2.0`.

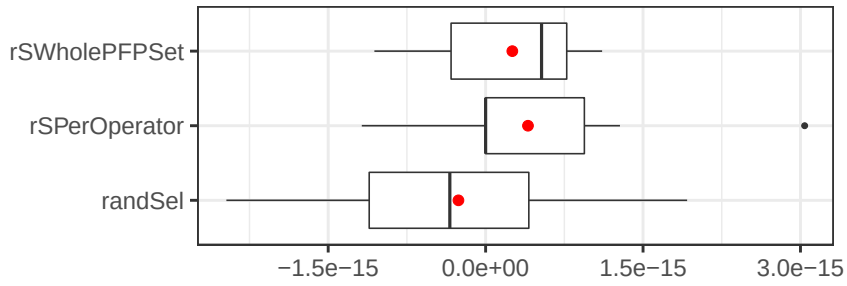


Figure 4.5. Average difference between expected amount of mutants and amount of generated mutants per mutant operator for each selection technique. randSel = Random Selection, rSPerOperator = representative subset per operator, rSWholePFPSet = representative subset whole PFP set

Fig. 4.5 depicts the average differences computed previously. The obtained values show that the three selection techniques generated a small difference when compared to the expected amount. Nevertheless, we could notice, by analyzing the standard deviation, that even when the average difference is close to 0, the distance to the mean per mutant operator is higher when using a fully random technique (*i.e., amountMutants*) than when using a technique based on representative subset.

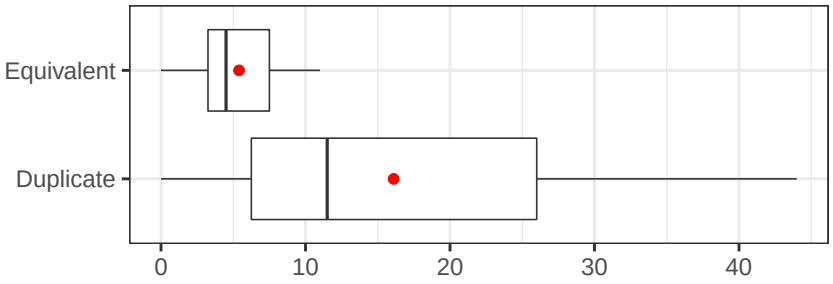


Figure 4.6. Average amount of mutants tagged as equivalent and duplicate per app

[4]Amount of mutants required to preserve the proportion between mutant operators when selecting a sample set

4.3.3 Duplicate and Equivalent Mutants

Regarding the duplicate mutants, we processed the output of `MutAPK 2.0` and found that for our apps dataset there was on average 16 duplicate mutants per app, being *com.nloko.android.syncmypix* the top-1 app with 44 mutants (∼2% of the mutants in the app).

In terms of equivalent mutants, we found ∼5.4 cases per app, being *com.eleybourn.bookcatalogue* the one with more equivalent mutants (*i.e.,* 11 out of 6990).

5

Mutation Testing: Discussion and Summary

The presented work aims to deal with the lack of (i) a thorough catalog of Android-specific mutant operators at APK level, (ii) an analysis of the applicability of state-of-the-art mutation tools on Android apps, and (iii) a characterization of the pros and cons of conducting mutant generation at source code or APK level. Leveraging the fault taxonomy created by Linares-Vásquez *et al.* [93] and focusing on Android-specific faults, we created 38 mutant operators applicable at APK level based on the android mutant operators proposed by Linares-Vásquez *et al.*.

Based on the proposed operators, we conceived and implemented a first version of a framework for the mutant generation of Android apps called MutAPK. This proposed framework injects mutations directly into the APKs. In addition, we conducted a study comparing Mdroid+ and MutAPK with other Java and Android-specific mutation tools. The study results indicate that both Mdroid+ and MutAPK, as compared to existing competitive tools, (i) can cover a more significant number of bug types/instances present in Android apps; (ii) are highly complementary to the existing tools in terms of covered bug types; and (iii) generate fewer trivial, non-compilable, equivalent and duplicate mutants.

Nevertheless, MutAPK generated around ten times more mutants than MDroid+. As mentioned before, our effort was focused on the generation process, leaving open the execution and analysis of the results of the mutant analysis. However, executing and analyzing around 1.3K mutants (the average amount created by MutAPK) is cumbersome and time-consuming. Therefore, reducing dead-code, equivalent and duplicate mutants reduces the number of mutants to be tested. Thus, to reduce the mutation testing effort required with APKs, we also presented `MutAPK 2.0`, an improved version of the original `MutAPK` tool. `MutAPK 2.0` removes dead-code mutants and duplicate and equivalent mutants by following the TCE approach by Papadakis *et al.* [114]. In addition, `MutAPK 2.0` allows for mutant selection using three sampling techniques. To the best of our knowledge `MutAPK 2.0` is the only tool having these capabilities.

Regarding future work, researchers can use and extend MutAPK 2.0 since it is an open-source project on GitHub. Based on the capabilities enabled during the implementation of MutAPK 2.0, researchers can use the different models generated to analyze and generate additional approaches. For example, modifying apps at the APK level enables a faster in-

jection of changes in the app to validate the impact of such changes in the execution of the app. This reduces the time required to test implementations of different changes since no decompilation of the app is required. This behavior, for example, can be used for testing the internationalization of Android apps by identifying hardcoded strings and analyzing the effect of extracting such hardcoded strings for the app.

An approach researchers might work on (by extending MutAPK) is the creation of high-order mutants, in which two research paths might be followed; first, focused on creating mutants with more than one fault injected to reduce the amount of executed "mutated" apps, therefore, reducing the execution time of the complete mutant set. This type of mutant can be created taking into account the call graph identified with MutAPK 2.0 to inject mutations in stems of the call graph that may be independent, providing mechanisms to test at the same time parts of a test suite focused on different components of the app. The second research path researchers might follow is injecting more than one fault on closely related components based on the call graph, by emulating changes that affect different components but are related by the execution flow; this behavior might emulate errors injected by developers.

When this document was written, the articles mentioned in this chapter had 25 citations altogether; after removing duplicates, the thesis, and cites from our articles, eight citations remained. Jeong *et al.* [132] proposed an approach to test embedded software by performing a thorough examination of dependencies; they used MutAPK to generate mutants of an AUT to evaluate the performance of their approach. Silva *et al.* [133] present a mapping study on the usage of mutation testing for Android applications. Based on their search, MutAPK is the only approach that works with SMALI. Sanchez *et al.* [134] presents a systematic review of mutation approaches available in Github; their review analyzes the objective of each mutant tool in their corresponding domain. Lin *et al.* [135] propose an automatic test generation approach that uses a test case as input and generates tests of the same feature following alternative paths on the UI. In order to evaluate the effectiveness of their approach, they created a set of mutants using MutAPK and executed the generated test to identify the mutated applications. Cooper *et al.* [136] presents an approach to detect duplicated video-based bug reports by extracting visual and textual information. They used MutAPK to generate recorded mutants while exploring and generating "mutated" video artifacts. Salma presents a survey on developers' tools for designing, implementing, testing and maintaining User Interfaces, where she analyzes different approaches, their benefits, and their limitations. Similar to what we reported, she highlight the limitation on the understandability of the results generated by MutAPK. Polo-Usaola *et al.* [137] study the impact of different cost reduction techniques on the execution time. Regarding MutAPK, they analyzed and extracted good practices done in the TSE article to compare and generate mutant operators. Marinho *et al.* [138] presents a review of approaches focused on testing resource-related failures in Android apps. They mention our studies during the introduction stating that the Android domain is constantly evolving. Zampetti *et al.* present an interview-based study in which they analyze the role of CI/CD in the design and development of Cyber-Physical Systems. They mention our articles to reinforce the requirement of designing and implementing domain-specific approaches.

Part III

Internationalization of Android Apps

Structure of Part III: Internationalization of Android Apps

- First section outlines the main concepts and previous work related to Internationalization of Android Apps.
- **Chapter 6** presents our study on automated detection of i18n issues in Android apps.
- **Chapter 7** depicts our efforts towards enhancing our tool to provide capabilities for reporting and reparing i18n issues in Android app.
- **Chapter 8** presents a summary of findings, contributions and open research opportunities created as result of our work regarding i18n of Android apps.

Software applications contain textual information that is displayed to users via command lines, GUI, exceptions, messages, logs, etc. When applications are expected to be deployed/delivered in different languages (English, Spanish, Chinese), *internationalization (i18n)* and *localization (l10n)* mechanisms should be included in the apps, to display the textual information in the language of the host device (*e.g.,* a mobile device or a web browser running on a laptop) without installing additional software. While *i18n* is focused on adapting textual information to different languages and regions, *l10n* focuses on the specific requirements of a particular region.

A general and widely used mechanism for internationalizing software applications, independently of the type of app, consists of using textual files — one for each of the target languages — with key-value pairs for the internationalized strings. In those files, a key is an identifier for referencing a string from the app code, and the value is the corresponding literal that will be displayed for a host language. For example, in the case of Android apps, an internationalized string definition in English looks like this:

```
1                    <string name="login_cancel_button">Cancel</string>
```

For Android apps, the internationalized strings are located in XML files stored in the `/res` folder of an app bundle. For each language, there should be a folder, containing the corresponding `strings.xml` file, *e.g.,* `/res/values-fr/string.xml` for French (`fr`) and `/res/values-ja/string.xml` for Japanese (`ja`).

Current approaches for automated support of internationalization have focused on five tasks [139, 140, 141]: (i) detection/location of strings that need to be translated, (ii) extraction of hard-code strings to resource files, (iii) translation of strings to target languages, (iv) automated detection/location of IPFs, and (v) automated fixing of IPFs. However support for internationalization tasks in Android apps is limited to off-line translation, detection of non-internationalized strings, and unit testing of layouts with mock data or pseudo-languages. Nowadays, there is no comprehensive approach, for web or Android apps, able to automatically (i) detect strings to be internationalized, (ii) create internationalized versions of the app under analysis, (iii) explore the app to build layout graphs of the apps, and (iv) detect *i18n* related changes in the GUIs. In addition, to the best of our knowledge, there is no empirical study aimed at reporting the amount and types of *i18n* changes and bugs in Android apps.

6

Detection of Internationalization Issues

Mobile markets allow developers to easily distribute mobile apps worldwide and collect complaints and feature requests in the form of user reviews and star ratings. Therefore, internationalization (*i18n*) of apps is a highly desired feature, which is currently supported in mobile platforms by using resources files with strings that can be internationalized manually. This manual translation can be a time consuming and error-prone task when the app is targeted for different languages and the amount of strings to be internationalized is large. Moreover, the lack of consideration of the impact of internationalized, strings can drive to collateral (*i.e.,* unexpected) changes and bugs in the GUI layout of apps.

In this chapter, we present an empirical study on how i18n can impact the GUIs of Android apps. In particular, we investigated the changes, bugs and bad practices related to GUIs when strings of a given default language (*i.e.,* English in this case) are translated to 7 different languages. To this, we created a source-codeless approach, `ITDroid`, for automatically (i) translating strings, and (ii) detecting bad practices and collateral changes introduced in the GUIs of Android apps after translation. `ITDroid` was used on a set of 31 Android apps and their translated versions. Then, we manually validated the i18n changes that introduced bugs into the GUIs of the translated apps. Based on these results, we present a taxonomy of i18n changes and bugs found along in the apps as well as implications of our findings for practitioners and researchers.

Structure of the Chapter

- Section 6.1 provides motivation for this chapter.
- Section 6.2 presents the architecture of the tool created for the study
- Section 6.3 presents the design of our study, as well as the data extraction procedure and analysis methodology.
- Section 6.4 discusses our results and findings.
- Section 6.5 presents the threats that affect the validity of our work.

Supplementary Material

All the data used in this chapter as well as our tool are publicly available. More specifically, we provide the following items:

Online Appendix[142]

The online appendix includes the following material:

- **Source Code.** `ITDroid` source code is available as Gitlab and GitHub repositories.
- **Taxonomy.** Taxonomy of faults identified on Android apps.
- **Usage instructions.** Detailed instructions to install and use ITDroid.
- **Replication Package.** Access to a folder with the artifacts and results of our study.
- **Empirical Study.** Additional information supporting the results presented in Section 6.3

Publications and Contributions

An empirical study of i18n collateral changes and bugs in guis of android apps.[14]
Escobar-Velásquez, C., Osorio-Riaño, M., Dominguez-Osorio, J., Arevalo, M., and Linares-Vásquez, M. In *Proceedings of the 36th IEEE international conference on software maintenance and evolution (ICSME), 2020.*

Taxonomy of *i18n* collateral changes and bugs exhibited on the 31 analyzed Android apps. → Fig. 6.4
We present and discuss a taxonomy of changes and bugs we built based on the results obtained as part of the study conducted in the previous article.

ITDroid: A tool for automated detection of i18n issues on android apps.[12]
Escobar-Velásquez, C., Donoso-Diaz, A., and Linares-Vásquez, M. In *Proceedings of the 8th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft), 2021.*

ITDroid → Section 6.2
We present `ITDroid`, an open source tool for automatically detecting **i18n** bad practices and collateral changes introduced in the GUIs of Android apps.

6.1 Introduction

Mobile apps are a type of software application that has no comparative in terms of adoption in human daily activities, nowadays. The distribution model provided by online app markets has pushed users and developers towards a new dynamic in terms of release engineering practices, apps consumption in mobile devices, and requirements elicitation [143, 144, 145]. Mobile markets allow developers to easily distribute mobile apps worldwide and collect complaints and feature request in the form of user reviews and star ratings at an unprecedented rate. Therefore, mobile app developers should be more aware of existing practices for making their apps accessible worldwide, which includes internationalizing the apps, but, without impacting the quality as perceived by users, and without being a "show-stopper" for the development process.

The term Internationalization (*i18n*) refers to mechanisms for adapting software applications to different languages. From the perspective of automated software engineering, internationalization-related tasks can be summarized in (i) detecting and translating strings that need to be internationalized, and (ii) testing that *i18n* efforts do not introduce bugs in the internationalized apps. Current practices for internationalizing software applications consist mostly on extracting typically hard-coded strings to resources files, and then, creating internationalized versions of the strings in the default language; it means, there should be a resource file for each language (included the default one), containing the texts that are displayed in the GUI. Mobile development IDEs provide wizard-based features for detecting hard-coded strings and generating (manually) the internationalized versions, however, there is no feature in the IDEs for automatically translating the strings. Recent approaches proposed by Wang *et al.* [146, 147, 148, 149] automatically detect need-to-translate strings, and translate the strings but in an off-line mode, *i.e.,* the strings are translated outside and not incorporated into the app.

On the other side, automated *i18n* testing of mobile apps has not been widely explored. Automated testing of mobile apps have focused on functional bugs [80, 131, 150, 151, 152], performance issues [153], vulnerabilities [154, 155], among other type of bugs. In spite of those efforts, in the case of *i18n* bugs in Android apps, there is no previous study on this type of bugs and there is no available approach for automatically detecting the bugs on Android apps.

Previous efforts on detecting *i18n* issues has been done but on the domain of web applications [140, 141, 156], which provide interesting insights and learned lessons. However, transferring the proposed approaches to the domain of Android apps requires to consider the specifics of the Android programming model. Therefore, in this paper we first describe a novel approach (`ITDroid`) for automatically detecting *i18n* changes in Android apps in a source-codeless fashion (*i.e.,* without having access to the apps source code), and then, we analyze different aspects of *i18n* changes detected in a set of 31 Android apps and their automatically internationalized versions to 7 different languages. `ITDroid` combines APK static analysis, automated translation of strings, and dynamic analysis techniques (*i.e.,* GUI ripping and automated replay), to identify violations of GUI constraints when simulating apps execution with different languages. `ITDroid` also detects strings hard coded and declared

in resource and code files that are not internationalized. The proposed approach operates in a source-codeless fashions, thus, it is agnostic of the native language used for creating the app (*i.e.,*, Java and Kotlin) because the analysis is done at the APK level.

6.2 ITDroid

With the purpose of providing developers and researchers with an automatic way to detect *i18n* changes in Android apps, in this section we describe the architecture and workflow of our `ITDroid` approach. `ITDroid` aims at:

1. Detecting non-internationalized strings (in source code and resource files),
2. Generating internationalized versions of an app under analysis,
3. Exploring the original version of an APK to generate a reference collection of GUI states (represented as a tree of layout graphs) and a replayable scenario of the execution,
4. Replaying the exploration scenario while using the app with different languages, and
5. Reporting internationalization (*i18n*) changes found on the internationalized versions.

`ITDroid` was designed to work with Android Packages files (APKs), which enables analysis when no source code is available, *i.e.,* `ITDroid` works on apps created with different native languages (*i.e.,* Java and Kotlin). Working with APK files instead of source code, reduces time overload when compiling source code to APK files.

Different than using mock-data or pseudo-languages, our approach automatically translates non-internationalized strings to a list L of targeted languages.

Fig. 6.1 depicts the `ITDroid` architecture and workflow. Note that the whole workflow is automated. We combine different techniques such as static analysis of APK files, automated GUI ripping and replay, and (*i18n*) changes detection. We describe each one of the techniques and components as follows.

6.2.1 Preprocessing

In order to obtain a representation of the APK under analysis, we rely on the SMALI representation that can be extracted from APK files. Our choice for SMALI is because it has been recognized as one of the top representations used for static analysis of APKs [157, 158]. `ITDroid` uses the apktool[115] library to extract the decoded resources and a SMALI representation of the dex code.

By using static analysis over the processed APK, `ITDroid` locates all the hardcoded strings. In SMALI, string literals are declared via `const-string` instructions (see Listing 12.1). Therefore, `ITDroid` generates the AST of the SMALI representation and by using a visitor pattern, it goes through all nodes looking for instructions that match the desired expression. After all hardcoded strings (HCS) are located, those are reported, and grouped by method and class. It is worth noticing that hardcoded strings are not translated.

Listing 6.1. Hardcoded String Example

```
1                   const-string v1, "mileage-export"
```

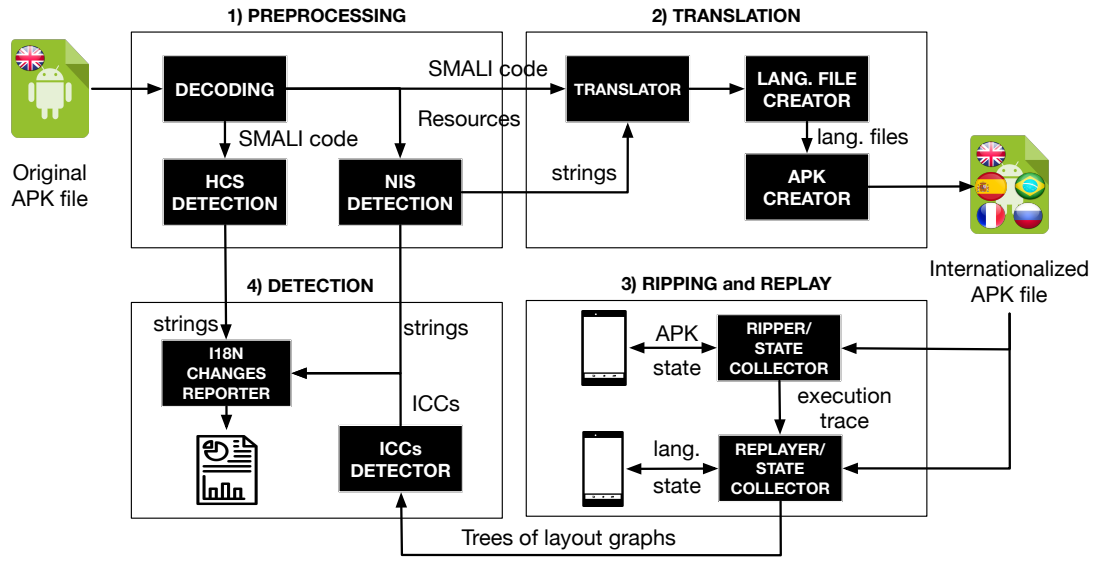


Figure 6.1. `ITDroid` architecture and workflow

Additionally, `ITDroid` identifies strings in the resources of the app that are not internationalized. `ITDroid` compares the `string.xml` file of the default language (*i.e.,* English) with the existing files (if any) of each target language. It is worth noting that there is a set of strings automatically added and translated by the Android SDK, therefore, `ITDroid` removes those strings from the analysis. The result is then a list of tuples $< lang, string\_id >$ that describes non-internationalized strings (NIS), with $lang$ being a target language in L, and $string\_id$ the id of an existing string in the default language but not internationalized to $lang$.

6.2.2 Translation

After identifying the non-internationalized strings in the app resources folder, `ITDroid` proceeds with the translation. By using a strategy pattern, `ITDroid` delegates the translation to an external engine through an interface. This design decision guarantees that `ITDroid` is agnostic to the translation engine. For our initial implementation, we consumed the IBM Watson Translation service, since it has a free and easy to use API.

Before doing the translation, `ITDroid` preprocesses the strings to replace tokens like *%s, %d*, which cause problems in the translation process. Once the strings in NIS are translated, the results are used to build corresponding `string.xml` files for each of the target languages in L. The files are included into the decoded original APK, and then repackaged into an internationalized APK (A^I). Note that the new version of the app is ready to be installed and tested using any of the selected languages.

6.2.3 Ripping & Replay

Our approach includes a regression testing-based approach for automatically identifying Internationalization Collateral Changes (ICCs) in an internationalized APK (A^I) . An Internationalization Collateral Change (ICC) is a change introduced in the GUI of a mobile app when the app is internationalized and executed in a device (physical or emulator) that is configured for a language different than the default app language. Note that Alameer *et al.* [140, 141] use the terms *layout failures* and *internationalization presentation failures* to describe presentation issues introduced in a GUI when internationalizing an application. However, in this paper we introduce the term Internationalization Collateral Changes (ICCs), because not all the induced changes produce bugs in a GUI. Therefore, we prefer to use *collateral change* instead of *failure* to describe changes induced in GUIs when using internationalized strings.

`ITDroid` collects GUI states in both A and A^I, assuring that (i) the same scenarios are executed automatically in both versions, (ii) the execution is independent of the existence of automated tests, and (iii) there is no need of human-collected replayable scenarios. Therefore, `ITDroid` automatically explores the GUI of A on an Android emulator, by following a DFS-based approach widely used in Android GUI rippers [81, 159, 160, 161]. Our approach uses a systematic exploration algorithm that traverses an app GUI, extracting a snapshot of the different UI states and processing those to identify the elements that can be exercised in the current view. Since a view can have several changes due to the execution of events on its elements, `ITDroid` uses a state-based execution that stores the characteristics of the current state of the view and generates a set of unique states while exploring the app. Because of this, `ITDroid` identifies the events that trigger new states during the exploration. Consequently, `ITDroid` reports the sequence of events that were generated during the app exploration.

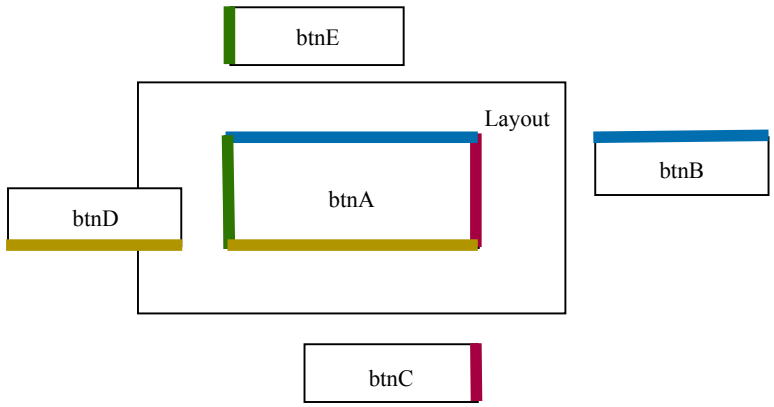
Additionally, to avoid manually exploring the internationalized APK, our ripper implementation replays the sequence of events, generated on the original APK, on emulators automatically configured by `ITDroid` to use the target languages. Therefore, the internationalized APK is explored seven times, one time for each target language

Since the set of steps that produce a change of state are already identified from the original exploration, when replaying the exploration, our tool does not execute all the intermediate events that did not trigger a new state; this fast replay that avoid useless events, allows `ITDroid` to improve the time required to explore an internationalized version of the app. During the ripping, the GUI states are stored in a tree structure, where each node represents a GUI state, and the edges are transitions that lead to different GUI states. Each node in the tree stores the state id, a screenshot of the GUI, and the list of GUI components. It is worth noticing that while replaying events, we ignore the text content of the GUI elements, to avoid miss-identification of elements due to the language change in the device.

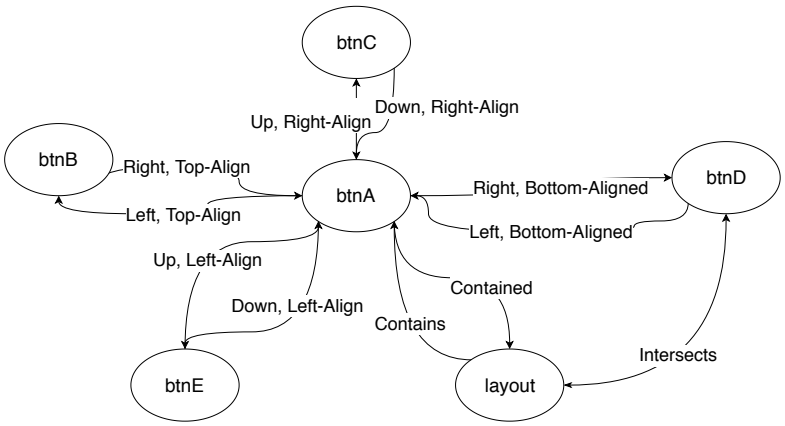
To avoid reproducibility issues, we assure a cold-start scenario, *i.e.,* before a each execution (either ripping o replay) the emulator is restarted, the app is always un-installed/installed and its local data is deleted. Also, `ITDroid` does not generate replayable steps that are coupled to components locations, which can lead to missing steps; we instead rely on locating the components via a composed id (xpath, and element id).

For identifying ICCs, `ITDroid` models the states by using the layout graph representation

proposed by Alameer *et al.* [140, 141]. A layout graph (TLG) describes the existing relationships between all components of a given GUI, in terms of **location** (*i.e.,* up, down, right, left), **alignment** (*i.e.,* top-, bottom-, right-, left-aligned), and **overlapping** (*i.e.,* contains, contained, intersects).



(a) GUI components



(b) Layout graph

Figure 6.2. Example of a layout graph: (a) view of GUI components distribution, and (b) corresponding layout graph.

Fig. 6.2 serves as an example for explaining layout graphs. In the example (see Fig. 6.2a) there are 5 buttons and 1 layout component: *btnA* is located at the center of the image and for the purpose of this example, we are going to analyze the relations between this button and the rest of the elements. *btnA* has **location** relations: *btnE* is **up**, *btnB* is to the **right**, *btnD* is to the **left**, and *btnC* is **down**. In terms of **alignment**, *btnE* is **left-aligned**, *btnB* is **top-aligned**, *btnD* is **bottom-aligned** and *btnC* is **right-aligned**. Each of these alignment relations is shown using a different color. Finally, in terms of **overlapping**, there is a **containment** relation with the *Layout* component. As the reader can see in Fig. 6.2a there is also a relation between *btnD* and *Layout* called **intersection**. Fig. 6.2b presents the layout graph ([140,

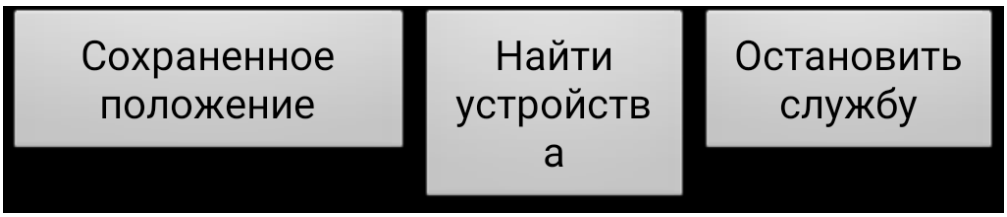
141]) depicting the relations. Note that the graph is bidirectional and the relations can be different under each direction, except for the case of **intersection**.

6.2.4 Detection and Reporting

Once the app has been explored for each language in L, the next step is to process the corresponding layout graphs. Therefore, using the default language' layout graph (TLG_{df}) as a baseline, `ITDroid` goes through each language's layout graph (TLG_{lang}, $\forall lang \in L$) looking for differences with the baseline graph. `ITDroid` starts by trying to find a matching node in TLG_{lang} for each one in TLG_{df}. Once all the possible nodes are paired, the next step is to recognize the relations that have changed between the two graphs. The result of this step is the set of relations added and lost for all pair of nodes in each state. Those changes are then detected as Internationalization Collateral Changes (ICCs).

Fig. 6.3 shows the English (a) and Russian (b) version of a set of buttons from the *a2dp.Vol* app' main activity. Each image contains 4 Android components: 3 buttons and a *linear layout*. In the English version (Fig. 6.3a) all the components are bottom-aligned –taking into account buttons margin. However, for the Russian version (Fig. 6.3b) only the first and third buttons remain bottom-aligned. Therefore, from the point of view of the left-most button in the Russian version, there are 2 relations that were lost: bottom alignment with the second button and with linear layout.

(a) English version



(b) Russian version

Figure 6.3. Example of an ICC

At the end of the process, `ITDroid` generates a report listing (i) hard-coded strings, (ii) non-internationalized strings declared in the default language but not traduced in the target languages, and (iii) internationalization collateral changes. To easily handle ICCs, `ITDroid` reports the changes by identifying: (i) the exploration state id, (ii) the id of the node, (iii) each node for which a relation was modified along with the details of the modified relations, and (iv) screenshots depicting the GUI state for both default and internationalized languages.

6.3 Study Design

As of today, there is no empirical study analyzing *i18n* bugs in Android apps. Therefore, we used `ITDroid` in an empirical study aimed at analyzing different aspects of *i18n* changes in Android apps. To this, we executed `ITDroid` on the APK files of 31 open source Android apps having English as their default language, with the purpose of answering the following research questions (*RQs*):

- **RQ$_1$:** *What types of i18n collateral changes and bugs are exhibited on GUIs of Android apps?*
- **RQ$_2$:** *What i18n collateral changes induce i18n collateral bugs on GUIs of Android apps?*
- **RQ$_3$:** *What are the GUI components and languages more prone to i18n collateral changes and bugs?*

6.3.1 Context of the study

To select the dataset of apps for this study, we started looking for Android apps used in studies aiming at evaluating testing approaches: DroidMate [162], CrawlDroid [163], MDroid+ [93, 164]). Thus, we created a dataset of 211 apps including the APKs used in those studies, and an internal dataset we built for previous studies. Afterwards, to evaluate whether the apps were valid for our study, we followed these inclusion/exclusion criteria:

- We discarded the apps that were not available at the Google Play Store.
- Given the fact that we were going to execute the apps on an API 27 Android Emulator, we removed apps that were not compatible with the API 27. Note that our choice for running the experiments on an emulator rather than on a physical device is because `ITDroid` is tailored for working with emulators, for instance, `ITDroid` automatically sets the emulator language via ADB commands.
- We included (i) apps with more than one activity, or (ii) apps with only one activity but exhibiting more than one GUI state.

Having into account these criteria we end up with a total of 31 APKs ready to be used with our experiments.

We used English as the default language, and for the target languages we used $L = \{$*Spanish, Hindi, Arabian, Russian, Portuguese, French, Italian*$\}$. These set of languages is based on the list of languages with the largest number of speakers. Chinese, Malay, and Bengali were not used in the study — despite being in the list of top languages — because the IBM Translation Service does not provide translations from English to those three languages. With that list of languages, and for each original APK, `ITDroid` generated an internationalized APK supporting the 7 languages in L, *i.e.,* the analysis was done on 31 original APKs, and 31 automatically internationalized APKs.

6.3.2 Open coding

To answer the *RQs*, we executed `ITDroid` on the 31 original APKs, which automatically generates internationalized APKs, and then used the statistics and information provided by the `ITDroid` report. The `ITDroid` results were validated and analyzed with an open coding inspired procedure. The validation aimed at detecting false positives reported by `ITDroid`, and identifying the ICCs inducing *i18n* collateral bugs (ICBs) in the apps.

The *i18n* changes were manually analyzed by all the authors with open coding sessions that were supported on a web tool we created. For each ICC assigned to a tagger, the tool showed: (i) the type of change; (ii) the GUI components involved in the change; (iii) a side-to-side comparison of the GUI state (*i.e.,* screnshoot) of the original APK and the same GUI state on a internationalized version, while highlighting the components involved in the ICC; and (iv) a set of input fields for selecting whether the depicted case is a false positive or a *i18n* collateral change, and whether the ICC is a *i18n* bug or not.

The tagging process consisted of two stages. First, the complete set of ICCs where distributed between the five authors by ensuring each ICC was assigned at least to two taggers. This first stage required each tagger to select between 3 main categories to tag (*i.e.,* false positive (FP), ICC or ICB).

After this first stage of tagging, we had 273 cases where the taggers agree with all the tags, and 129 conflicts in which there was a disagreement in at least one of the tags.

The second stage, consisted of solving the tagging conflicts. For the conflicts resolution, the cases where distributed between 3 authors. For this process, the web app was modified to show the tags provided by the original taggers; the identities of the original taggers were not disclosed to avoid biasing the conflicts solver.

6.3.3 Analysis method

To answer $\mathbf{RQ}_1$, we built a taxonomy of *i18n* changes and bugs, by analyzing the results reported by `ITDroid` and the open coding process. The `ITDroid` report includes hard-coded strings, non-internationalized strings, *i18n* changes, and screenshots of the changes. In addition to the taxonomy, we provide in Section 6.4 qualitative examples of the *i18n* changes and bugs.

To answer $\mathbf{RQ}_2$ we analyze and report, with representative qualitative examples, the cases marked as *i18n* bugs during the coding phase. In the case of $\mathbf{RQ}_3$, we used the frequencies and statistics collected for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, and report the results grouped by (i) GUI component types involved in *i18n* changes (*e.g.,* button-label, label-image, layout-button), and (ii) language where the *i18n* changes and bugs were detected.

Note that the frequencies in the taxonomy report the *i18n* changes detected by `ITDroid` in the 7 targeted languages (overall). It is also worth noting that because an *i18n* change is a relationship between two GUI components $GC1$ and $GC2$, there is a dual nature. In the direction of $GC2 \rightarrow GC1$ the relation could be different than in $GC1 \rightarrow GC2$. For example, in the $GC1 \rightarrow GC2$ case, the change could be a miss-alignment, but in the case of $GC2 \rightarrow GC1$ the change could a position-related one. Thus, when reporting frequencies, if the change is the same in both directions we counted it as one instance, otherwise, there are two different *i18n* changes.

All the details of the apps, original and internationalized APKs, the reports generated by `ITDroid` with the internationalization changes, and the `ITDroid` code are publicly available within our online appendix[142].

6.4 Results & Findings

In this section we report the answers for each one of the research questions in our study. We present and discuss our findings by using a taxonomy of changes and bugs we built with the results reported by `ITDroid` and the open coding phase. Fig. 9.5 presents the taxonomy in which we distinguish *i18n* changes from *i18n* bugs. Concerning the later, there are bugs introduced by developers because the lack of *i18n* practices (*e.g.,* hard-coded strings), and collateral bugs induced by the *i18n* collateral changes. The discussion of the results includes visual examples of the changes and the bugs exhibited by the analyzed apps. In addition, we highlight with ***bold and italic*** the learned lessons and implications for the developers and researchers communities. All values presented in this section do not include the false positive found during the open coding process. In particular, we found 30 false positives out of 402 ICCs reported by `ITDroid`.

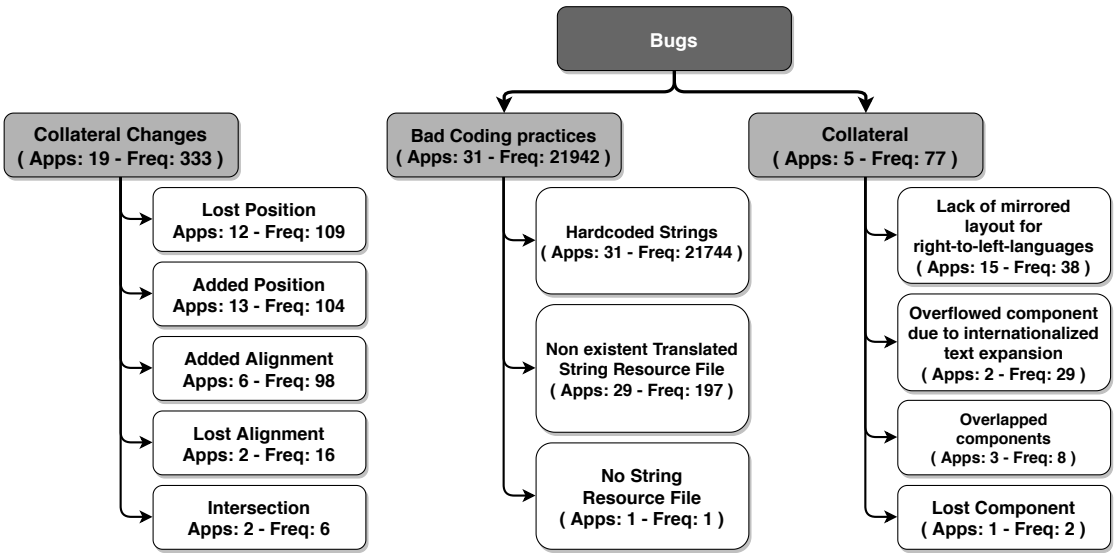


Figure 6.4. Taxonomy of *i18n* collateral changes and bugs exhibited on the 31 analyzed Android apps. Each box in the taxonomy reports the number of apps with the change/bug, and the number of times (Freq:) the change/bug was detected.

6.4.1 RQ_1: What types of i18n collateral changes and bugs are exhibited on GUIs of Android apps?

We found that *i18n* collateral changes and *i18n* code bad practices are more frequent than collateral bugs in the analyzed apps. In the following we describe the results for each cate-

gory of changes, bad practices, and bugs listed in Fig. 9.5.

***i18n* Collateral Changes (ICC)**. Based on the layout graphs proposed by Alameer *et al.* [140, 141], `ITDroid` found instances of 4 types of changes. The most frequent ones are related to components position: *lost position*, *added position* and *intersection*. It is worth noticing that the values of *i18n* changes reported in the the taxonomy refer to individual changes in the apps GUI. Therefore, one action such as shifting an element to other line can have different individual changes, *e.g.,* a case of lost position (*right or left*), a case of added position (*below or above*) and finally, a case of added alignment (*left_aligned or right_aligned*).

Fig. 6.5 is a representative example of the *lost position* category, in which we can see how the **right** relation between the “close app" and “ask me again" buttons is lost when the internationalized texts in Portuguese expand. We found this type of change in 109 cases (12 apps). Fig. 6.5 also represents a case of *added position*, because there is a change from **right** to **bottom** between the “close app" and “ask me again" buttons. We found added positions in 104 cases (13 apps).

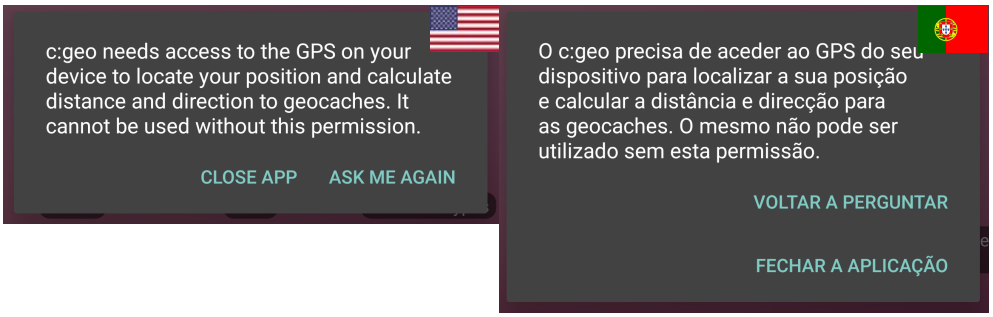


Figure 6.5. Example of added and lost relations in the *cge.geocaching* app.

Besides lost and addition of positions, we found 6 cases in which changing to a non-default language generated *intersections* that did not exist in the original APKs. There are two types of intersection: *overlapping* and *breakage of parent-imposed constraints*. *Overlapping* happens because a containment relation is introduced between two components when the app is used in a non-default language. An example of this behavior is shown in Fig. 6.6: in the English version, the “Select Preset" label and the down-arrow icon are next to each other, however, when language is changed to Arabic, the constraints of the text are not well defined and the text overlaps with the icon.

Figure 6.6. Example of an intersection relation in the *com.adermark.forestpondfree* app

The *breakage of parent-imposed constraints* type, groups the cases where a containment relation is lost. Specifically, an intersection relation replaces the lost one, generating a behavior like the one presented in Fig. 6.7.

Figure 6.7. Example of an overlapped component in the *org.ethack.orwall* app

Concerning changes related to components alignment we found 114 instances: 98 cases of *added alignment* and 16 cases of *lost alignment*. Fig. 6.8 presents an example of *added alignment* between the second and third buttons (from left to right), where after translating to Portuguese, the text content of the third button expands and fills the height of its container. This results in a new bottom-alignment with the second button. However, in the English version the buttons were not aligned.

Figure 6.8. Example of an added alignment relation in the *com.android.logcat* app.

Opposite to the previous case, the *lost alignment* category gather all the cases where a relation is removed. For example, Fig. 6.9 shows a case where most of the buttons lose its top-alignment relation with the first button, when the app is explored in Hindi.

Figure 6.9. Example of a lost alignment relation in the *com.android.logcat* app.

***i18n* Bad Coding Practices (IBCP).** `ITDroid` found 21942 cases of bad coding practices in the 31 analyzed apps. The bad practices are organized in three groups: *hard-code strings*, *non-existing string resources*, and *non-existent internationalization file*. The *hard-coded string* practice refers to strings that are placed directly in the source code. From a internationalization point-of-view, this is a bad practice done by developers, since those strings do not change when the device language is modified. The most common appearance of this behavior is to set a value from Activities or Fragments code into a widget. This bad practice is exhibited 21744 times in the 31 analyzed apps.

The *non-existent string resource* bad practice occurs when an app has no `strings.xml` file. This leads to the extreme case where all the strings in the application are hard-coded, hindering the creation of an internationalized version of the app. In our study we identified only one app with this behavior: *com.example.android.musicplayer*.

The lack of `strings.xml` file can be also extended to the corresponding files for non-default languages. We called this case as the *non-existent internationalization file*. When

Figure 6.10. Example of lost content in *com.teleca.jamendo* app. Due to space limitations, this example contains a snippet of the full screen size example that can be found in our online appendix.

considering the 7 target languages used in this study we found that in the analyzed apps, in 29 apps least one of the languages in the list is not internationalized.

The prevalence of these bad coding practices suggests that developers do not use the utilities available at the Android Studio IDE for detecting and extracting hard-coded strings. ITDroid is an option for detecting these bad practices, outside of the IDE and without the need of having access to source code. However, if developers prefer tools embedded in the IDE, we encourage them to use the existing features, and customize detection rules for non-existent string resources and non-existing internationalization files (extending the Lint tool available with Android Studio).

i18n* Collateral Bugs (ICB)**. We found 4 types of ICBs induced by *i18n* collateral changes, which group a total of 39 bugs belonging to 5 apps. The most prevalent type of ICB is *overflowed component due to internationalized text expansion* (29 instances). This type of bug is visible in GUIs because it breaks the original design when large texts push the components to be out of their expected dimensions, positions and alignments. For instance, Fig. 6.5 shows how in the *cgeo.geocaching* app, when changing to Portuguese, the horizontal arrangement of the dialog buttons is pushed to be vertical because of internationalized text expansion in the buttons. When looking into the details of the *cgeo.geocaching* app's layout we found that the bug is exhibited by an Android *AlertDialog*. ***Therefore, developers must be aware that even Android composite widgets are prone to i18n bugs.

We also found that using linear layouts instead of constraints layout is a common error in Android apps. Junior developers prefer linear layouts because are easier to use; constraint layouts are more complex to handle when there is no deep knowledge of the available constraints. ***In addition to being performance friendly, constraint layouts can help developers to avoid issues when dimensions of GUI components are modified dynamically. Therefore, developers should be knowledgeable of the constraint types and be aware that changes in text lengths can break the layout drastically, in particular when changing the default language.***

Other types of ICB, but less frequent, are *lost component*, *lack of mirrored layout for right-to-left-languages*, and *overlapped components*. The former type relates to cases in which a visi-

ble component, is pushed out of the display view because a text component is re-dimensioned (see Fig. 6.10). This does not seem to be a problem at first sight, however, this type of issue could hinder the execution of automated tests that expect certain components to be visible. The *lack of mirrored layout for right-to-left-languages* is a very specific bug that is produced when developers do not consider bidirectionality in their layouts. Bidirectionality means that for languages that read from right-to-left (RTL), UIs should be mirrored to ensure understandability. One example of this bug is Fig. 6.11. ***To avoid this type of bugs, developers should follow bidirectionality guides that describe how to mirror layouts at the design concept and implementation levels [165, 166]. Static tools can be a solution here, by automatically analyzing and implementing the bidirectionality and RTL guidelines.***

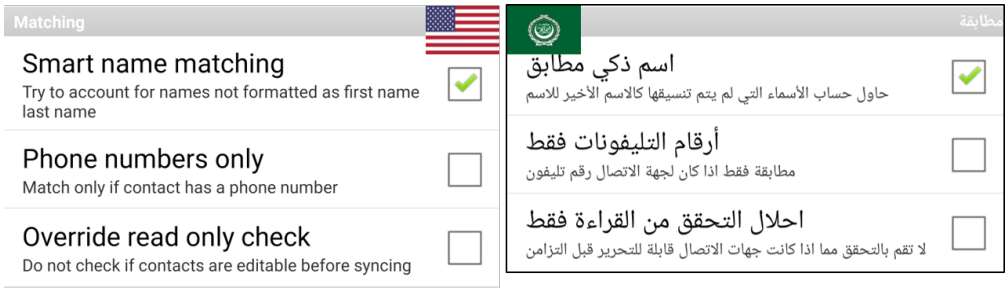


Figure 6.11. Example of a lack of mirrored layout for right-to-left–language in the *com.nloko.android.syncmypix* app.

Finally, the *overlapped components* bug is mainly caused by the lack of proper constraints between two elements. This bug can happen between two aligned elements, as it can be seen in Fig. 6.6, and between an element and its container (see Fig. 6.12).

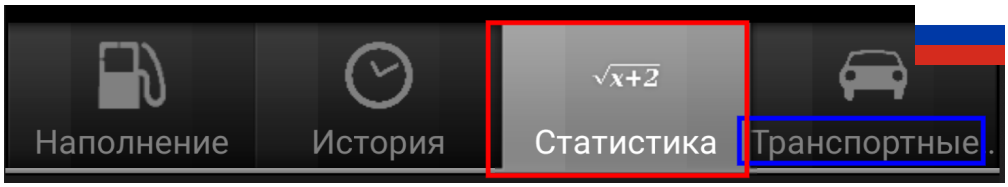


Figure 6.12. Example of an overlapping element and its container in *com.evancharlton.mileage* app when internationalized to Russian. The red and blue squares show the overlapping relation

In general, most of the reported ICBs are caused by incorrect definition of constraints on components with text and the lack of tools supporting automated detection and fixing. The bugs induced by internationalized text expansions can also be easily fixed by using the `ellipsize` attribute of the GUI components. In addition, although, `ITDroid` is a partial solution that helps developers to detect IBCPs and ICCs, they have to manually go over the ICCs reported by `ITDroid` to identify ICBs. Therefore, future work could focus on extending the `ITDroid` approach for automatically detecting the bugs by relying on automated image-based comparisons or by statically detecting (i) constraints

incompatibilities and issues, and (ii) missing configurations and resources for enabling bidirectionality.

Other potential impact of the IBCPs and ICBs described here, but not investigated in our study, is related to the behavior of screen readers when IBCPs and ICBs are exhibited in internationalized apps explored by users with visual disabilities. Although it is an aspect not deeply investigated yet, an empirical study by Vendome et al. [167] reports that internationalization of assistive content in Android apps is a concern expressed by some developers at Stack Overflow.

6.4.2 RQ_2: What i18n collateral changes induce i18n collateral bugs on GUIs of Android apps?

Overflowed component due to internationalized text expansion. This type of bug is generated in most of the cases by position-related changes, since the element that is shifted to a new line lose all its position relations with the elements that were in the same line. An example of this behavior is Fig. 6.5. Because the button expanded into the next line, the position alignments between the “ask me again" and “close app" buttons are lost, and instead a new **below/down** relation is added.

Lost Component. For this type of bug, the 2 cases we found are related to internationalized text expansions that push other components out of the user view. In the layout graphs, this is represented as a lost position of the components pushed out of the view, since those components (after the change) are not visible anymore on the UI. The example in Fig. 6.10, shows how there is a button at the end of the English version, but, the button is not visible in the French version. This is represented in the LayoutGraph as an element that does not appear in the graph of internationalized version.

Lack of mirrored layout for right-to-left-languages. This type of bug is mainly represented as the lack of of alignment and position changes, since layouts for RTL languages should mirror the position of most of its elements. For example, all **right** alignment and position relations should became **left** alignment and position relations. Note that we found only 2 cases with `ITDroid` during the open coding of the ICCs. However, such a small number of cases is explained because the layout graph-based approach [140, 141] used by `ITDroid` focuses on detecting changes between layout graphs. The *lack of mirrored layout for RTL languages* is in fact a bug that should be identified when no changes are detected (*i.e.,* there is no mirroring) in components such as [165]. Therefore, we manually analyzed the screenshots collected by `ITDroid` when the emulator was configured for Arabic. We found 15 apps suffer from this bug (see Fig. 6.11). ***In order to detect this type of bug, for the specific case of RTL languages, the layout graph-inspired detection should look for components not changing positions.***

Overlapped components are exhibited mainly by intersection and lost alignment relations. This bug considers all the cases where one element overflows its container, for instance, the existing alignments of right and left borders between contained elements and its container are broken.

6.4.3 RQ_3: What are the GUI components and languages more prone to i18n collateral changes and bugs?

Fig. 6.13 presents the amount of ICC and ICB distributed along the languages used with our study. Arabic is the language with more ICCs accounting for a total of 151 collateral changes, nonetheless, 39 ICCs were considered as ICBs during the manual tagging phase. Most of the ICCs generated when internationalizing the analyzed apps to Arabic, are explained because the lack of awareness of the RTL nature of language, when designing the layouts. As already mentioned in the previous section, ***layouts for Arabic versions of apps should adapt alignment features to achieve bidirectionality; nevertheless this might generate a conflict for developers since it might require to create two different versions of the app, however, since API 17, Android apps accept RTL orientation by defining this behavior in the layout files (see [166]). For example, developers might want to use the `paddingStart` attribute instead of `paddingLeft`, which delegates the orientation to the operating system [166].***

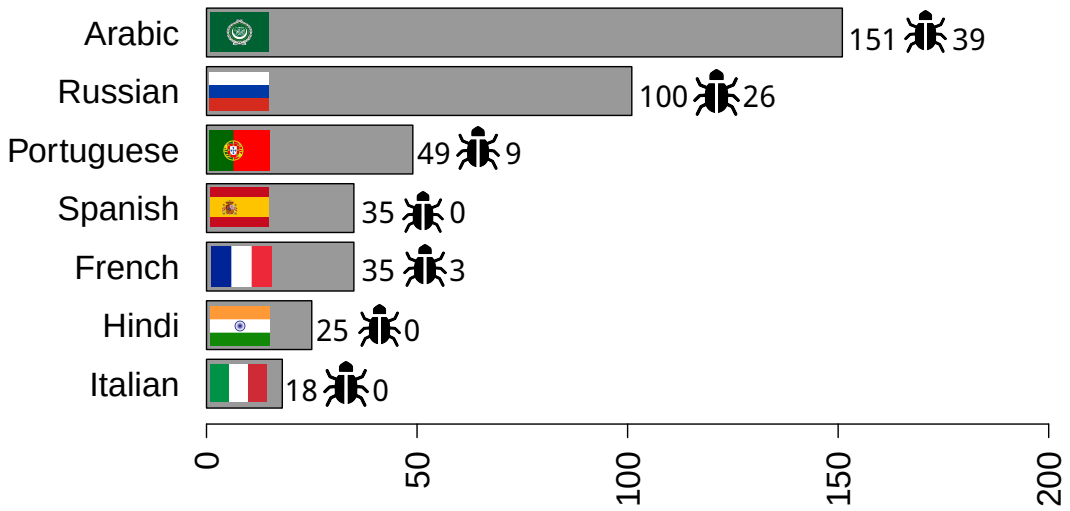


Figure 6.13. Distribution of ICC and ICB in the analyzed languages. The values next to the bar are the amount of ICCs, while the amount shown next to the bug icon is the amount of ICBs.

Russian is the top-2 language involved in ICCs, with 101 cases reported by `ITDroid`. The ICCs generate 26 collateral bugs distributed in *Overflowed component due to Internationalized text expansion* and *Overlapped components* (See Fig. 6.12) As in most of the cases reported in this study, ***the bugs are induced by text expansions that are not properly controlled with layout constraints or by shortening texts using an ellipsis (i.e., using the `ellipsize` attribute [168]). Using English as the default language in Android apps, and not conducting proper i18n testing, makes apps prone to i18n collateral changes and bugs because English is among the languages with smallest words [169].***

Fig. 6.14 reports the distribution of ICC and ICB at the GUI component level. `TextView` is the top component, with 125 ICCs but only 6 ICBs. A similar behavior can be found with `LinearLayout`, where there are 79 ICCs but only 5 were tagged as ICBs. In contrast, `Button` the top-3 component in terms of ICCs, is also the one with most ICFs: 22 of the 73 ICCs where tagged as ICBs. Note that ICCs related to *Buttons* are commonly found in custom `Dialogs` where layout constraints are not properly defined.

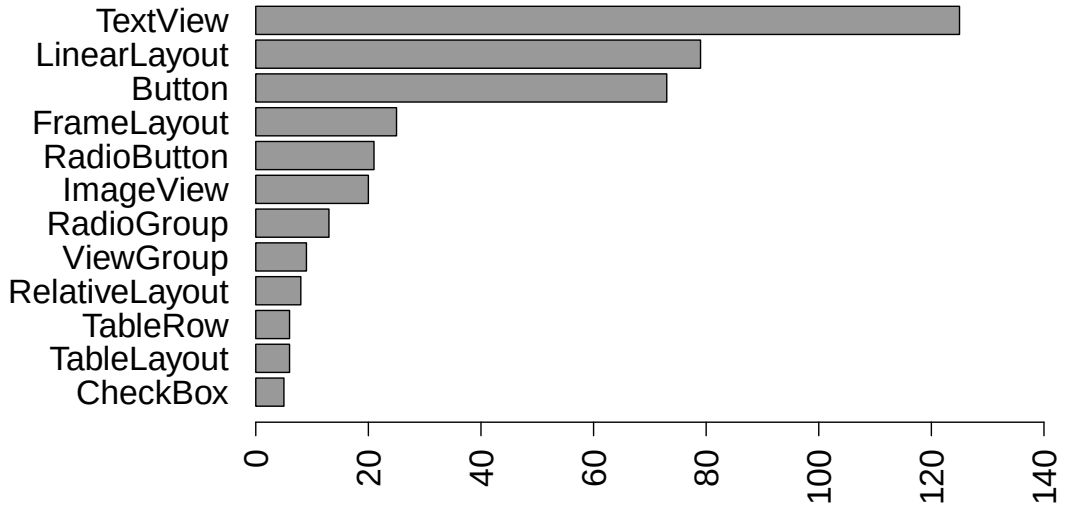


Figure 6.14. GUI components involved in the ICCs detected by `ITDroid`.

6.5 Threats to Validity

External validity. This type of threats are concerned to whether the results can be generalized to other settings or populations. In this study we used a sample of 31 Android native open source apps, which is not representative of the whole population of Android apps; therefore, we can not generalize our results to the whole set of Android apps that include hybrid and mobile web apps. Additionally, it is known that some popular apps have mechanisms to avoid being repackaged, this behavior might change the result of the tool execution.

Future work should be devoted to replicate the study on a larger sample of apps; also, similar approaches should be implemented for Android hybrid and mobile web apps and other platforms such as iOS. In addition, although we analyzed ICCs and ICBs for 7 languages from the top-spoken languages list, we can not claim that our results generalize to other languages with different characteristics, *e.g.,* isolating (Chinese) or agglutinating (German) languages.

Internal validity. Concern confounding factors in the independent variables that can affect the results (*i.e.,* dependent variables). The apps sample is a potential threat, because they could belong to a specific category. However, we reduced this threat by random sampling the apps from a publicly available repository (F-Droid) that has been widely used by other researchers; our sample is diverse in terms of size and categories (see online appendix). Another potential threat is the choice of target language as the independent variable for the number of ICCs and ICBs, however, in our study we executed the experiment on a set of 7 different languages to avoid any bias introduced by the language choice. Finally, the choice of a ripper that follows a DFS strategy is a potential threat to internal validity, because despite of trying to explore as many states as possible, rippers are known to have limitations. Therefore, we recognize that it is possible that there are more GUI states in the apps that were not reached by the ripper. In that sense, our results are reported as a lower bound of the number of ICCs and ICBs in the analyzed apps.

7

Prevention and Fixing of i18n Issues in Android Apps

Correct internationalization of UI is a desired behaviour in mobile apps, as we showed in Chapter 6 this is not fullfilled in real-world apps due to lack of support for commonly used languages and incorrect presentation of content in case of Right-to-Left (RtL) languages. In order to support different languages, developers must add resource files that depict the strings to be used on each specific language. Unfortunatelly, this also requires for developers to use the defined mechanism by Android OS to reference the strings defined in the aforementioned files. Nevertheless, as we present in this chapter, developers commonly define used strings within the source code. Additionally, to support RtL languages developers must use a set of defined properties when defining the UI so the OS can mirror the elements when a RtL language is being used.

In this chapter we present our efforts towards improving `ITDroid` capabilities towards detection and localization of i18n issues, along. At the same time, we present our first results in the automated repairing of i18n issues. In order to achieve the improvements and capabilities extension of `ITDroid`, we enhanced our SMALI processing algorith to identify hardcoded strings within the UI files and improve the identification process over the hardcoded strings found within the source-code. Additionally, we added a

Structure of the Chapter

- Section 7.1 provides motivation for this chapter.
- Section 7.2 discusses the improvements done to `ITDroid`
- Section 7.3 presents our results and findings.

Acknowledgments

The research presented in these chapter was conducted in collaboration with Andrés Donoso-Diaz and Mario Andrade-Vargas, past undergraduate students at Universidad de los Andes, Colombia. The approach presented in this chapter are also part of their undergraduate thesis. [170, 171]

7.1 Introduction

Internationalization of Android Apps is achieved using a set of defined mechanisms. First, developers must define a set of resource files that contains all the strings to be used within the app, in order to support a specific language a resource file must be created following a naming pattern defined by Android OS and must contain a set of key-value registers specifying the possible values of the strings in the supported language. Once this files are created, developers must used references within the source code and layout files to access the values of the strings defined in the aforementioned files. Unfortunately, developers hard-code the strings within the code, tampering the internationalization capabilities of the app as it was shown in the previous chapter. Nevertheless, there are a certain type of strings that might be found inside the code and should not be extracted to the resource files, such as links and references of code components. As it was mentioned in the previous chapter, to the best of our knowledge, the current approach to internationalized Android applications heavily relies on the interaction of developers with psedo-automated tools that solve partially the found issues.

Similar to the aforementioned issue, developers do not have a mechanism to analyze different version of an app according to the supported languages. Therefore, in order to achieve this, developers must executed their test individually over the new language version of the app, increasing the amount of work associated to the quality assurance process of the app. Additionally, since the execution of this tests are independent, practitioners and developers must review separete reports to identify issues.

Finally, as it was presented in previous chapter, internationalization also takes into account the behaviour of the application when used with a Right-to-Left language. According to our results, most of the apps present issues following the standards defined to support Right-to-Left languages. Due to this, we concluded that developers where not taking into account Right-to-Left languages.

In this chapter, following the work presented previously, we present an enhanced version of our tool, `ITDroid`, that supports extraction of strings hard-coded in layout files and generates new string resources within the app files to repair internationalization issues generated by the lack of aforementioned resources and its references. Additionally, we improved our visual report to present an unified dashboard of the execution of the different language version of the app in order to ease the identification of i18n issues by developers. Finally, we enable a first versión of the repair task to fix mirroring issues within Android apps.

7.2 Approach

In this section we describe the changes that we implemented in `ITDroid` to support and enable a new set of functionalities.

7.2.1 Hard-Coded String Extraction

Since developers might use strings within the code to defined routing our specific characteristics of different algorithms, we must process the list of hardcoded strings to recognize the set of strings that must be extracted to new string resources and then replaced within the code to enable internationalization.

Logic Files

Logic files are the smali representation of the original java files after decoding the android app, in these files, we can find hard-coded strings that are going to be displayed in the layouts at some point by following the program's logic, or strings that are used directly in the logic. Strings that are used for the logical execution of the app need different validations as not all of them can be considered hard-coded.

The string processing starts by finding and saving all the necessary information used to validate and, if possible, replace the hard-coded strings. This information comes from the smali files that represent the java classes. In this part of the process, it's possible to discard strings as hard-coded based on the content of the string itself, because there are some know words or letter patterns that shouldn't be translated. For example, URLs and package names.

At SMALI level, classes and methods require explicit definition of the capabliities they have access to, in our case, we must be sure that classes and methods have the permission to query strings located in the *strings.xml* files. Therefore, after identifying a hardcoded string, we must modify the class and method definition to enable aforementioned capability. Once we have enable the access to such resources, we proceed by generating an identifier that will be used within the "logic" files. This identifier must be assigned in the logic file as replacement of the hardcoded string and must also be defined inside the "R.smali" file that contains all the resources id and matches them with their identifier inside the *strings.xml* file.

It is worth noticing, that the hardcoded string that was extracted now is defined only within the defaults language resource file. This is fixed later, when the comparison of XML files is done.

Layout Files

The layout files refer to the XML that contains the base GUI layout for the app. Inside said files, the hard-coded strings are easier to identify because they are explicitly declared by XML attributes.

The processing for the layout files starts by checking all the XML layout files to find XML layout tags that contain any displayed text. This tags are composed normally by a tag and a value, therefore, after identifying the tags with key is related to text manipulation and presentation, we analyze the value of such tags to identify if it references a resource that is defined correctly (*i.e.,* within *strings.xml* file) or if it is a hardcoded string. In case the text is identified has hardcoded, a reference is saved in order to proceed with a batch extraction of this resources following a protocol similar to the mentioned for logic files.

After all the hardcoded strings are located string resources are created within the *strings.xm* file for the default language, and the files containing such hardcoded strings are modified to enable its access to resource files and the the strings are replaced with a new identifier.

7.2.2 Implementation

In this section, each of the steps mentioned in the previous section will be described more technically, as to explain how the solution was implemented. The changes made to the Smali code were done by following the documentation provided by the Smali developer [172].

Locating the Strings

For the location of hard-coded strings, ITDroid already used Abstract Syntax Trees (ASTs) in order to navigate through the code in a more comprehensive and structured manner. ASTs help to branch the class files into all the declared components (methods, variables, constants), so it's more accurate for the purpose of finding the exact hard-coded strings.

To start, it is crucial to filter the tree nodes that follow the syntax of a possible hard-coded string, "const-string". These nodes contain the necessary information for the possible extraction of the strings, including the name of the method containing the string,the local variable referencing the string and the content of the const-string object.

```
(I_STATEMENT_FORMAT21c_STRING const-string v0 "CanvasThread")
(I_STATEMENT_FORMAT21c_STRING const-string v17 "spriteCount")
(I_STATEMENT_FORMAT21c_STRING const-string v17 "animate")
(I_STATEMENT_FORMAT21c_STRING const-string v0 "GLThread")
(I_STATEMENT_FORMAT21c_STRING const-string v5 "j")
(I_STATEMENT_FORMAT21c_STRING const-string v19 "vertsAcross * vertsDown >= 65536")
(I_STATEMENT_FORMAT21c_STRING const-string v19 "vertsAcross")
```

Figure 7.1. Example of tree nodes of potential hcs

Validating the Possibility of String Extraction

After getting the relevant information from potential hard-coded strings, the next step is to filter out the non-hard-coded strings, based on a set of rules; this can be done in any order, as not following one of them is enough for it to be discarded as a hcs.

- **String content matching a non-translatable regular expression:**

  Information mainly used for code and the app's logic (emails, URLs, package names) are positive to be the same in any language, meaning that these strings can't be considered hard-coded, as they are constants. Every cons-string variable gets its content validated by a matcher using regular expressions, as can be seen in Fig7.2.

```java
private static Boolean isExtractable(String str){
    Boolean isTransla = true;
    //All the patterns of strings that should not be reported
    String numPattern = "^-?\\d+\\.?\\d*$";
    String packageName = "^([A-Za-z]{1}[A-Za-z\\d_]*\\.)+[A-Za-z][A-Za-z\\d_]*/*[A-Za-z]*$";
    String url = "(http|ftp|https):\\/\\/([\\w_-]+(?:(?:\\.[\\w_-]+)+))([\\w.,@?^=%&:\\/~+#-]*[\\w@?^=%&\\/~+#-])";
    String email = "^[^\\s@]+@[^\\s@]+\\.[^\\s@]+$";
    String camelCase = "[a-zA-Z]+[A-Z0-9][a-z0-9]+[A-Za-z0-9]*[()]*";
    String path = "([A-Za-z])+/([A-Za-z])+";
    String hyphenatedWord = "[A-Za-z\\d]+(-[A-Za-z\\d]+)+";
    String atString = "^@+$";
    String specialChar = "[^\\u0000-\\u00FF]";
    String camelDotVariation = "([a-zA-Z]+[()]*\\.)+[a-zA-Z]+[()]*";
    Pattern patternReplace = Pattern.compile("[^\\x00-\\x7F]");//Check for non unicode chars
    Matcher matcherReplace = patternReplace.matcher(str);

    if(matcherReplace.find()){
        isTransla = false;
    }
    str = str.replace("\"", "");//escapes the " char in order to get accurate matches
    if(str.matches(numPattern))
        isTransla = false;
```

Figure 7.2. AST

```
(I_SUPER Landroid/app/Activity;)
```

(a) A valid class to extent from

```
(I_SUPER Ljava/lang/Object;)
```

(b) Extending from a class with no resources access

Figure 7.3. Examples of nodes to validate resources access

- **Parent class has no access to resources:**

  Extracted strings will eventually be replaced by a resource identifier, but one caveat of these identifiers is that they reference the content of the strings in the resources; the only classes that can access these resources need to extend from the activity or context classes.

  To check if the parent class of the string extends from either of these classes, the ASTs prove to be useful again, as the root node of the tree (the class definition) contains the information of the class extension. In Fig. 7.3, we can observe two examples of nodes that determine if the resources can be accessed or not.

- **Static parent method:**

  Similarly to the class validation, the methods need to be non-static as the string resources can change values, so the problem comes as the string's parent method not having access to the resources. The validation of this property can also be done by

checking the node with the parent method declaration, as shown in Fig.7.4

```
(I_ACCESS_LIST private static)
```

Figure 7.4. Example of node with method access information

Extracting the Hard-coded Strings

After the list of valid hard-coded strings is defined, said strings need a string resource identifier to replace them in the code. Before doing that, it's necessary to understand how Android manages these identifiers.

This is an example of a resource id: '0x7f**04**0519'. The highlighted characters represent which of the xml resource files (strings, drawables, layouts) the id belongs to; this is assigned randomly every time the app is compiled. The numbers after these characters are unique value for each id in the category.

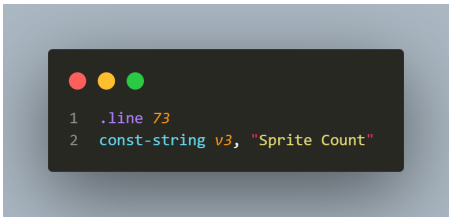
Taking this into account, it's imperative to previously have gotten the value of the characters that the app defined for the string resources; this is done for consistency's sake, as the app won't compile again if the string identifiers don't match. The following code in Fig.7.5 displays the implemenation of the helper method that finds the characters.

Now, having the string id, the assignment of the resource id is pretty straightforward. This is done with the helper method shown in Fig.7.6, The only thing to be careful about is continuing the ascending numeration that the app follows for the ids; this is clearer with an example: if the last string id was '0x7f040010', then the first replaced HCS must be '0x7f040011'.

Finally, with all the information saved from initial string location and with the resource ids, the only thing left is to modify the files, extracting the HCSs and replacing them with the new ids.

```java
public static String getStringIdentifier(String folderPath){
    Collection<File> files = FileUtils.listFiles(new File(folderPath), TrueFileFilter.INSTANCE, TrueFileFilter.INSTANCE);
    String identifier = "";
    Boolean checked = false;
    for (File file : files) {
        //only the file that holds the strings ids has to be checked
        if(file.getName().contains("public.xml")){
            List<String> fileLines = readLines(file.getAbsolutePath());
            for (int i = 0; i < fileLines.size() && !checked; i++){
                String currentLine = fileLines.get(i);
                //The identifier is extracted from the first string reference, so it stops there.
                if(currentLine.contains("type=\"string\"")){
                    String identLine = currentLine.split("id")[1].replace("=\"","");
                    String identValue = identLine.substring(4, 6);
                    if(identValue.startsWith("0")){
                        identifier = identValue.substring(1);
                        checked = true;
                    }
                    else{
                        identifier = identValue;
                        checked = true;
                    }
                }
            }
        }
    }
    return identifier;
```

Figure 7.5. Method to find the id that string resources use in the app



```
.line 73
const-string v3, "Sprite Count"
```

(a) Hard-coded string before extraction

```
.line 73
invoke-virtual {p0}, Lcom/android/spritemethodtest/SpriteMethodTest;->getResources()Landroid/content/res/Resources;

move-result-object v2

const v3, 0x7f040001

invoke-virtual {v2, v3}, Landroid/content/res/Resources;->getString(I)Ljava/lang/String;

move-result-object v3
```

(b) Same HCS content now being accessed through the app string resources

Figure 7.7. Before and after extracting an HCS

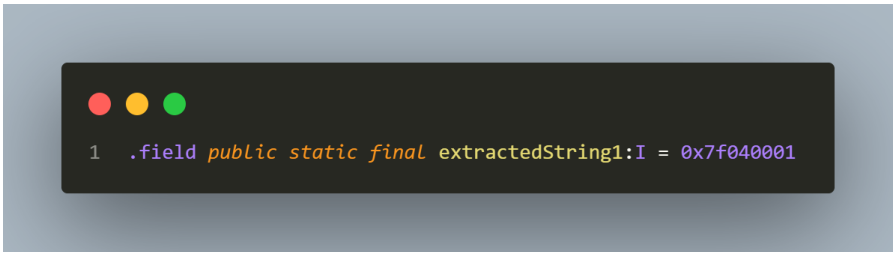
```
private static String setStringId(int index, String identifier){
    String stringId = "";
    //Checks the current number of the index to correctly create the id
    //always starts with 0x7f0 then followed by the identifier(given when created the APK) and then the index
    if(identifier.length()==1){
        identifier = "0"+identifier;
    }
    if( index>=0 && index < 10)
        stringId = "0x7f"+identifier+"000"+index;
    else if(index>= 10 && index < 100)
        stringId = "0x7f"+identifier+"00"+index;
    else if(index>= 100 && index < 1000)
        stringId = "0x7f"+identifier+"0"+index;
    else if(index>= 1000 && index < 10000)
        stringId = "0x7f"+identifier+index;
    return stringId;
}
```

Figure 7.6. Method to assign the resource id to the extracted HCSs

To properly replace the HCSs with the string ids, some added steps need to be done to allow the access to the string resources. An example is presented in Fig.7.7: First the resource of an activity must be called and the result object is saved in a local variable; then, the resource object calls the string file to search for the string linked with the resource id passed as argument (local var v3); finally, the string content associated to that id is saved in the same local var that stored the string when it was hard-coded.

The previous step cannot work properly if the string reference is not added to the resource XML files. This is done by writing in three files; the string id in the public XML file that holds the resources ids of all categories; the string content in the string resource XML file; and the resource id again in the smali string file that links the XML with the code. An example of this can be seen in Fig.7.8.

(a) String added to the string resource file



(b) String resource id added to the smali resource file

(c) String resource id in the public XML file

Figure 7.8. Example of the way the string content is linked to the resource id

Extracting the Layout Hard-coded Strings

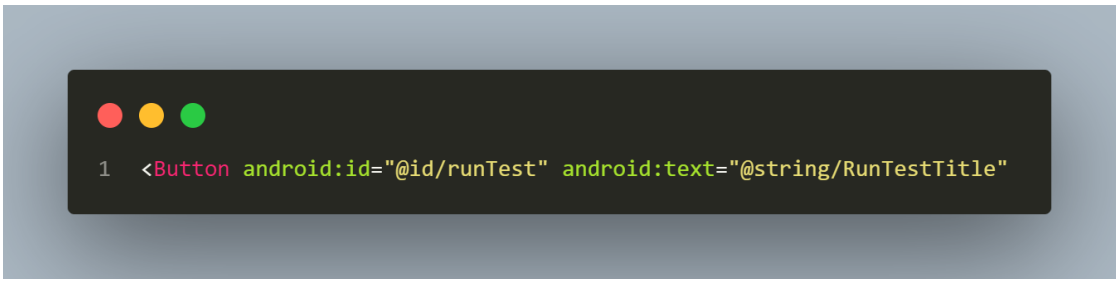
The extraction begins by finding all the HCSs in the layout files, then narrowing the list down by making validations on the content of the strings.

Finding the possible HCSs comes down to reading layout XML files line by line, because, unlike the logic process, the use of an AST to navigate through hierarchically was not possible. Therefore, the algorithm looked for XML layout tags that contained the following attributes:

- android:text
- android:hint
- android:contentDescription

These TextView attributes will display text in the view that they are declared. No matter what the layout object is, they always inherit these attributes from TextView if they display

(a) XML attribute with a hard-coded string



(b) XML attribute with a resource string after the extraction of an HCS

Figure 7.9. Example of the extraction and replacement of a HCS in a layout

any text, so by checking if these attributes for hard-coded text, we are guaranteeing that all the displayed strings in the app won't be hard-coded by the end of the process.

The only thing left before extracting the strings is to validate if the text can be translated, using the same regular expression matcher used in the logic files, as pictured in Fig.7.2.

Finally, the extraction comes to writing in all the files the new string resource id for each HCS; this id is generated by the same method as in Fig.7.6.The result will be as pictured in the example in Fig 7.9

The other files to modify are the public and String XML files by adding the string content and the corresponding resource id, the same way as is done in the logic files.

7.2.3 Right-to-Left Language Support

ITDroid already featured the analysis of support of right-to-left (RTL) languages in Android apps. To improve this further, it's decided to offer a fix for the lack of mirroring for RTL languages in some android apps.

Android RTL Support

Before talking about the implementation, previous research was needed to decide if some existing solution will be used or if the source-less approach would require the development of a new solution.

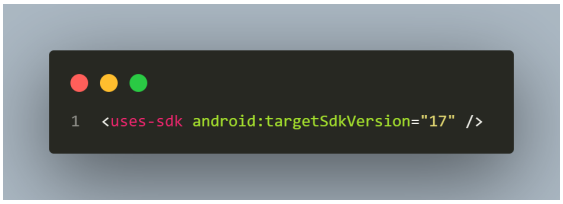


Figure 7.10. Attribute containing API version to check for RTL support

The most reliable solution is the official Android RTL support implementation, as it won't require adding external libraries that could change the app's behavior in ways the original developer didn't intend to. Another benefit of using the Android implementation is that they published a comprehensive guide [173] to implement the necessary changes for the app to support proper mirroring.

Previous Testing. The Android guide was designed for apps that are in the development phase, or for developers to access the source code. For this reason, it was necessary to manually test the changes with the decompiled code before coding the automatic solution. The test has shown that it was possible to add mirroring support to the layout files, but the logic files results were inconsistent and didn't guarantee proper mirroring.

Implementation

The implementation is done by following the Android RTL support documentation [173]. Each step is adapted to be done with the decompiled code since it usually is done manually in the development stage.

Validating the API Version. One big caveat of using the Android support is that the version of the Android API which the app released for matters. If the API version is lesser than 17 (or Android 4.2), then the app will skip the changes made in the implementation.

To check the app support, the 'targetSdkVersion' attribute declares the version in the app's manifest file Fig.7.10. Therefore, it was decided that it would be better to validate the API version before making changes in the code pointlessly if it has no support.

Updating the Layouts

To start, the app needs to declare in the manifest file that the support for RTL languages is enabled, like in Fig.7.11. This is done by searching for the application XML tag and adding the 'supportsRtl' attribute.

Following the documentation, the next step is to update all the XML attributes that use the keywords 'left' and 'right', for 'start' and 'end', respectively; Fig.7.12 shows an example. This is a more logical approach since every language has defined from which side it is supposed to start to be read to make sense. The implementation is similar to the replacement of HCSs in XML files: read line by line, find the keyword, and then replace it with the new word.

These changes are enough for the app to recognize a change in language and adapting the mirroring of all the layouts declared in XML files.

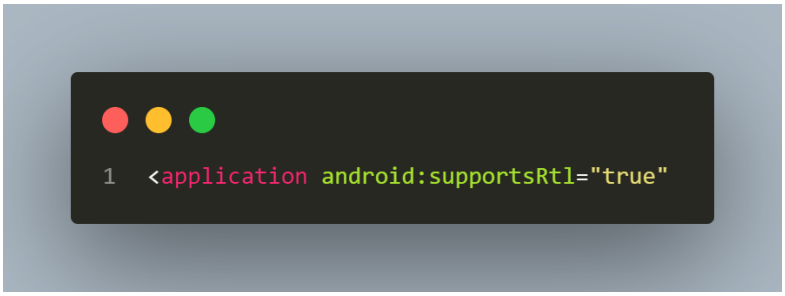


Figure 7.11. Declaring RTL support in the AndroidManifest file

(a) XML attribute referencing left and right

(b) XML attribute now referencing a start and end

Figure 7.12. Example of the change in location attribute tags

7.2.4 Web Report Extension

The aim of this section is to show and explain the components added to the ITDroid web report; these components contain information about the app's modifications to fix the RTL support and HCS issues. The new components use the same libraries as the previous version and follow the existing design system.

Report Information

Hard-coded Strings information In the process of extracting the HCSs it was decided that these would be classified into three categories for the web report:

- Extracted from logic
- Extracted from layout
- Not extracted

The distinction is made so the developers using ITDroid could find the strings easily. For each category, a JSON file was generated with each HCSs and its information is a JSON object. The information for each HCS was gathered during the whole process of extraction; it includes the exact file location, the resource generated (if it was extracted), and the string content.

Right-to left languages support information As the results of the RTL process are easier to understand visually, the information gathered is the screenshots of every layout screen for the original mirroring and the repaired mirroring. ITDroid already saved the screenshots to display them in the report, so it was only necessary to create a JSON file containing the location of the images and the relation between them.

Hard-coded Strings Component

For the changes made regarding HCS extraction, a new component was implemented with a table for each of the categories mentioned in the information section. The table aims to show the user in detail what happened with each HCS, so they can decide by looking into these strings directly in the code. An example of this component is shown in Fig.7.13.

Right-to-Left Languages Support Component

For the result of the whole RTL support process, it was decided that a comparison side by side would be appropriate to showcase the difference that having proper mirroring makes for the RTL language users. This component is a table that takes the screenshots from three versions: the English original, the RTL language without proper mirroring, and the RTL fixed. This can be seen with an example of an app that has Arabic as one of its languages in Fig.7.14

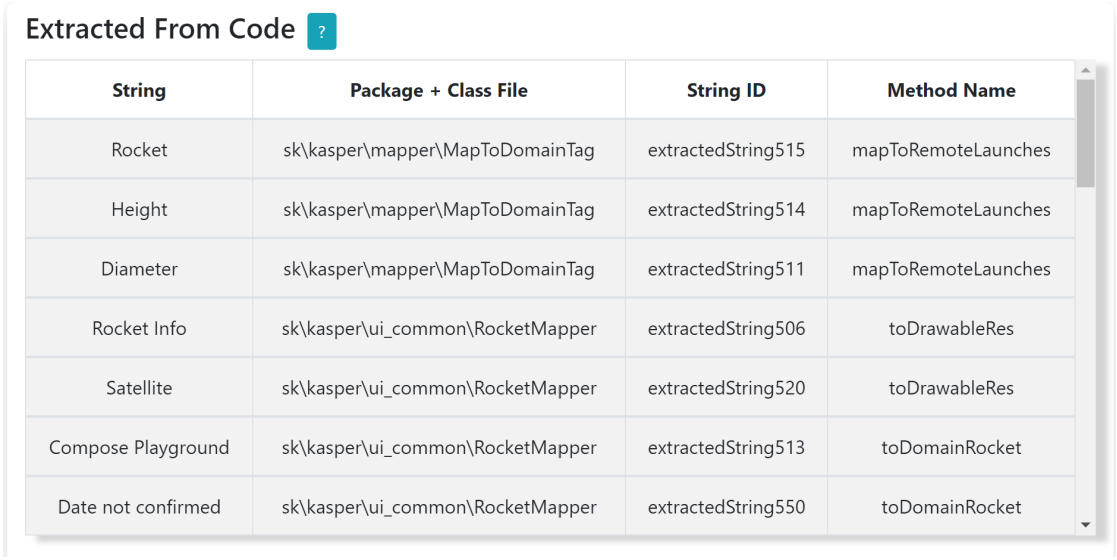
7.3 Results

7.3.1 Comparison Between New and Previous ITDroid Versions

By the end of the implementation process of the new features, we compared the behaviour of our original approach and the new version of ITDroid. In order to achieve this, we executed ITDroid 2.0 with a set of Android app we used for the evaluation of the original tool. The idea was to analyze the results given by both versions and point out the relevant changes.

Firstly , a comparison of the number of HCSs found by both versions was made, since one of the components modified impacted the identification and location of HCSs.

As we can see in Fig.7.15 the results vary between versions, and there's no noticeable pattern. In the instances that the new version found fewer HCS, the explanation can be that the validations in this version filter the strings that should not be translated; When checking the strings both versions found, this checks out. As for the apps with more strings in the new version, it can be attributed to ITDroid reporting and extracting the HCS found in the layouts, something that the previous version did not do.



Extracted From Code ?

String	Package + Class File	String ID	Method Name
Rocket	sk\kasper\mapper\MapToDomainTag	extractedString515	mapToRemoteLaunches
Height	sk\kasper\mapper\MapToDomainTag	extractedString514	mapToRemoteLaunches
Diameter	sk\kasper\mapper\MapToDomainTag	extractedString511	mapToRemoteLaunches
Rocket Info	sk\kasper\ui_common\RocketMapper	extractedString506	toDrawableRes
Satellite	sk\kasper\ui_common\RocketMapper	extractedString520	toDrawableRes
Compose Playground	sk\kasper\ui_common\RocketMapper	extractedString513	toDomainRocket
Date not confirmed	sk\kasper\ui_common\RocketMapper	extractedString550	toDomainRocket

(a) Table with the information about extracted HCS from the logic

Extracted From Layout ?

String	Layout File	String ID
Type	activity_main	extractedString603
Probe	activity_main	extractedString615
Watch Live	activity_main	extractedString605
show before this time	settings	extractedString609
Unconfirmed Launches	settings	extractedString612

(b) Table with the information about extracted HCS from the layout

Not Extracted ?

String	Package + Class File
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp
Intent.ACTION_VIEW	sk\kasper\base\SpaceApp

(c) Table with the information about strings that were not extracted

Figure 7.13. Component for the results of the extraction of HCSs

Right-to-left Languages Repairment Results ?

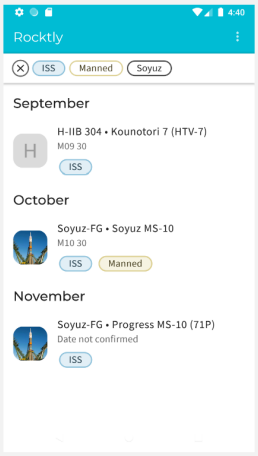
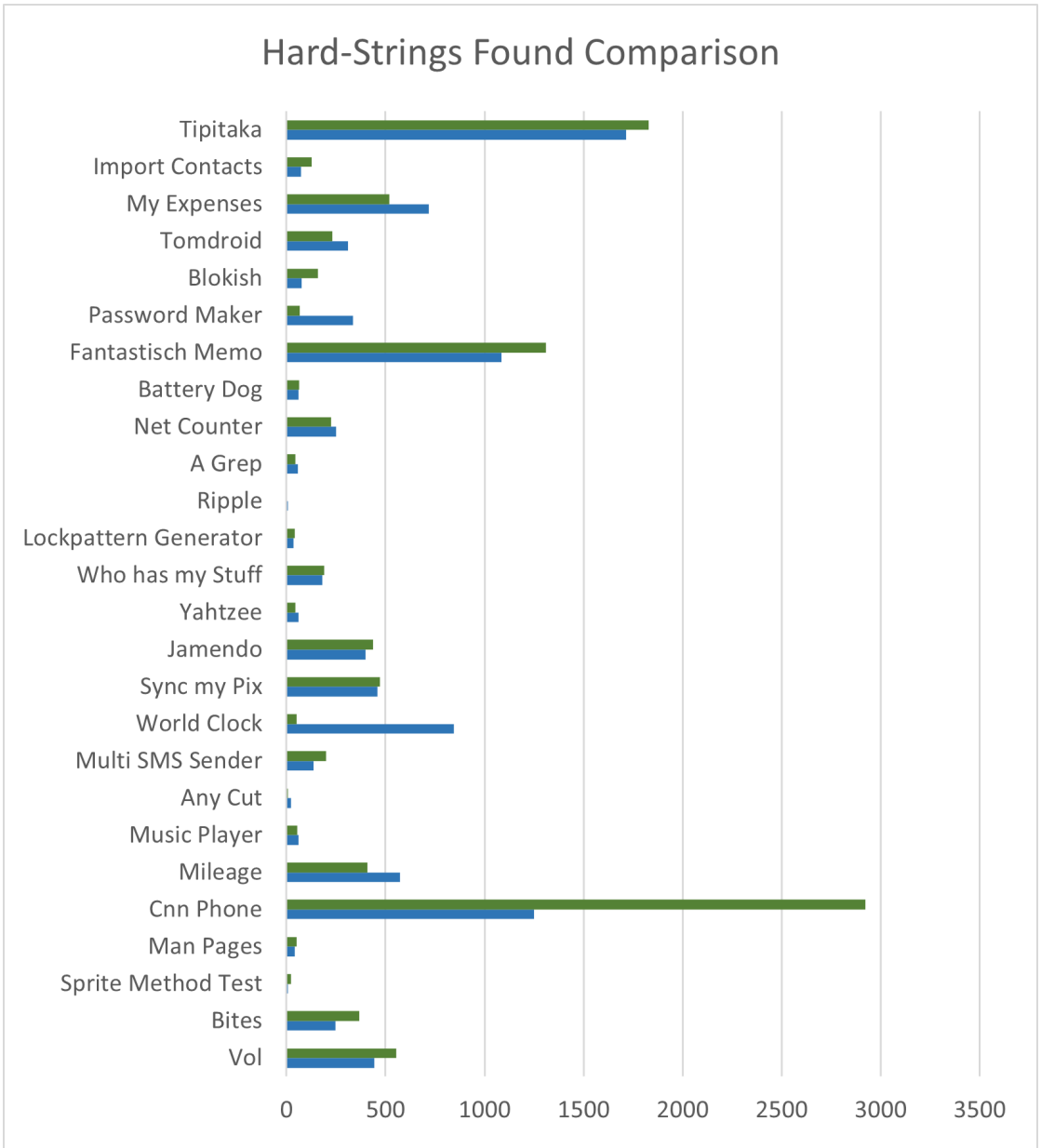
State	English Version	Arabic Original Version	Arabic Repaired Version
1			

Figure 7.14. Table for the comparison of the RTL support results



7.3.2 Functionality of the Apps with the New Version

As previously discussed in the implementation sections about problems that can arise if the wrong strings are extracted. By the moment the test was being carried out, it was noticeable that the apps were failing in some functionalities, or they were straight out crashing; this only happened when testing languages other than the default.

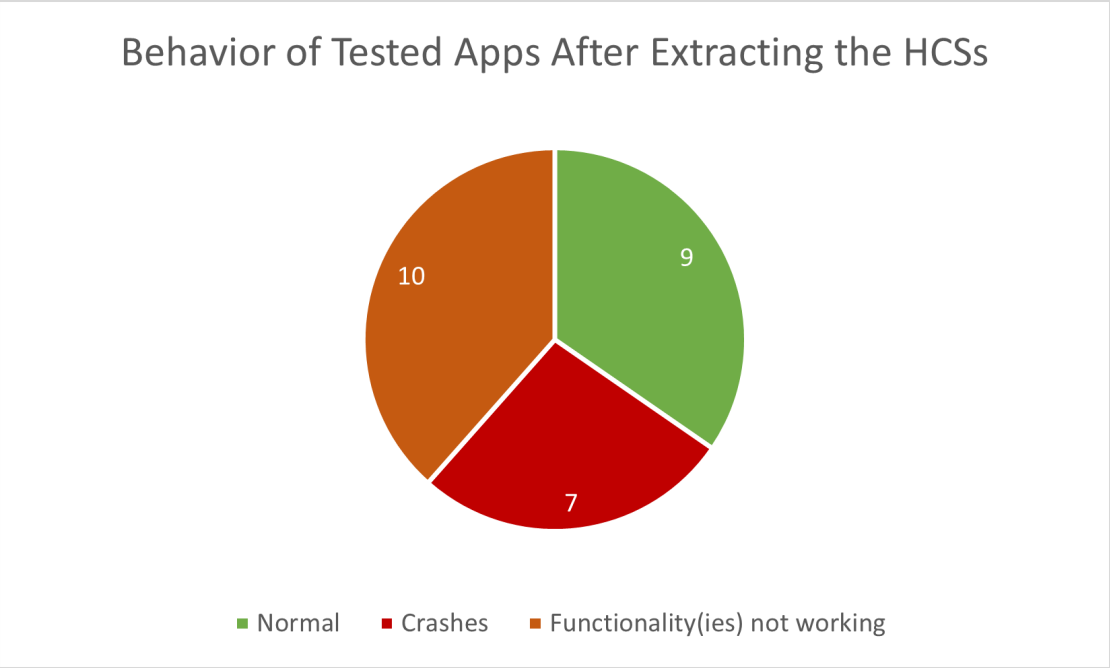


Figure 7.16. Behavior of all the apps tested in the new version

In Fig.7.16 we can see the results of the functionalities in the tested apps when using a different language than the original. By following the error trace in all the apps with malfunctions, it became clear that the problem came from translated words used in the logic of the program, so in every non-default language, the app gets unexpected strings.

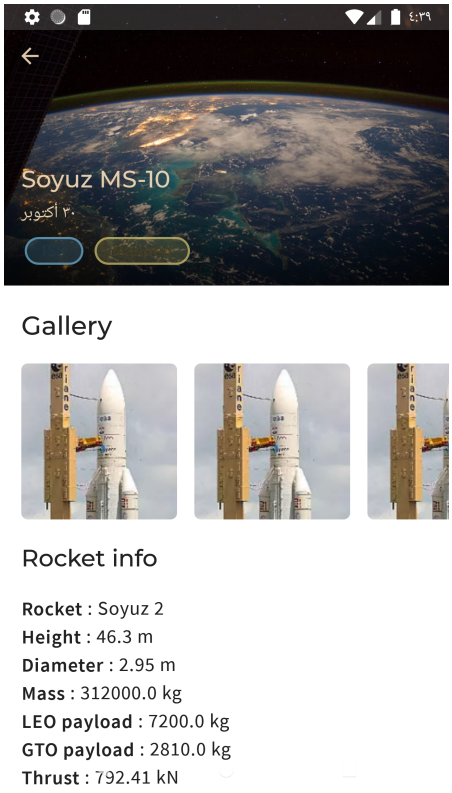
7.3.3 Results of Right-to-Left improved mirroring

For this section, the tested apps changed, as most of the previous apps were developed for older Android versions with no RTL support available. Instead, newer open-source Kotlin apps were tested.

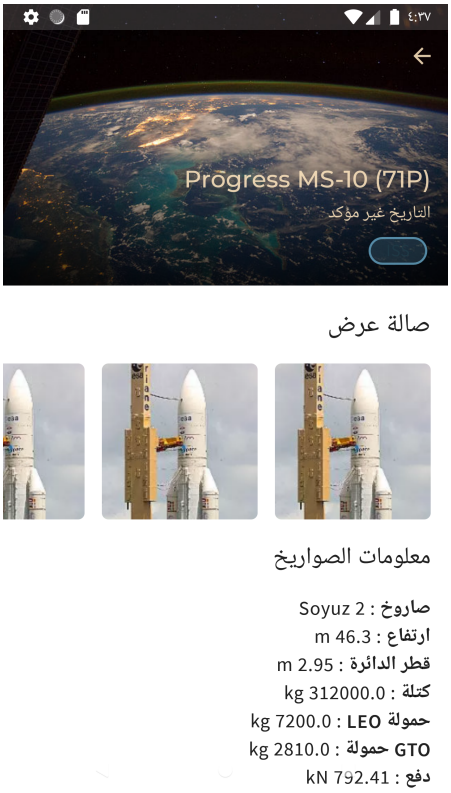
This test was straightforward as it only required running ITDroid and comparing the results of the original RTL language version and the improved version. As we can see in the examples in Fig.7.17, the mirroring improvement was done correctly, and now the app follows the implementation standard of Android RTL support. The result was the same with the other Kotlin apps, so no real issue was found with this feature.

(a) First screen of tested app with no mirroring

(b) First screen of tested app with the correct mirroring



(c) Second screen of tested app with no mirroring



(d) Second screen of tested app with the correct mirroring

Figure 7.17. Comparison of the mirroring in a Kotlin tested App

8

Internationalization of Android Apps: Discussion and Summary

Although `ITDroid` was not designed to automatically translate strings within apps, since the translation process is implemented with a strategy pattern, any user could implement its own `ITDroid` that works with another translation service. We used the IBM Watson engine, but any translation engine can be easily plugged in.

Despite the existence of mechanisms for supporting mobile app internationalization, issues such as collateral changes, bad practices, and collateral bugs were found in the analyzed apps. Therefore, existing practices should be promoted more in the practitioner's community, and researchers should envision and implement tools for the automated detection of *i18n* bugs in the case of issues induced by bidirectionality and text expansions.

Following this, we implemented a first approach to fix i18n issues automatically, opening a set of research paths to extend ITDroid to support a comprehensive set of validations and improve the identification, location, and fixing of i18n issues. The results presented in these chapters provide insights to developers on how to avoid i18n issues and additionally provide them with a tool that can be easily added to their current quality assurance protocol to reduce the number of i18n issues present in their apps.

Moreover, implementing these features on an automatic tool provides mechanisms to exploit the generated models, such as Layout Graph (LG) and the enhanced AST, to create new tools to improve the development process. For example, the relations shown in the LG can be used in visual regression testing to study the behavior of the UI when using different devices. Therefore, using the Layout Graph might help practitioners recognize issues present due to the fragmentation of the Android environment.

Based on the extension done in Chapter 7, we identified that it is possible to further improve ITDroid, by offering repairments to the issues that the tool was already reporting; this provides practitioners with newly available capabilities to enhance the experience of the GUI when internationalizing their applications.

Part IV

Connectivity Management for Android Apps

Structure of Part IV: Connectivity Management for Android Apps

- First section outlines the main concepts and previous work related to Connectivity Management for Android Apps.
- **Chapter 9** presents our empirical study on manual identification of connectivity issues present on Android apps.
- **Chapter 10** depicts our efforts towards generating a tool capable of automatically identify connectivity issues. Additionally, presents our study validating the capabilities of the aforementioned tool.
- **Chapter 11** presents a summary of findings, contributions and open research opportunities created as result of our work regarding Connectivity Management for Android Apps.

There are no limitations regarding the locations or conditions in which mobile devices are used, and *offline-first* practice is a highly desired quality of mobile apps. However, despite high-speed access to the internet is being more and more common worldwide and "data plans" are more accessible/cheap in terms of costs, there are still complex connectivity scenarios, such as locations with zero/unreliable connection. Therefore, inappropriate handling of connectivity issues may lead to bugs and crashes that negatively affect the user experience when an app is used under *Eventual Connectivity (ECn)* scenarios.

Offline first practices – i.e., programming practices that allow an app to provide network-enabled features without internet access – have been initially promoted for web applications (in particular progressive web apps), and are motivated by the need for providing a better user experience even when there is no connectivity [174, 175]. Several specific implementation practices are well known in the web development community, such as using app shells, local caching in the browser, bundle network requests when users are offline, and connectivity awareness. These practices have also been transferred to the mobile app domain and complemented with app-specific "guidelines/practices" proposed by practitioners [176] and Google developers [177]. An example of those guidelines is offline data-synchronization with backend services via APIs (*e.g.,* local caching enabled by Firebase).

Network permission is the most popular permission among Android apps [178, 179] which suggest that Android apps are highly dependent on internet access. This has pushed the research community to study network-related aspects of mobile apps, and in particular: (i) network traffic profiling/characterization/usage [179, 180, 181, 182, 183, 184, 185, 186, 187, 188]; (ii) security and privacy concerns, namely how private information is being manipulated and protected in apps [189, 190, 191, 192, 193, 194, 195]; and (iii) network-related vulnerable scenarios which can be exploited by mischievous attackers [196, 197, 198]. To the best of our knowledge, there is no empirical study analyzing bugs/crashes in Android apps that are caused by connectivity issues. As a consequence, it is unknown the prevalence of those issues and their impact on the quality as perceived by users.

Acknowledgments

The research presented in these chapters was conducted in collaboration with Ph.D. Alejandro Mazuera-Rozo from Universita della Svizzera italiana. The results presented in here are also part of the final document presented for his dissertation.

9

Identification of Connectivity Issues on Android Apps

Mobile apps have become indispensable for daily life, not only for individuals but also for companies/organizations that offer their services digitally. Inherited by the mobility of devices, there are no limitations regarding the locations or conditions in which apps are being used. For example, apps can be used where no internet connection is available. Therefore, *offline-first* is a highly desired quality of mobile apps. Accordingly, inappropriate handling of connectivity issues and miss-implementation of good practices lead to bugs and crashes occurrences that reduce the confidence of users on the apps' quality. In this chapter, we present the first study on *Eventual Connectivity* (ECn) issues exhibited by Android apps, by manually inspecting 971 scenarios related to 50 open-source apps. We found 304 instances of ECn issues (6 issues per app, on average) that we organized in a taxonomy of 10 categories. We found that the majority of ECn issues are related to the use of messages not providing correct information to the user about the connectivity status and to the improper use of external libraries/apps to which the check of the connectivity status is delegated. Based on our findings, we distill a list of lessons learned for both practitioners and researchers, indicating directions for future work.

Structure of the Chapter

- Section 9.1 provides motivation for this chapter.
- Section 9.2 presents the design of our study, as well as the data extraction procedure and analysis methodology.
- Section 9.3 discusses our results and findings.
- Section 9.4 presents the threats that affect the validity of our work.
- Section 9.5 draws our conclusions.

Supplementary Material

All the data used in this chapter are publicly available. More specifically, we provide the following items:

Online Appendix.[199] The online appendix includes the following material:

- **Taxonomy**. Taxonomy of eventual connectivity issues and descriptive informationconcerning each issue.
- **Examples**. Interactive examples showing multimedia content regarding someeventual connectivity issues existing in real-world open-source Android applications.

Publications and Contributions

Studying eventual connectivity issues in Android apps[16]
Escobar-Velásquez, C., Mazuera-Rozo, A., Bedoya, C., Osorio-Riaño, M., Linares-Vásquez, M., and Bavota, G. In *Empirical Software Engineering, 2022.*

Taxonomy of *i18n* collateral changes and bugs exhibited on the 31 analyzed Android apps. → Fig. 6.4
We present and discuss a taxonomy of changes and bugs we built based on the results obtained as part of the study conducted in the previous article.

9.1 Introduction

Mobile apps have become an indispensable tool for daily activities. There are no limitations regarding the locations or conditions in which mobile devices are used, and *offline-first* practice is a highly desired quality of mobile apps. However, despite high-speed access to the internet is more and more common worldwide and "data plans" are more accessible/cheap in terms of costs, there are still complex connectivity scenarios, such as locations with zero/unreliable connection. Therefore, inappropriate handling of connectivity issues may lead to bugs and crashes that negatively affect the user experience when an app is used under *Eventual Connectivity (ECn)* scenarios.

Offline first practices – i.e., programming practices that allow an app to provide network-enabled features without internet access – have been initially promoted for web applications (in particular progressive web apps), and are motivated by the need for providing a better user experience even when there is no connectivity [174, 175]. Several specific implementation practices are well known in the web development community, such as using app shells, local caching in the browser, bundle network requests when users are offline, and connectivity awareness. These practices have also been transferred to the mobile app domain and complemented with app-specific "guidelines/practices" proposed by practitioners [176] and Google developers [177]. An example of those guidelines is offline data-synchronization with backend services via APIs (*e.g.,* local caching enabled by Firebase).

As of today, there is no empirical study analyzing bugs/crashes in Android apps that are caused by connectivity issues. As a consequence, it is unknown the prevalence of those issues and their impact on the quality as perceived by users. Indeed, state-of-the-art studies are mostly devoted to (i) network traffic characterization [180, 181], (ii) its security and privacy implications [189, 190], and (iii) possible exploitable scenarios addressed by malicious agents [196]. Previous studies have indeed focused on cataloging bugs/crashes for mobile apps in general [200], and bugs/crashes specific to quality attributes such as performance [153, 201, 202, 203, 204], security [205, 206, 207], behavioral/GUI inconsistency [208, 209, 210, 211], and energy consumption [212, 213, 214, 215, 216].

Moreover, there is no tool available for detecting this type of bugs statically or dynamically. The closest available approaches/tools for automated detection of crashes related to the lack of connectivity are CrashScope [160, 217], Thor [218], and Caiipa [219]. Those approaches systematically explore an app, generate events like turning-off WiFi, and look for crashes (*i.e.,* the application stops). However, as we will show in this work, crashes caused by a lack of connectivity only represent a subset of the connectivity issues that affect mobile apps.

Despite the lack of literature strictly related to connectivity issues, the latter are prevalent in Android apps. Indeed, as a preliminary step towards the study we present in this work, we checked whether connectivity-related issues were discussed in the issue trackers of open source Android apps hosted on GitHub. By mining issues from 3,256 apps used in previous work [220, 221] we collected ∼219k issues of which 11,350 — belonging to 943 apps — matched in their title or description with the keywords *offline*, or *connectivity*, that we used as a mechanism to identify the presence of connectivity-related issues. Know-

ing that false positives are likely to be retrieved in this way, the first two authors manually inspected 400 instances each to verify whether they were true positives (i.e., reports actually related to connectivity-issues) or false positives; 400 instances ensure a significance interval of ±5% with a confidence level of 95%. After solving 81 conflicts through an open discussion, they agreed on 213 true positives (53.2%). Being conservative, and assuming a level of precision of 50% for the employed keyword-based heuristic, this suggests that ∼5.6k connectivity-related issues could be present in the issue trackers of the mined apps. More detailed information regarding this analysis can be found on our online appendix [199].

In this work we present the very first study on *Eventual Connectivity (ECn)* issues exhibited by Android apps *in-the-wild*. Our study aims at building the empirical foundations needed to (i) increase practitioners' awareness about the prevalence of these issues and how they manifest in Android apps; and (ii) build techniques and tools aimed at automatically detecting the *ECn* issues we document. Compared to other studies (focused on other type of bugs) that followed a mining-based strategy over code repositories and online markets, we preferred an inspection-based approach to (i) avoid any of the imprecisions and limitations of analyzing user reviews [222, 223, 224, 225], and (ii) have detailed information of the conditions that triggered the bugs in the apps. In particular, we manually executed and inspected 50 open source Android apps and we build a catalog of bad practices/issues that are exhibited by those apps in ECn issues. The execution is based on a total of 971 scenarios we designed for the 50 apps; on average, each scenario has 3.5 steps (i.e., interactions with the app). We found 320 instances of *ECn* issues that we grouped into a taxonomy of 10 categories. In addition, we contribute an online appendix [199] that describes the cause of the identified issues with videos, execution steps and code snippets. Our results and online resources can be used by researchers and practitioners to (i) create approaches for the automated detection of these issues, (ii) being aware of testing cases that should be included into the quality assurance processes of mobile apps, and, in general, (iii) avoid the issues reported in our taxonomy.

9.2 Study Design

Despite mobile apps widely rely on connections to back-end services/resources via WiFi or cell network, there is no previous work that analyzes the practices followed by Android developers. Therefore, we carried out an empirical study in which we manually analyzed 50 open source Android apps to build a taxonomy and an online catalogue of the most common issues/bad practices in Android apps that are related to eventual connectivity. We relied on open source apps for the analysis because we were interested in the coding practices followed by developers, thus requiring access to the code. The analysis is based on the execution of the apps and the manual localization of the issues in the code. For the 50 apps, we analyzed 971 scenarios, 320 of which allowed to spot issues related to eventual connectivity. The tagging process for building the taxonomy was performed by three authors, who categorized each spotted issue and organized them into a taxonomy. The generated taxonomy was revised by the remaining authors. That taxonomy was the foundation for answering the following question:

RQ_1: What are the eventual connectivity issues observed in Android open-source apps?

With this research question, our goal is to find and classify eventual connectivity issues in a group of open-source Android apps. Also, we intend to understand how these issues affect user-experience and what are the issues in the code causing them. To this aim, we (i) test the apps directly on Android devices, and (ii) check the source code repositories. Additionally, to ease the discussion of the ECn bugs, we also link them to criteria described in the ISO/IEC 25010 quality model, indicating the quality attributes on which each type of bug has an impact. The answer to this question is expected to help developers and researchers to prevent eventual connectivity issues in apps and to develop new tools for automatic identification of ECn issues.

In the rest of this section, we describe the open-source mobile apps used for this study and how we selected them. Then, we present the manual analysis we followed to identify/classify the eventual connectivity issues.

9.2.1 Context Selection

To answer RQ_1, we targeted the selection of 50 popular open-source Android apps with features relying on Internet connection. The limit to a maximum of 50 apps was defined due to the expensive manual process adopted in our study to test the apps (details follow). The apps were manually selected from publicly available lists of open source Android apps: (i) the Wikipedia list, featuring 90 free/open source Android apps [226], and (ii) a GitHub repository listing over 300 of open source Android apps [227]. We looked for apps meeting the following criteria:

1. *To be a native Android application (built in Java or Kotlin):* we focused our study on native Android apps excluding hybrid apps, since they may be subject to different types of connectivity issues and we wanted to keep our analysis cohesive. Three authors manually inspected the code repositories of the apps by checking their manifest, the programming language and the code in the project to ensure that the selected apps were not hybrid.
2. *To be an open-source app available on Google Play:* for this study, we needed access to the source code of the evaluated applications, since we were interested in analyzing code snippets exhibiting issues. In addition, the app should be publicly available on Google Play to ensure that the analyzed apps are neither a library nor a class/toy project.
3. *To be a popular app:* to guarantee we are studying eventual connectivity behaviors on real apps used by a large amount of people, we focused on popular apps in the market. We consider an application to be popular if (i) it has at least 1,000 downloads and (ii) its average rating is above 3.0. Fig. 9.1 depicts the number of apps per downloads range. The download ranges are defined by the Google Play website. The number of downloads is reported as a single number (*e.g.,* 500) when it is lower than 1,000, otherwise, the number is reported as a range *e.g.,* 10,000 - 50,000. Fig. 9.2 depicts the distribution of average rating for the 50 apps.

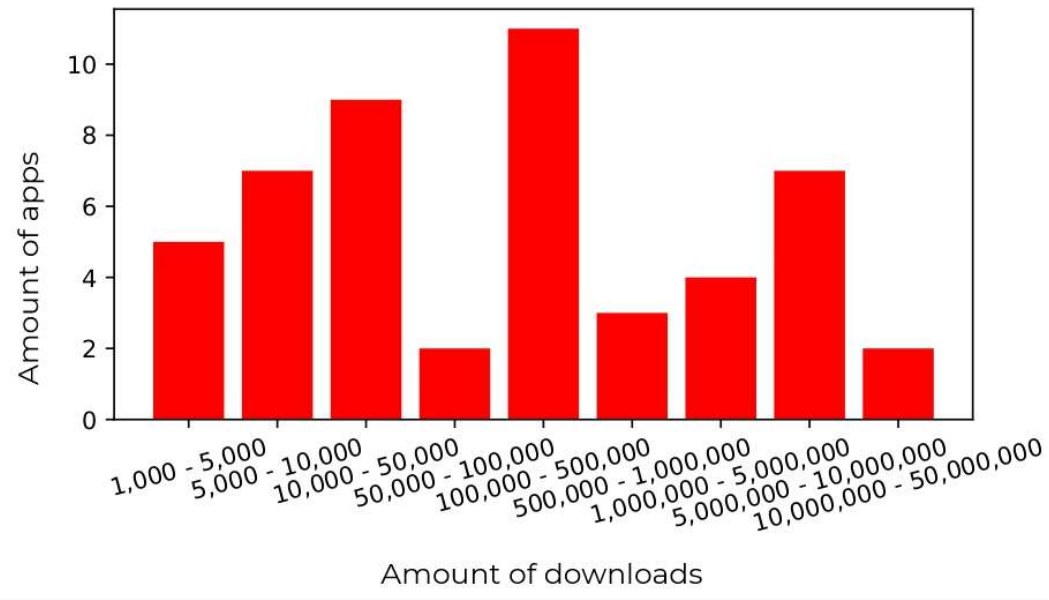


Figure 9.1. Frequency of apps (histogram) per downloads range.

4. *Covering a diverse set of categories on the Google Play store:* We selected apps covering 20 different categories from the Google Play store, as depicted in Fig. 9.3.
5. *To have functionalities relying on or network connection:* as we wanted to study issues related to eventual connectivity, we needed our selected apps to have features relying on Internet or network connections. We manually checked this criterion by looking for network-related features in the apps' description (e.g., README file, shared screenshots) and permissions definition.

The list of 50 selected apps is presented in Table 9.1. Our rationale for limiting the study to 50 applications is based on the amount of effort required to manually test the apps (details about the testing process are reported in the following). The column "Date" corresponds to the date in which the information and APKs were downloaded. The "Downloads interval", "Average Rating", "Version", and the "Category" information of each app was collected from the Google Play Store. The values presented in the column "Version" refer to the latest version of the artifacts available on the market when we collected the information. As we can see, the set of apps we selected cover a total of 20 different categories, ensuring some diversity of the selected apps. Still, we do not claim that our dataset is representative of all native Android apps, since this would require to run our analyses on a much larger number of apps.

Table 9.1. Analyzed applications.

ID	Application	Category	Downloads	Average rating	Version	Revision date
1	QKSMS	Communication	100,000 - 500,000	4.1	v2.7.3	13/07/2017
2	Wire	Communication	1,000,000 - 5,000,000	4.1	2.38.352	13/08/2017
3	Surespot	Social	100,000 - 500,000	4.2	70	23/08/2017
4	Galaxy Zoo	Education	5,000 - 10,000	4.6	1.69	02/08/2017
5	Fanfiction reader	Books and reference books	100,000 - 500,000	4.3	1.51	20/08/2017
6	PressureNet	Weather	100,000 - 500,000	4	5.1.10	02/08/2017
7	AnntenaPod	Video Players and Editors	100,000 - 500,000	4.6	1.6.2.3	02/08/2017
8	iFixit	Books and reference books	1,000,000 - 5,000,000	4.3	2.9.2	03/08/2017
9	DuckDuckGo	Books and reference books	1,000,000 - 5,000,000	4.4	3.1.1(101)	20/08/2017
10	OsmAnd	Maps and navigation	5,000,000 - 10,000,000	4.2	2.7.5	20/08/2017
11	Mozilla Stumbler	Tools	50,000 - 100,000	4.5	1.8.5	20/08/2017
12	XOWA	Books and reference books	1,000 - 5,000	3.9	2.1.173-r-2017-06-25	21/08/2017
13	Journal with Narrate	Lifestyle	50,000 - 100,000	4.2	2.4.0	08/09/2017
14	Wikimedia Commons	Photography	10,000 - 50,000	4.3	2.4.2	21/08/2017
15	WordPress	Social	5,000,000 - 10,000,000	4.2	8.0	24/08/2017
16	GnuCash	Finance	100,000 - 500,000	4.4	2.2.0	21/08/2017
17	My Expenses	Finance	500,000 - 1,000,000	4.4	2.7.9	24/08/2017
18	FBReader	Books and reference books	10,000,000 - 50,000,000	4.5	2.8.2	24/08/2017
19	K-9 Mail	Communication	5,000,000 - 10,000,000	4.2	5.207	24/08/2017
20	MAPS.ME	Travel & Local	10,000,000 - 50,000,000	4.5	7.4.5-Google	24/08/2017
21	Omni Notes	Productivity	100,000 - 500,000	4.4	5.3.2	24/08/2017
22	Hubble Gallery	Education	50,000 - 100,000	4.6	1.5.1	08/09/2017
23	Prey	Tools	1,000,000 - 5,000,000	4.2	1.7.7	24/08/2017
24	Forecastie	Weather	5,000 - 10,000	4.3	1.2	24/08/2017
25	Twidere for Twitter	Social	100,000 - 500,000	4.1	3.6.24	24/08/2017
26	Opengur	Entertainment	50,000 - 100,000	4.4	4.7.1	24/08/2017
27	AnkiDroid	Education	1,000,000 - 5,000,000	4.5	2.8.2	31/08/2017
28	Transportr	Maps and navigation	5,000 - 10,000	4.6	1.1.8	02/10/2017
29	Ouroboros	Communication	10,000 - 50,000	3.5	0.10.5.1	01/10/2017
30	EarthViewer Beta	Personalization	10,000 - 50,000	4.4	0.6.1-BETA	18/09/2017
31	Open Weather	Weather	1,000 - 5,000	3.5	4.4	28/09/2017
32	Cannonbal	Game - Word	1,000 - 5,000	4.4	1.0.2	10/09/2017
33	OpenBikeSharing	Maps and navigation	1,000 - 5,000	4.6	1.10.0	10/09/2017
34	c:geo	Entertainment	1,000,000 - 5,000,000	4.4	2017.08.23	10/09/2017
35	PAT Track	Maps and navigation	10,000 - 50,000	4.1	7.0.6	10/09/2017
36	Stepik	Education	50,000 - 100,000	4.8	1.42	18/09/2017
37	RunnerUp	Health & Fitness	10,000 - 50,000	4	1.2	10/09/2017
38	Wake You in Music	Tools	10,000 - 50,000	3.6	1.1.1	19/09/2017
39	Habitica: Gamify your Tasks	Productivity	500,000 - 1,000,000	4.3	1.1.6	19/09/2017
40	Openshop.io 1.0	Bussiness	1,000 - 5,000	4	1.2	19/09/2017
41	Glucosio	Medicine	10,000 - 50,000	4.2	1.4.0	29/09/2017
42	Signal Private Messenger	Communication	5,000,000 - 10,000,000	4.6	4.11.5	02/11/2017
43	Tasks: Astrid To-Do List Clone	Productivity	100,000 - 500,000	4.4	4.9.14	02/10/2017
44	Materialistic - Hacker News	News & Magazines	50,000 - 100,000	4.8	3.1	26/09/2017
45	Kontalk Messenger	Communication	10,000 - 50,000	4.3	4.1.0	02/11/2017
46	Open Food Facts	Health & Fitness	100,000 - 500,000	4	0.7.4	28/09/2017
47	RedReader	News & Magazines	50,000 - 100,000	4.6	1.9.8.2.1	10/10/2017
48	Tram Hunter	Travel & Local	100,000 - 500,000	4.6	1.7	09/10/2017
49	PocketHub for GitHub	Productivity	10,000 - 50,000	3.4	0.3.1	09/10/2017
50	Kickstarter	Social	1,000,000 - 5,000,000	4.5	1.6.1	28/09/2017

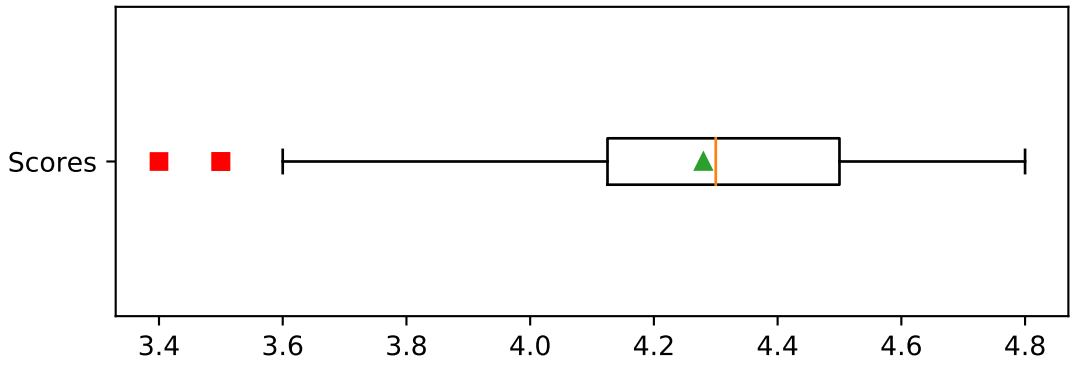


Figure 9.2. Average rating distribution (box plot) of the 50 analyzed Android open source apps. The mean is reported as a triangle and the outliers as squares.

Table 9.2. Test

ID	Application	Category	Downloads	Average rating	Version	Revision date
1	QKSMS	Communication	100,000 - 500,000	4.1	v2.7.3	13/07/2017

9.2.2 Data Collection

The testing and analysis process of the selected apps was organized in three-steps: (i) scenarios design, (ii) scenarios execution, and (iii) taxonomy building. The process was carried out by three authors with experience in mobile apps development (hereinafter referred as “analysts") and took about six months of work with partial dedication. Details about each of the three steps in the data collection are reported in the following.

Scenarios design

Each application was assigned to one of the three analysts, who had as first task to explore the application and the features it provides. Then, the analyst had to fill-in a template composed by two parts:

1. *App information:* Which contains for the given app (i) its purpose, (ii) its category, (iii) the link to its Google Play page, and (iv) the link to its source code repository.

2. *Scenarios:* The set of scenarios to be executed on the app during the testing. We define a scenario as *a specific feature to be executed in the app in a specific context*. With “context”, we refer to the connectivity conditions of the app (*e.g., execute the feature when the WiFi connection is off*, *turn off the connection after starting the execution of the feature*, *etc.*). In summary, a scenario includes the description of the feature to be executed and the steps to follow to reproduce the scenario in the desired context (*e.g.,* which connection to turn on/off at a certain moment).

There are six different types of scenarios the analysts followed for defining the execution steps; the scenarios were identified by letters: *a, b, c, d, e, f*. Four of them, depicted in

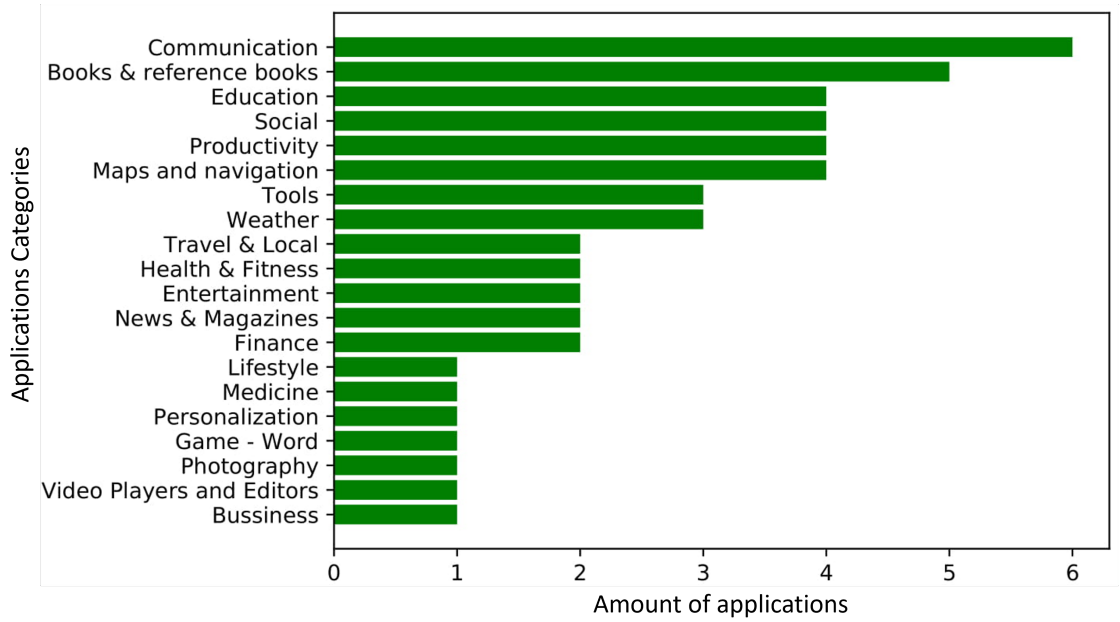


Figure 9.3. Distribution of number of apps per category.

Fig. 9.4, were defined for each app. In Fig. 9.4, the blue line in each scenario represents time and the vertical red line represents the time of an event which can be (i) the execution of the feature, and (ii) the evaluation of the app's behavior. Finally, a vertical dashed line represents a transition between connectivity states (*e.g.,* from *connection on* to *connection off*).

Scenario *a* evaluates a feature using Internet or network connection (connection state) ON. In scenario *b*, the feature was tested with the connection state OFF (*i.e.,* Airplane mode on). In scenario *c* the goal was to execute the feature while the connection is OFF, and then turning the connection ON after executing the feature. Finally, the scenario *d* evaluates if the app has a normal behavior when executing the feature right after turning ON the connection. These scenarios allow to test cases in which the features that rely on back-end services include the implementation of local caches (in the devices), local queues for later dispatching of messages when the connection is recovered, *etc.*

There are two additional scenarios (*e* and *f*) that were only defined for specific apps that can support them. Scenario *e* aims at testing features related to SMS sending. For this action, network connection is usually required. In scenario *e*, we trigger the sending process with Internet connection (*i.e.,* WiFi) but without phone network connection to test the behavior of the application. For example, the Glucosio app (*v1.4.0*) allows users to invite friends via SMS message. When a user tries to send the invitation without both Network and Internet connection an error message reports "*Message failed to send*". If a user has Internet connection but no Network connection, the application shows a success message saying "*Your invitation has been sent*". However, the message is not delivered.

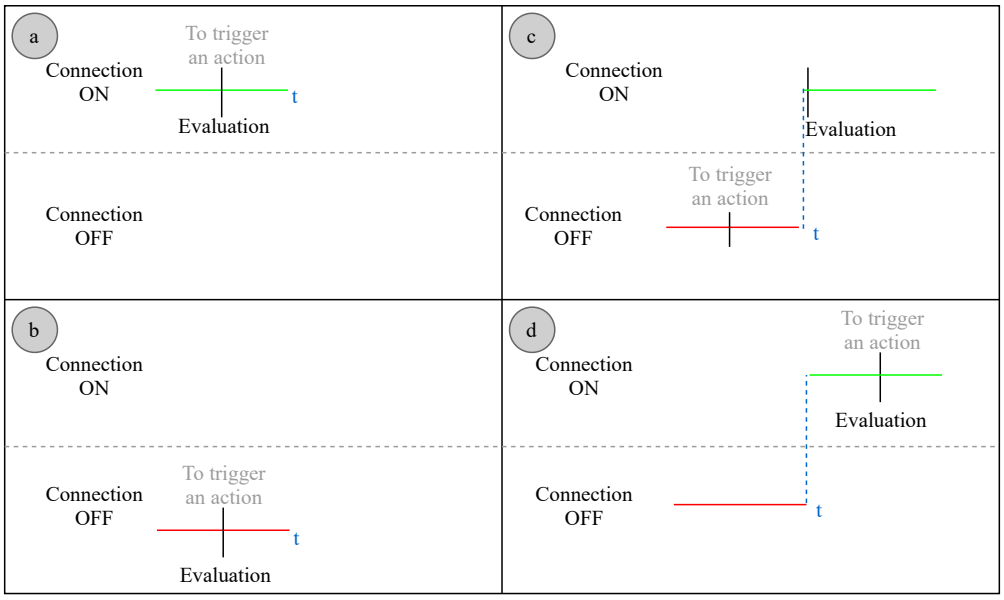


Figure 9.4. Types of scenarios for testing eventual connectivity.

Scenario *f* covers app-specific cases that were defined by the analysts. For example, in Hubble Gallery app (*v1.5.1*), when a user tries to see the details of an image the application request a large amount of data. Therefore, one of the *f* scenarios for this app consists of removing the available connection while the download process is in progress.

A total of 971 scenarios were defined for the 50 apps. Note that for one app we can have multiple *x*-type scenarios (where *x* is one of the six types of scenario we described) involving different features. The minimum number of steps found in a scenario is one, for the apps that present a network-dependent activity when launched (*e.g.,* maps, news feeds), and a maximum of eight steps. These numbers do not take into account the steps required to set the network state to be tested (*e.g.,* turn on airplane mode).

Scenarios execution

Once the scenarios for an app were designed, an analyst executed each of them at least twice by running the app on a physical device. This re-execution process was performed to avoid issues related to inconsistent behaviours. All the executions were made under the same WiFi connection, and the offline cases were obtained by activating the airplane mode. The mobile devices used for this step were a Samsung S6, an Asus Zenphone 2, and a Motorola Moto G, all equipped with the Android API level 25. During the execution, each analyst collected (i) the specific steps followed to execute a scenario (including screenshots of the app and a video of the execution), (ii) the results obtained when executing the scenario, and (iii) the issues exhibited in the app.

Regarding the issues, those were documented by assigning them a description, a name, and a tag. For instance, the “Redirection to a different application without connectivity

check" tag was assigned to the cases in which the app does not check the availability of a connection before redirecting the user to a different view or app that requires the connection. This lack of verification derives in views or apps displaying errors (or displaying nothing) because of the assumption of an existent connection. Every-time a new tag was assigned, the other analysts were notified to verify whether the new tag also applied to apps already analyzed. Note that multiple tags could be assigned to the same issue.

9.2.3 Taxonomy building

After executing all the scenarios, we analyzed the assigned tags to build a taxonomy categorizing the issues exhibited in the apps. The taxonomy was defined by the three analysts through multiple open meetings. Once the taxonomy was built, for each scenario exhibiting an issue we analyzed the source code of the corresponding apps to identify the parts involved in the issue and the performed implementation choices. Using this information, we derive a set of lessons learned describing good practices that should be followed by developers to avoid eventual connectivity issues. All the created scenarios, the documents resulting from their execution, and an extensive list of examples for the detected issues are available in our online appendix [199].

9.2.4 Impacted Factors

As it was mentioned previously, we enhanced our results by analyzing the impact of the identified issues over different quality attributes. In order to do this, two authors used the list of quality attributes depicted in the ISO/IEC 25010 standard to evaluate separately the impact of each ECn category using a 4 values scale (*i.e.,* High, Medium, Low, None). Once both authors finished this step, they removed the quality attributes that where not impacted by any ECn category and compared the assigned values for the remaining ones. Additionally, they solved the conflicts by presenting examples of issues that support the assigned value. After solving the conflicts the obtained classification was presented to other two authors to validate it. At the end, they identified 7 impacted quality attributes: (i) Functionality, (ii) Performance, (iii) User Experience, (iv) User Interface Aesthetics, (v) Availability, (vi) Resource integrity and Consistence, and (vii) Testability.

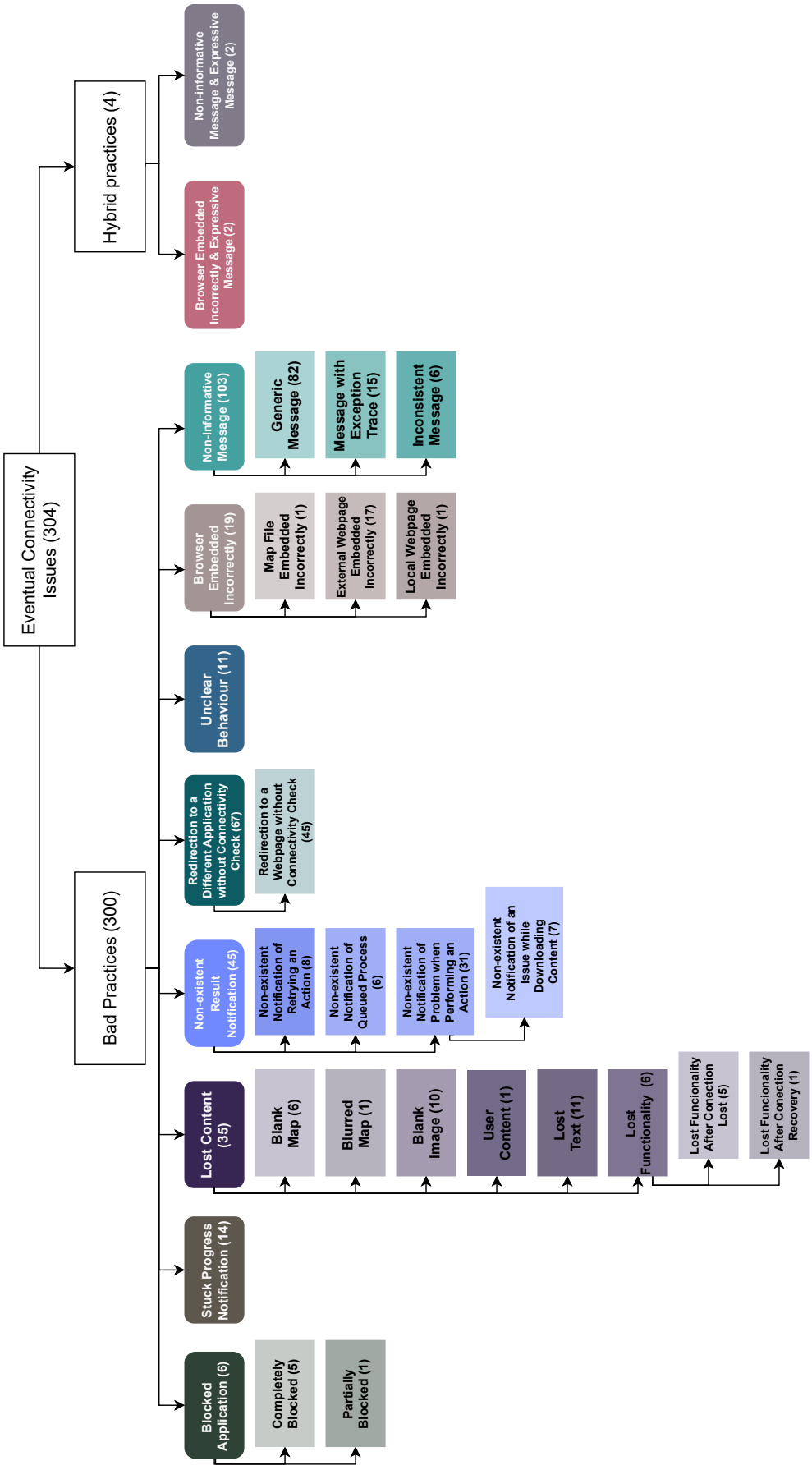


Figure 9.5. Taxonomy of eventual connectivity issues in the 50 analyzed apps. Each node contains the tag of the corresponding issue and the number of instances we found for each issue.

Table 9.3. Summary of amount of instances and affected apps per ECn issues

ECn Issue	# Instances	# Apps	Applications
Blocked Application			
Completely Blocked	5	3	7, 49, 50
Partially Blocked	1	1	6
Stuck Progress Notification			
Lost Content			
Blank Map	6	4	6, 33, 34, 35
Blurred Map	1	1	37
Blank Image	10	9	10, 15, 18, 22, 26, 30, 36, 39, 50
User Content	1	1	39
Lost Text	11	7	22, 31, 36, 39, 44, 46, 50
Lost Functionality	6	5	
Lost Functionality after Connection Lost	5	5	21, 35, 37, 39, 44
Lost Functionality after Connection Recovery	1	1	39
Non-existent Result Notification			
Non-existent Result Notification of Retrying an Action	8	4	41, 44, 48, 49
Non-existent Result Notification of Queued Process	6	4	4, 14, 19, 44
Non-existent Result Notification of Problem when Performing an Action	31	20	8, 15, 21, 25, 26, 30, 31, 35, 36, 37, 38, 39, 45, 46, 50
Non-existent Result Notification of an Issue while Downloading Content	7	5	4, 5, 10, 12, 20
Redirection to a Different Application without Connectivity Check			
Redirection to a Webpage without Connectivity Check	45	33	1, 2, 4, 6, 7, 8, 11, 12, 13, 14, 15, 16, 17, 18, ...
Unclear Behaviour			
Browser Embedded Incorrectly			
Map File Embedded Incorrectly	1	1	24
External Webpage Embedded Incorrectly	17	11	5, 9, 10, 14, 23, 25, 26, 27, 39, 47, 49
Local Webpage Embedded Incorrectly	1	1	18
Non-Informative Message			
Generic Message	82	33	1, 2, 3, 4, 5, 6, 8, 14, 15, 16, 18, 19, 22, ...
Message with Exception Trace	15	8	7, 12, 19, 34, 44, 46, 48, 49
Inconsistent Message	6	6	6, 7, 14, 23, 34, 38
Browser Embedded Incorrectly & Expressive Message			
Non-informative Message & Expressive Message			

9.3 Results & Findings

Fig. 9.5 depicts the taxonomy of eventual connectivity issues, which can be described as errors, usability issues, or confusing behaviors of an app in scenarios where a connection is required. The taxonomy includes high level categories (represented by rounded squares) and low level categories (represented by rectangles). We found 21 low-level types of issues, grouped into 10 high-level categories. Our taxonomy includes a total of 304 issues identified during our analysis & testing process. Note that the high level categories are organized into two sub-trees. The first, representing the bulk of our taxonomy, is related to bad practices we observed that resulted in connectivity issues. The second, named "Hybrid practices", groups cases in which we identified an issue in the tested app but also accompanied by a correct behavior. For example, when a connectivity issue happened, the app shows both a non-informative message and an expressive and useful message. Table 9.3 summarizes the number of instances and affected apps for each ECn issue presented in the taxonomy. Additionally, we link the ECn issues with the apps' IDs presented in Table 9.3.

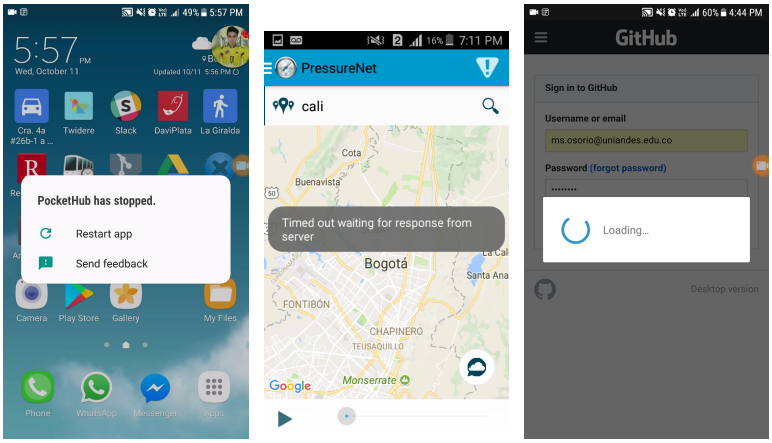
In order to highlight the more common issues identified within the study, we only discuss the categories in our taxonomy having more than one instance. Additionally, we present qualitative examples of code snippets that generate the identified issue. In our online appendix [199] we provide the complete list of eventual connectivity issues we identified, the corresponding code snippets, and videos exhibiting the issues.

9.3.1 Bad Practices

Blocked Application (BA)

In this category we classified scenarios where the app gets blocked (or crashes) due to a connectivity problem. This type of issue usually happens when the user tries to perform an action requiring network or Internet connection and the GUI is blocked as result of the lack of connection. The app and/or the device does not respond to the user's commands and it is necessary to restart the app or to wait for a long time until it unblocks. We divided this high-level category into two types: *Completely blocked* and *Partially Blocked*.

Completely Blocked (CB) refers to cases in which it is necessary to restart the app after it gets blocked. This case implies a crash or an Application-Not-Responding error (ANR), which means that the app stops and needs to be launched again. Subfig. 9.6a depicts the bug found in PocketHub *(v0.3.1)*, with the app crashing because it is not properly handling the lack of connectivity. When a user selects the Repositories tab, a new request to a backend service is made. However, there is no code for handling errors in the request/response. To get more details about the code-level issue, we inspected the source code and we found that the app is missing a null check for a returning value. In this case (see Listing 9.1, lines 112 and 113), the application uses the `getRepository` method for a service generated using Retrofit library. Therefore, due to connectionless state, `getRepository` returns a null value. Because of this, the concatenated calls starting at line 114 fall into a `NullPointerException` that is not properly handled.



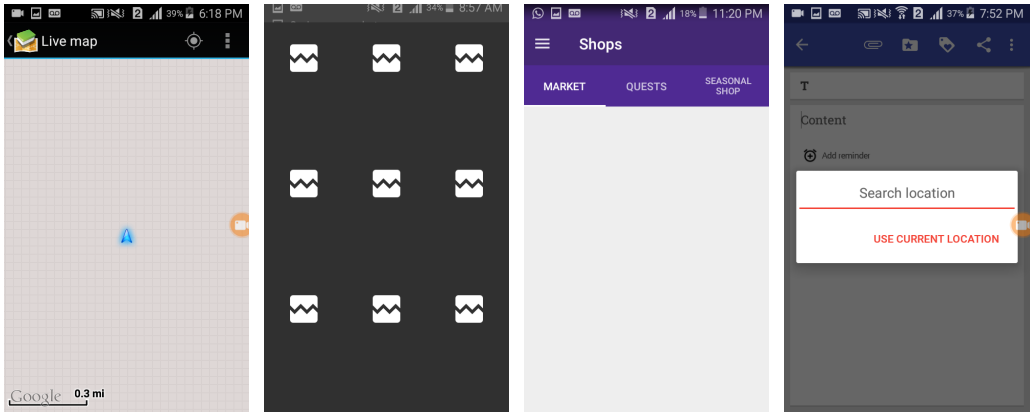
(a) Blocked Application → Completely Blocked *(b)* Blocked Application → Partially Blocked *(c)* Stuck Progress Notification - Pockethub

Figure 9.6. Examples of connectivity issues regarding Blocked Application and Stuck Progress Notification (1)

Listing 9.1. Code snippet from PocketHub (app/src/main/java/com/github/pockethub/android/ui/repo/RepositoryViewActivity.java) showing the reasons behind an example of CB.

```
106 if (owner.avatarUrl() != null && RepositoryUtils.isComplete(repository)) {
107         checkReadme();
108 } else {
109         avatars.bind(getSupportActionBar(), owner);
110         loadingBar.setVisibility(View.VISIBLE);
111         setGone(true);
112         ServiceGenerator.createService(this, RepositoryService.class)
113                 .getRepository(repository.owner().login(), repository.name())
114                 .subscribeOn(Schedulers.io())
115                 .observeOn(AndroidSchedulers.mainThread())
116                 .compose(this.bindToLifecycle())
117                 .subscribe(response -> {
118                         repository = response.body();
119                         checkReadme();
120                 }, e -> {
121                         ToastUtils.show(this, R.string.error_repo_load);
122                         loadingBar.setVisibility(View.GONE);
123                 });
124 }
```

The *Partially Blocked (PB)* type covers the cases in which the app does not respond to commands given by the user, but just for a short period of time. After waiting, the app just keeps working normally and sometimes displays a message indicating that there was an issue. This happened in the PressureNet app (see Subfig. 9.6b).



(a) Lost Content → Blank Map *(b)* Lost Content → Blank Image *(c)* Lost Content → Lost Text *(d)* Lost Content → Lost Functionality

Figure 9.7. Examples of connectivity issues in the analyzed apps (3)

Stuck Progress Notification (SPN)

The *Stuck Progress Notification* type is exhibited when a progress notification gets stuck in the GUI (not necessarily blocking the app) because of a connection problem. For example, this happens in PocketHub *(v0.3.1)* (see Subfig. 9.6c): The app shows a loading dialog while singing in and there is no connectivity. However, the dialog gets stuck. PocketHub uses an embedded `WebView` to show the GitHub login page. In PocketHub the `WebViewClient.onPageStarted` method is overridden to show the loading dialog every time a webpage is requested (see Listing 9.2); and the `WebViewClient.onPageFinished` method is overridden to dismiss the dialog when a requested page URL finishes loading.

The `onReceiveError` callback is supposed to define the corresponding action to execute. However, PocketHub does not override the method which is the right place for dismissing the loading dialog when there is a connectivity error.

Listing 9.2. Code snippet from PocketHub showing the reasons for an example of SPN.

```
59 @Override
60 public void onPageStarted(android.webkit.WebView view, String url, Bitmap favicon) {
61 dialog.show();
62 }
63 ...
64 @Override
65 public void onPageFinished(android.webkit.WebView view, String url) {
66 dialog.dismiss();
67 }
```

Lost Content (LC)

A *Lost Content* issue happens when an app (i) does not have connectivity, (ii) it shows empty, incomplete, or blurred content where it is supposed to be, and (iii) it does not notify the user about the connectivity problem.

The *Blank Map (BM)* type refers to cases in which an app provides a map functionality, and due to lack of connection, the map is blank (*i.e.,* does not show any layer of the map) and there is no message notifying the issue. For instance, the C:Geo app (see Subfig. 9.7a) allows users to see a "Live Map". However, when a user selects this option when connectivity lacks, the app shows a blank space in the view dedicated to the map. After reviewing the code (see Listing 9.3) we found that the map is created and loaded without checking the connection state.

Listing 9.3. Code snippet that generates a blank map in C:Geo

```
433 // initialize map
434 mapView = (MapViewImpl) activity.findViewById(mapProvider.getMapViewId());
435 mapView.setMapSource();
436 mapView.setBuiltInZoomControls(true);
437 mapView.displayZoomControls(true);
438 mapView.preLoad();
439 mapView.setOnDragListener(new MapDragListener(this));
```

The *Blurred Map (BMA)* type describes scenarios in which the app still shows a map even in the absence of connection. However, the map is blurred. This behavior was found only once in RunnerUp *(v1.2)*.

The *Blank Image (BI)* low-level type covers the cases in which the app uses external images inside the application and, due to connectionless state, the app displays a blank image (or empty content) where the image is supposed to be. For example, OpenGur*(v4.7.1)* shows a grid with images by using the Android-Universal-Image-Loader library. However, if the app is launched without Internet no image is loaded in the grid (Subfig. 9.7b) and the user is not notified about the issue. After inspecting the code we found that the issue is generated because (i) the connection state is not validated before filling the grid, and (ii) the developer is not using the caching feature provided by the library (Listing 9.4).

Listing 9.4. Code snippet for displaying an image using the Android-Universal-Image-Loader library (Opengur)

```
48 protected void displayImage(ImageView imageView, String url) {
49         if (imageLoader == null) {
50                 throw new IllegalStateException("Image Loader has not been created");
51         }
52
53         imageLoader.cancelDisplayTask(imageView);
54         imageLoader.displayImage(url, imageView, getDisplayOptions());
55 }
```

The *User Content (UC)* type describes scenarios in which, due to lack of connectivity, content generated by the user is lost. This behavior was found only in the chat feature of Habitica *(v1.1.6)*.

The *Lost Text (LT)* low-level type covers the cases in which a text section is not available due to connectionless state. For instance, Habitica app *(v1.1.6)*: when a user tries to enter to the "Shops" option without connection (see Subfig. 9.7c), the app shows an empty view. Specifically, Habitica uses the Retrofit library when retrieving the 'Shops' option information. As it can be seen in Listing 9.5, when there is an error it is handled using the "handleEmptyError" method (see line 132).

Listing 9.5. Code snippet showing the 'Shops' information retrieving process in Habitica

```
this.inventoryRepository.retrieveShopInventory(shopUrl)
    .map { shop1 ->
        if (shop1.identifier == Shop.MARKET) {
            val user = user
            if (user != null && user.isValid && user.purchased.plan.isActive) {
                val specialCategory = ShopCategory()
                specialCategory.text = getString(R.string.special)
                val item = ShopItem.makeGemItem(context?.resources)
                item.limitedNumberLeft = user.purchased.plan.numberOfGemsLeft()
                specialCategory.items.add(item)
                shop1.categories.add(specialCategory)
            }
        }
        shop1
    }
    .subscribe(Action1 {
        this.shop = it
        this.adapter?.setShop(it, configManager.shopSpriteSuffix())
    }, RxErrorHandler.handleEmptyError())
```

Nevertheless, as it can be seen in Listing 9.6, "handleEmptyError" sends a message to the Crashlytics log but does not notify the user.

Listing 9.6. Code snippet showing how errors are handled in Habitica

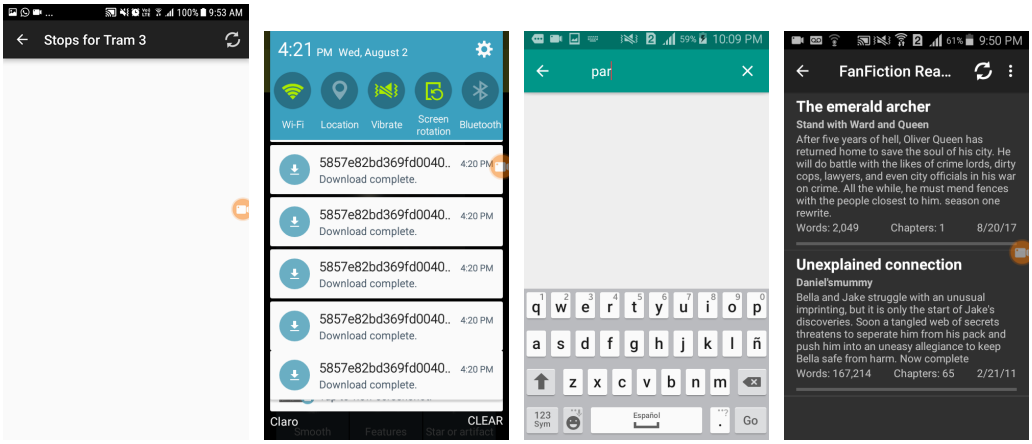
```
public static Action1<Throwable> handleEmptyError() {
    //Can't be turned into a lambda, because it then doesn't work for some reason.
    return new Action1<Throwable>() {
        @Override
        public void call(Throwable throwable) {
            RxErrorHandler.reportError(throwable);
        }
    };
}
```

The *Lost Functionality (LF)* type describes scenarios in which a functionality that requires Internet is missed, or hidden when there is no connectivity, and the user is not notified about the case. Within this category we defined two sub-categories, namely *Lost Functionality After Connection Lost (LFACL)* and *Lost Functionality After Connection Recovery (LFACR)*.

The *Lost Functionality After Connection Lost (LFACL)* category describes the behavior in which a feature of an app is no longer functional when connection status is OFF. One example from Omni Notes *(v5.3.2)* is depicted in Subfig. 9.7d. When a user selects the location option and there is connection, then a dialog is displayed; however, if the same feature is invoked without connectivity, the dialog does not show up and there is no notification about the connection error. In this specific case, the application uses the latest location the phone registered before losing connection.

Lost Functionality After Connection Recovery (LFACR) covers the issues appearing in mobile apps when the following events occur: (i) a user performs an action in the app (with connectivity) and everything works as expected; (ii) the user performs the same action again but this time without connectivity; (iii) the action is not performed because it relies on having a connection (and it is the expected behavior); (iv) the user turns on the required connection (or the connection is back) to perform again the action; (iv) the functionality is not avail-

able/functional anymore. Note that this is similar to *LFACL*. However, while *LFACL* happens when there is no connectivity, in the case of *LFACR* the functionality is “lost" after recovering connectivity. We found only one instance of this issue in the Habitica app *(v1.1.6)*, that is detailed in our online appendix [199].



(a) NRN → Non-existent notification of retrying an action

(b) NRN → Non-existent notification of queued process

(c) NRN → Nonexistent notification of problem when performing an action

(d) NRN → Non-existent notification of an issue while downloading content

Figure 9.8. Examples of connectivity issues in the analyzed apps (4)

Non-existent Result Notification (NRN)

In this case the app does not show any result or message indicating that (i) the action executed by the user or by the system was performed successfully (or not), or (ii) the user must execute a retry/refresh action. We found 45 examples of *NRN* in 26 apps.

The *Non-existent notification of retrying an action (NNRA)* type covers the cases in which a user must retry an action due to connection state but she is not notified about it. TramHunter *(v1.7)* invokes an external service each time a “Tram Station" is selected to show buses/trams schedule. However, as shown in Subfig. 9.8a, if there is no connection the application neither shows any content nor a message warning that the action must be retried later because there is no Internet connection.

A *Non-existent notification of queued process (NNQP)* issue happens when the following sequence of events occur: (i) the user attempts to perform an action without connection and the app does not execute the background process; then (ii) the connection is recovered and the application executes the actions implemented by the background process. However, the user is not notified about the execution. Galaxy Zoo app *(v1.69)* exhibits this bad practice (see Listing 9.7) when a user attempts to download a galaxy image and there is no connection. In this scenario, the user clicks the “Download" option, and the download is not executed because of the lack of connection. However, the action is queued for later execution without notifying the user that it will be executed when a connection is established, and then it is started automatically without notifying the user (again). Therefore, if the user clicks *n* times, when the connection is recovered *n* download requests will be executed. This is an issue that can have implications on resource consumption and usability, because background processes can consume a lot of device resources and the user is not notified about them.

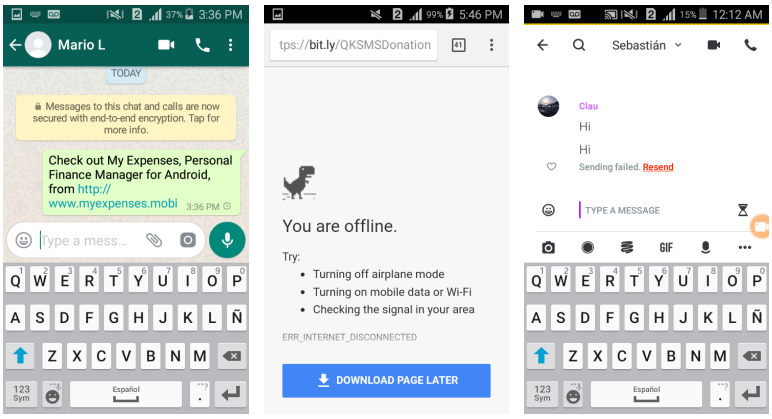
Non-existent notification of problem when performing an action (NNPPA) refers to scenarios in which an app had a problem while performing a connectivity-related action and it does not inform users about the issue. For example, with the Good Weather app *(v4.4)* users can query weather conditions for a specific location. However, when they try to search for a location without Internet, the application is unable to invoke external services for locations that match the queries and it shows an empty list of locations as illustrated in Subfig. 9.8c. It is worth noticing that the issue identified by this tag is the lack of notification regarding the disabled internet connection, not the lack of text from the response. Namely, this category should not be confused with the *Lost Text category*.

Listing 9.7. Code snippet download request handling from the GalaxyZoo app

```
195 final DownloadManager.Request request = new DownloadManager.Request(uri);
196 request.setNotificationVisibility(DownloadManager.Request.VISIBILITY_VISIBLE_NOTIFY_COMPLETED);
197
198 final Activity activity = getActivity();
199 if (activity == null) {
200     Log.error("doDownloadImage(): activity was null.");
201     return;
202 }
203
204 final Object systemService = activity.getSystemService(Context.DOWNLOAD_SERVICE);
205 if (systemService == null || !(systemService instanceof DownloadManager)) {
206     Log.error("doDownloadImage(): Could not get DOWNLOAD_SERVICE.");
207     return;
208 }
209
210 final DownloadManager downloadManager = (DownloadManager)systemService;
211 downloadManager.enqueue(request);
```

The *Non-existent notification of an issue while downloading content (NNDC)* issue describes scenarios in which an app provides a download option and there is no notification when an error occurs. This is an important category in our taxonomy since it represents 22.5% of the issues identified within NNPPA. For example, the FanFiction Reader app *(v1.51)* allows users to check manually if there are updates for the books downloaded before. However, if the

user refreshes the book without connectivity, the application tries to check for new chapters but it fails and then it does not notify the user about the failure, which could make the user thinks the last version is already downloaded. This behavior is depicted in Subfig. 9.8d.



(a) RDAC *(b)* RDAC → RWPC *(c)* UB

Figure 9.9. Examples of Redirection to a Different Application without Connectivity Check and Unclear Behavior connectivity issues in the analyzed apps

Redirection to a Different Application without Connectivity Check (RDAC)

This category is the second most frequent in our study. It occurs when an app does not check the connection before redirecting a user to a different view in which Internet connection is required. It is worth noticing that the interaction with the app the user is being redirected to is not necessarily synchronic, nevertheless, the lack of network state validation could impact the usability of the original app. For this category, we identified a special subtype in which the application to which the user is redirected to is an external browser.

In the My Expenses app *(v2.7.9)*, users can "Tell a friend" about their My Expenses usage. This feature redirects to the messaging apps in the phone (*e.g.*, WhatsApp) without checking connection state. Most of the messaging apps (chats) require connection. Thus, if there is no connection, the responsibility of telling the user that there is no connection relies on the communication apps. From the usability perspective this is not ideal because the app starting the redirection should inform the user about the connection state. Subfig. 9.9a shows an example of RDAC in which the flow is redirected from MyExpenses (without connectivity check) to WhatsApp (note the loading icon in the message because of the connectionless state). As shown in Listing 9.8, the issue happens since the app sends the *implicit intent* without checking connection state.

Listing 9.8. Example of redirection to an app without connectivity check at MyExpenses

```
603 case R.id.SHARE_COMMAND:
604     i = new Intent();
605     i.setAction(Intent.ACTION_SEND);
606     i.putExtra(Intent.EXTRA_TEXT, Utils.getTellAFriendMessage(this));
607     i.setType("text/plain");
608     startActivity(Intent.createChooser(i, getResources().getText(R.string.menu_share)));
609     return true;
```

A *Redirection to a Web Page without Connectivity Check (RWPC)* describes scenarios in which an app needs to redirect users to the browser to open a web page. An example of this case is in QKSMS *(v2.7.3)*. This app lets users send and receive SMS/MMS. When a user tries to donate with PayPal through QKSMS, it redirects the user to a web page in a browser. If there is no connection, after a while waiting for a response, the browser displays a message declaring that there is no connectivity. Subfig. 9.9b shows an example of this behavior in QKSMS when redirecting the user to a browser. This behavior is caused by not validating the connection state before doing the redirection as it can be seen in Listing 9.9.

Listing 9.9. Example of redirection to a web page without connectivity check at QKSMS

```
216 public void donatePaypal() {
217     Intent browserIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("https://bit.ly/QKSMSDonation"));
218     mContext.startActivity(browserIntent);
219 }
```

Unclear Behavior (UCB)

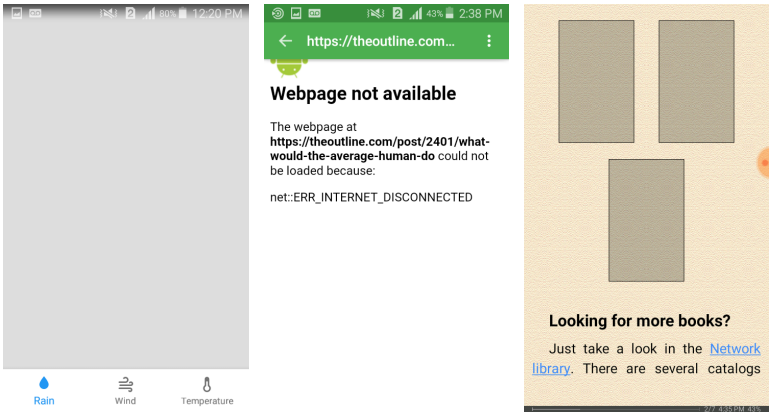
This category refers to the cases in which an app can have different or unexpected behaviors as a consequence of connectivity issues and the reasons are not clear (from the user point of view).

UCBs are exhibited in a situation in which the application, under the same scenario, has different results/responses after performing the same action multiple times. In Wire (a messenger app), when a user tries to send multiple messages to a contact without connectivity, the app shows different behaviors like (see Subfig. 9.9c). First, a message of error with a "Resend" option is displayed. After sending a few messages, a message saying "Sending..." is displayed. This situation confuses the user because the app seems to have different behaviors depending on the number of times that an action is performed.

Browser Embedded Incorrectly (BEI)

This category represents the scenarios in which the app redirects the user to an activity with an embedded content and it does not work because there is no connection. We found three sub-types for this category: *Local webpage Embedded Incorrectly*, *External Webpage Embedded Incorrectly*, and *Map File Embedded Incorrectly*.

The *Map File Embedded Incorrectly (MFEI)* type happens when an app opens an Android activity with a local map (*i.e.,* there is a local file with the map information) and it does not work because there is no connectivity. An example of this issue was detected in Forecastie *(v1.2)*, an app for checking surrounding weather conditions (see Subfig. 9.10a). It is worth



(a) BEI → Map File embedded incorrectly

(b) BEI → External webpage embeddedembedded incorrectly

(c) BEI → Local webpage embedded incorrectly

Figure 9.10. Occurrences of Anti-Patterns in Studied Applications (7)

highlighting that unlike *Blank Map* and *Blurred Map*, MFEI uses a local file to display the map, therefore, the ECn issue is generated by the developers.

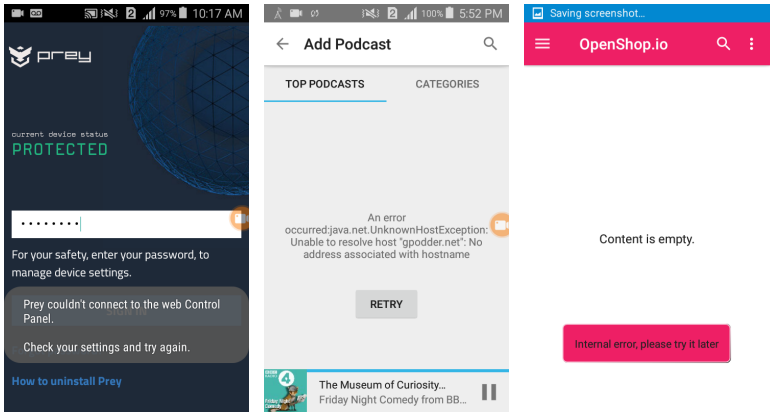
The *External Webpage Embedded Incorrectly (EWEI)* type is exhibited in scenarios in which the app opens an Android Activity with an embedded WebView that points to an external source (URL). RedReader *(v1.9.8.2.1)*, an unofficial open source client for Reddit, is affected by this issue. When a user tries to access an article without connection, the app shows an embedded page with an error stating "Web page not available. The webpage could not be loaded because: net::ERR_INTERNET_DISCONNECTED" as in Subfig. 9.10b. This scenario is problematic because (i) the error message shown by the WebView is not user friendly; and (ii) the user has to wait for a time-out until the notice. It is worth noticing that there is a difference between EWEI and RWPC, since EWEI issues do not redirect users to an external app.

The *Local Webpage Embedded Incorrectly (LWEI)* type characterizes the scenarios in which the app opens an Android Activity that has an embedded WebView, but the web content is not displayed because of lack of connectivity. We detected an instance of this type in FBReader *(v2.8.2)*, a free ebook reader (see Subfig. 9.10c).

Non-Informative Message (NIM)

A *Non-Informative Message* manifests in apps showing generic, unclear or inconsistent messages when there is a connection problem. Examples of NIMs are "An error occurred" or "<Exception>: <Exception_trace>". A more detailed list of NIMs can be found in our online appendix [199]. We distinguish three low-level types in this category: *Generic Message*, *Message with Exception Trace* and *Inconsistent Message*

The *Generic Message (GM)* type describes those scenarios in which a user is performing an action in the app, a connectivity problem arises and a message is presented asserting that



(a) Non-Informative Message → Generic Message

(b) Non-Informative Message → Message with Exception Trace

(c) Non-Informative Message → Inconsistent Message

Figure 9.11. Examples of connectivity issues (Non-Informative Messages)

something happened. However, the message does not tell anything about what the problem is, how to solve it, or what is going to happen with the user action. In Subfig. 9.11a we present an example of this behavior at the Prey app *(v1.7.7)*. Since the error handling mechanism in the app is generic (*i.e.,* it does not distinguish different types of error), Prey does not have a catch block for specific exceptions, showing a generic message to the user. Therefore, as it can be seen in Listing 9.10, the app throws a general exception called `PreyException` with the string `error_communication_exception`, that contains the message shown in Subfig. 9.11a: "Prey couldn't connect to the web Control Panel. Check your settings and try again". A similar programming error (*i.e.,* generic error handling) was found in the Surespot Encrypted Messenger app *(v70)* when using the feature to invite a friend. Any connection error when invoking the REST URL used in that feature is handled by the `onFailure` method which always displays "Could not invite friend, please try again later".

Listing 9.10. Example of error handling code with generic message (Prey)

```
304 try {
305     String uri=PreyConfig.getPreyConfig(ctx).getPreyUrl().concat("profile.xml");
306     PreyHttpResponse response = PreyRestHttpClient.getInstance(ctx).get(uri, parameters, apikey,
            password);
307     xml=response.getResponseAsString();
308 } catch (Exception e) {
309     throw new PreyException(ctx.getText(R.string.error_communication_exception).toString(), e);
310 }
```

Instances of *Message with Exception Trace (MET)* happen when there is an exception trace displayed in the app as a result of a connectivity problem. In Subfig. 9.11b we present an example of this behavior in AntennaPod *(v1.6.2.3)*, showing a message saying "An error ocurred: java.net.UnknownHostException: Unable to resolve host "gpodder.net" [...]". As is can be seen in Listing 9.11, the message that is shown to the user is built using the caught

exception.

Listing 9.11. Example of code for displaying Message with Exception Trace

```
147 txtvError.setText(getString(R.string.error_msg_prefix) + exception.getMessage());
```

Inconsistent Message (IM) groups the cases in which the app reports the execution of an action, which from the user's perspective is not related to the action triggered. In these cases, it is worth noting that despite performing the correct action, the message shown by the app suggests that another action was performed. For instance, Openshop.io *(v1.2)* exhibits this issue (see Fig. 9.11c). When a user accesses to the Terms section with Internet connection, it displays info related to the terms when buying clothes and accessories. Conversely, when the user attempts to perform the same action without Internet connection, the app states that the content (*i.e.,* the terms section) is empty (in the same place where the terms should be listed). This message is inconsistent because there is actually content that should be displayed and that, due to the lack of connection, cannot be retrieved.

Another example is in AntennaPod *(v1.6.2.3)* when a user tries to subscribe to a podcast without having connection. It is worth noting that AntennaPod downloads all media related to the podcast when the "Subscribe" button is pressed. To this, AntennaPod creates an instance of `DownloadService` (Listing 9.12). Then, the background service starts downloading the data. During the download, the method `saveDownloadStatus` sets a boolean variable `createReport` in the case of lack of connectivity. Finally, this variable is used to create a notification (Listing 9.13).

The notification displays the string `download_report_content` which is defined as "%1 $d downloads succeeded, %2$d failed" (Listing 9.14). In summary, this scenario has an inconsistent message from the user's perspective because it is not informing about the status of the "'Subscribe" request; the message reports the number of successful and failed downloads.

Listing 9.12. Code snippet showing how a background service for downloading data is started in AntennaPod

```
82 Intent launchIntent = new Intent(context, DownloadService.class);
83 launchIntent.putExtra(DownloadService.EXTRA_REQUEST, request);
84 context.startService(launchIntent);
```

Listing 9.13. Code snippet for generating a notification in AntennaPod

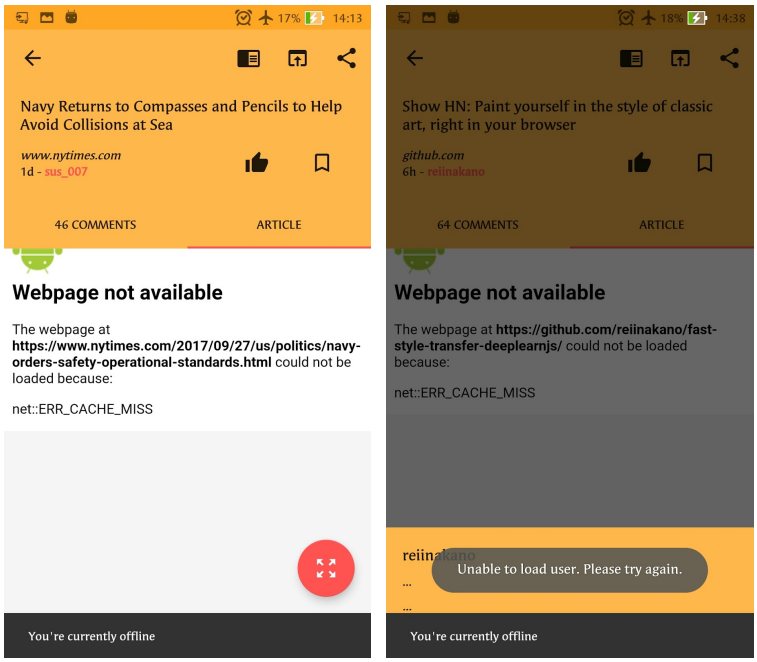
```
501 if (createReport) {
502         Log.d(TAG, "Creating report");
503         // create notification object
504         NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
505                 .setTicker(getString(R.string.download_report_title))
506                 .setContentTitle(getString(R.string.download_report_content_title))
507                 .setContentText(
508                 String.format(
509                 getString(R.string.download_report_content),
510                 successfulDownloads, failedDownloads)
511                 )
```

Listing 9.14. Inconsistent message definition at AntennaPod

```
212 <string name="download_report_content">%1$d downloads succeeded, %2$d failed</string>
```

9.3.2 Hybrid practices

Hybrid practices represent scenarios in which both good and bad practices are applied by developers. An example of these practices is when the user is trying to send an e-mail without Internet connection and two messages are displayed at the same time: One of the messages says “An error occurred" and the other message states “Your e-mail cannot be sent. It is going to be saved in the draft folder.". Note that both messages should appear separately to be considered as a hybrid practice. In this situation, there is a generic message about an error, but at the same time, there is a clear message about what is going to happen with the email because of the lack of Internet connection. This example of a hybrid practice is called *Non-Informative Message & Expressive Message (NIM-EM)*.



(a) Browser Embedded Incorrectly but with an Expressive Message

(b) Inexpressive Message plus Expressive Message

Figure 9.12. Occurrences of Hybrid Patterns in Studied Applications

In this section we present the hybrid practices identified in the analyzed apps. In order to be clear which issue and good practice are happening, the hybrid practices are named using names referring to the observed issue and good practice, using the issue name first in order to emphasize the error. Each hybrid practice has its description and a label which represents it. It is worth noting that these are less common and we found only 4 instances in the “Materialistic - Hacker News" mobile app *(v3.1)*.

Browser Embedded Incorrectly & Expressive Message (BEI-EM)

In this case, the application shows an activity with an embedded browser displaying the default content for connectivity errors, and at the same time it displays an error message instead of the expected content. We consider this case as "Browser Embedded Incorrectly" because when there is a connectivity problem, the embedded browser should not be displayed or should not display the default error page (which is not expressive). This example was found in the Materialistic - Hacker news app when a user selects a news from a list (this action triggers the display of the content associated with the news). However, for this purpose the app uses an activity with a header section where some metadata is shown (*e.g.,* News title, username of the contributor, etc.) and a body section where the url from the news is loaded using a WebView component (*i.e.,* embedded browser). Despite the error, after the load event is triggered by the WebView, the app makes a validation of the connection state in order to show a message. This message, even being an expressive message, is shown at the same time of the load message, as it can be seen in Subfig. 9.12a. Therefore, the user sees two messages that do not match in their meaning.

The complete example at code level can be seen in our online appendix [199]. However, the main code snippets are shown here. First, the starting point of this behavior as is can be seen in Listing 9.15 happens when application call "bindData" method at line 177 and then validates the connection to show a message.

Listing 9.15. Browser Embedded Incorrectly but with an Expressive Message Code Snippet in Materialistic app

```
176  if (mItem != null) {
177          bindData(mItem);
178  } else if (!TextUtils.isEmpty(mItemId)) {
179          mItemManager.getItem(mItemId,
180                  getIntent().getIntExtra(EXTRA_CACHE_MODE, ItemManager.MODE_DEFAULT),
181                  new ItemResponseListener(this));
182  }
183  if (!AppUtils.hasConnection(this)) {
184          Snackbar.make(mCoordinatorLayout, R.string.offline_notice, Snackbar.LENGTH_LONG).show();
185  }
```

When "bindData" method is triggered the webview that shows the news is started and launched as it can be seen in Listing 9.16 at lines 412 and 413.

Listing 9.16. Browser Embedded Incorrectly but with an Expressive Message Code Snippet in Materialistic app

```
407  if (story.isStoryType() && mExternalBrowser && !hasText) {
408          TextView buttonArticle = (TextView) findViewById(R.id.button_article);
409          buttonArticle.setVisibility(View.VISIBLE);
410          buttonArticle.setOnClickListener(v ->
411                  AppUtils.openWebUrlExternal(ItemActivity.this,
412                          story, story.getUrl(), mCustomTabsDelegate.getSession()));
413  }
```

Non-Informative Message & Expressive Message (NIM-EM)

In this case the app shows at the same time multiple messages when performing an action, some of them are inexpressive messages while the rest are expressive ones. For example, the Materialistic - Hacker news app redirects the user to an activity that shows that the operation was not successful, but at the same time, it shows an expressive message confirming that the user is using the app without an internet connection.

The reported example shows up when after selecting a news from a list, the user clicks on the contributors username to see more information; this action displays the content associated with the contributor. Due to connectionless state the information can not be retrieved and the app shows a generic message that does not provide enough information about the problem. However, the app makes a validation of the connection state in order to show a message. This message even being an expressive message is shown at the same time of the inexpressive message Subfig. 9.12b.

9.3.3 ECn issues impact on Quality attributes

In addition to the taxonomy figure, we present in Fig. 9.13 how the ECn bug types identified in Android might potentially impact the set of quality attributes defined in Section 9.2.4. In the figure, the columns represent quality attributes, and the rows represent ECn issues. The colors associated to each issue type (*i.e.,* from white to black using different scales of gray) indicate their level in the taxonomy, with black being root categories and white the leaf categories. A black entry at the intersection between an issue type and a quality attribute indicates a possible *negative impact* of the issue on the attribute, while a white entry represents *no impact*. By *negative impact* we mean that the issue can affect the quality attribute as perceived by the user.

The starting point for defining the impact of the ECn issues was an open coding-inspired stage in which the first two authors, who are Ph.D. students with experience in mobile app development, assigned impact-level tags to each combination of issue type and quality attribute. Once the tags were independently assigned by the two authors, a merging stage was conducted to identify the cases in which there was disagreement: The two authors agreed on 189 out of the 217 entries, which represent the intersections of ECn issue types and quality attributes, resulting in a Krippendorff's alpha coefficient [228] indicating a substantial agreement level (0.73). All conflicts have then be solved through an open discussion.

Accordingly to our assessment, ECn bugs can thoroughly impact several quality characteristics within Android applications. From a quality attribute perspective, it is worth noticing that there are three main vertical clusters. Firstly, the most impacted quality attributes are User Experience (UUX) and User Interface Aesthetics (UUI). Therefore, when users face ECn issues in most cases they cannot successfully accomplish their tasks within the app, since (i) they cannot conduct such tasks neither in an effective nor efficient manner, and (ii) the interface does not enable a pleasing interaction in presence of ECn issues. A representative issue negatively impacting such quality attributes is *Non-Informative Message*.

Secondly and consequently with the previous mentioned cluster, it is important to mention that ECn issues widely impact the Functionality (FUN) of the apps; this happens, for example, when an application is completely blocked, thus, the application does not provide internal functionalities meeting the needs of a user when experiencing lack of connectivity. Thirdly and in regards to Testability (TI), when apps exhibit ECn issues, some tests cannot be easily performed because complete test sequences can not be executed; therefore, code coverage and fault detection capabilities can be limited if ECn issues do not allow test cases to explore and execute features of an app under test.

The ECn issues having a negative impact on the widest range of quality attributes is *Lost Content*, particularly the ones related to *Lost Functionality*.

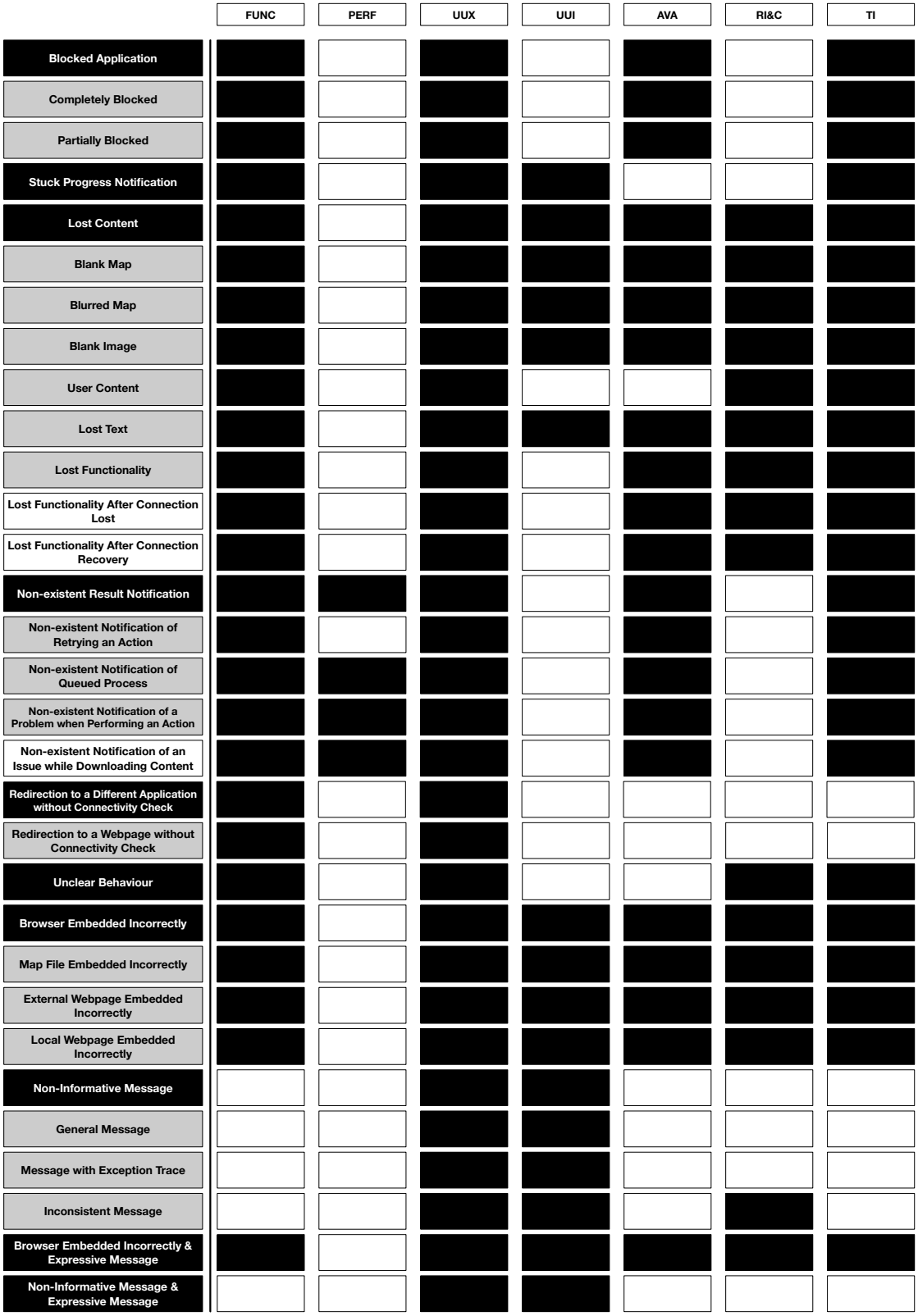


Figure 9.13. Impact of Eventual Connectivity bugs in Android apps to quality attributes. (FUNC) Functionality, (PERF) Performance, (UUX) User Experience, (UUI) User Interface Aesthetics, (AVA) Availability, (RI&C) Resource integrity and Consistency, and (TI) Testability.

9.4 Threats to Validity

Construct validity. In our study, they are mainly related to the measurements we performed, and in particular, the subjectivity during the manual tagging and construction of the taxonomy. The catalog of connectivity issues was extracted by manually testing Android apps and the testing was driven by execution scenarios defined by the authors accordingly to the guidelines described in Section 9.2.2. It is possible that some scenarios that lead to connectivity issues were not executed. However (i) we tried to be exhaustive when defining scenarios for any of the visible features requiring connectivity; (ii) the apps were tested manually to avoid any issue with automated tests (*e.g.,* inconsistent behaviour), and (iii) all the scenarios and corresponding information (including videos of the issues) are available in our online appendix [199]. In the case of issues tagging, we mitigated the subjectivity bias with weekly meetings (involving four authors) to revise the defined tags. Three of the authors have experience in developing Android apps. When a new tag was reported or when the taxonomy changed (*e.g.,* because two tags were merged) the scenarios execution was redone to check whether the already executed scenarios generated behaviors representative of the new tags. The tags, description of the tags, taxonomy, and mapping between scenarios and tags were always visible to the taggers to reduce the probability of duplicating tags. Subjectivity is also a concern when it comes to the definition of ECn issues itself. Indeed, different app's users can perceive as more/less problematic specific types of ECn issues defined in our taxonomy. In our work, we considered as an ECn issue anything that could negatively affect the user experience in eventual connectivity scenarios. Assessing the relevance of the issues we identified with developers is part of our future research agenda. Finally, a threat could be generated due to the usage of real devices, since background tasks can impact the evaluation of the scenarios and using different devices could lead to have different execution conditions between taggers.

Internal validity. We are aware that external factors we did not control could affect the apps execution and results. To mitigate the effect of those factors we designed detailed scenarios that describe the device, app and their version, connectivity states, among other information (see Section 9.2.2). Therefore, the scenarios could be replicated with the same conditions. In addition, the scenarios execution were recorded and screenshots of the results were collected (see our online appendix [199]).

External validity. We analyzed 50 open source native Android apps. We focused on open source apps because we wanted to analyze source code looking for implementation errors leading to the connectivity issues. Therefore, we recognize that we can not generalize our results to commercial/non-free apps that could follow different implementation and testing practices. Our target of 50 apps is another threat to external validity. However, we found such a number to be a good compromise between the generalizability of our study and the effort required for the data collection. Indeed, for each app we had to build the scenarios, manually execute them and manually analyze the source code looking for the root cause of the observed issue. Such a process required six months of work. While we do not claim generalizability of our findings over the set of apps we analyzed, we targeted heterogeneity of the considered apps that belong to 20 different domain categories, and have a medium-

to-high-quality as perceived by users and reported in review ratings (min=3.4, median=4.3, max=4.8). Nevertheless, we acknowledge that our findings cannot be generalized outside of the set of 50 studied apps.

It is also important to highlight our focus on connectivity issues related to the Internet connection rather than to other network protocols (e.g., Bluetooth, NFC). Such an analysis, that would require the usage of multiple devices rather than the single one we used, is part of our future work.

Finally, it is worth mentioning that, due to our selection criteria we focused on apps having an average rating above 3.0. This means that the derived taxonomy might not be fully representative of apps having a substantially lower rating that could be affected by a more diverse set of ECn issues, possibly even more severe. Additional studies are needed to investigate this aspect.

Conclusion validity. This type of threats concerns the relationship between treatment and outcome. This is an observational study and we did not consider statistical tests because we were neither interested on comparing samples nor measuring significance of differences. However, we used exploratory data analysis techniques to understand the apps sample. We report frequencies for each of the identified issues (see Fig. 9.5), and each scenario is mapped to their corresponding tags.

9.5 Additional recomendations for Connectivity management

Based on our findings we presented as an additional result a list of recommendations discussed in the following.

9.5.1 Checking the Connection Status

Most of the issues we found are related to a missing verification of the connection state before performing an action. This was the root cause for many of the errors and exceptions that, when not being properly handled, cause application blockage, execution break, and the lack of informative messages. The connection status can be obtained using the “Connectivity Manager" [229, 230]. The correct verification of connection status can prevent several of the error types in our taxonomy. For example, the *Blocked Application* issues could be avoided by not triggering a request in case the connection is not ready. Note that request time-outs can also lead to those issues, however, for this case a post-API invocation check is required to avoid issues, such as try-catch blocks for catching the corresponding exceptions, or `onErrror` method definitions when the network operation is invoked with a Future/Promise. If practitioners prefer to monitor the connectivity status while the app is running, it can be done by using a `BroadcastReceiver` as described in [231].

Another example is the Map Component that belongs to Google Maps. It uses an API in order to display a map as a Fragment. However, it does not provide a mechanism to handle the lack of connectivity when retrieving map information. This results in showing a *black map* when using this component without connectivity. We also found several cases in which accessing a WebView component without checking the connection status resulted in errors.

9.5.2 Proper Handling of Libraries

Correct use of callbacks

Most of the libraries that provide HTTP services use callbacks function. It is important to use wisely the error callback to improve the user experience. Additionally, if the backend service is not managed by the app developer, it is important to fully understand the default values used by the library as response when no result is found (*e.g.,* due to missing connectivity). Therefore, if there is an error or the response is the library's default value, the app could react properly without execution breaks.

Thread Handling

When using libraries to connect to a backend service it is important to understand how responsibilities are delegated. Some of the libraries delegate the management of threads to developers. Also, heavy processes must be performed in a worker/secondary thread (rather than in the main one handling the user interface) to improve the user experience and avoid GUI lagging and ANRs [202][1]. In the case of Kotlin, coroutines can be used with an specific dispatcher that is optimized for network operations off the main thread (*i.e.,* `Dispatchers.IO`))[232, 233]. In the case of Java-Android apps, worker threads for network operations can be implemented by using `Executor`[234], `Handler` [235], the deprecated `AsyncTask`[236], or the classic Java Threads. Finally, off-the-main-thread HTTP requests are automatically handled by libraries like Volley[237] and Retrofit with asynchronous calls [238].

Map Libraries

Using the GoogleMaps component is not the only way to display maps in Android apps. Other libraries allow developers to improve the map experience, applying custom themes and layers. However, it is important to be aware of possible issues these libraries could have, such as those documented in our study. Even more important, it is crucial for apps to properly inform the user about what is being shown. For example, if an app downloads a shell map that is used as default map when there is no connection, the user must be aware of this and knowing the specific features (*e.g.,* zooming) will not be available.

9.5.3 Considering the Behavior of Android Components

Knowing the behavior of the Android components is fundamental to properly handle cases in which connectivity errors arise. For example, the `SwipeRefreshLayout` component allows the user to use the "Swipe-down" gesture to refresh the GUI. Such a behavior is managed by overwriting the `onRefresh()` method that is called each time the user "refreshes" the GUI. While the application processes the refresh action, the `SwipeRefreshLayout` displays a

[1]Note that Androids apps are prone to GUI lags and ANRs because of the single thread policy of the Android framework.

progress notification that hides only once `setRefreshing(false)` is invoked. Therefore, if the process in `onRefresh()` fails and the `setRefreshing(false)` method is not called, the progress notification will stay on screen until a further (successful) refresh is performed.

9.5.4 Proper use of Informative Messages and Notifications

One of the most common issue we found is related to messages not providing the correct information to the user. These issues can be avoided by adopting the following practices:

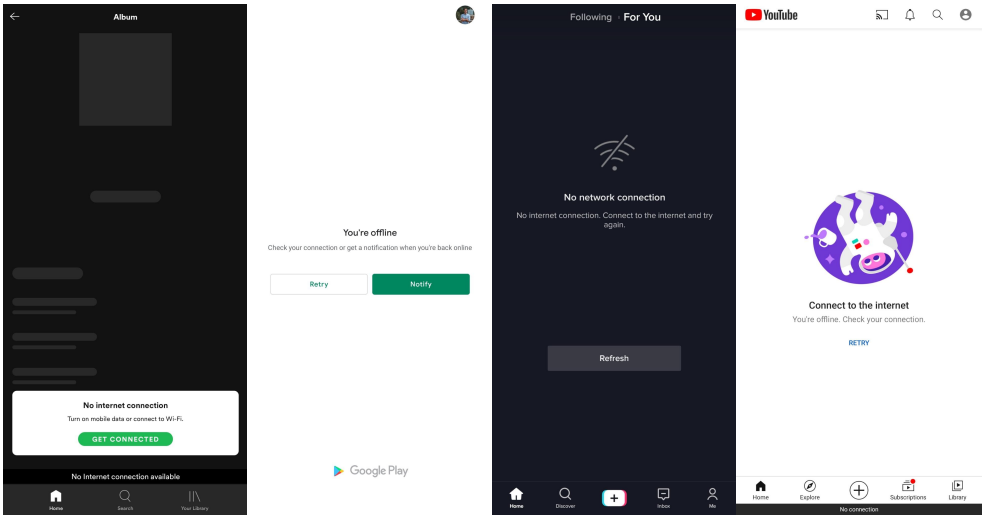
- Exception Messages, unless properly phrased by the app developer, must not be included in the messages shown to the user. Most of the times users do not know the meaning of message like: `IndexOutOfBoundsException`, `NullPointerException`, textttConnectException, etc".
- The phrasing of the messages must take into account their "consumer" (*i.e.,* the app's user): It is important to avoid technical words (*e.g.,* server, exception, connection error, *etc.*).
- Avoid reusing messages in several parts of the application: They could lead to misunderstandings due to the fact those messages are, more often than not, quite generic and may not provide enough information for users to understand the problem.

Similarly, a proper use of notifications is required. As recommended in the Android Developers Guide (ADvG), notifications should be used for foreground services and, in general, for notifying users of events relevant for them (*e.g.,* a request sent to an online service, the progress of such a request, *etc.*). A proper combination of notifications and meaningful messages can substantially improve the user experience. Fig. 9.14 illustrates examples of this notifications in Android apps that follow good practices for notifying users about connectivity issues.

9.5.5 Queuing Actions Managing User Generated Content

As shown in our study, connectivity issues can result in the lost of content or actions generated by users. To avoid this, all user-generated content and actions that require network operations (*e.g.,* sending a message in a communications apps) must be queued or locally stored when connection is not available. In this way, in case of connectivity errors, the content and actions will stay in a queue/cache and can be triggered/submitted again when connectivity is back. It is also important to show to the user that its content has not been sent/processed yet, but it is stored for future actions. This makes the user aware of both the connection problem and the "safe" state of its data. To enable this, offline-first strategies [239] must be implemented by relying on a combination of cached/local data and an actions queue. Local caching can be implemented in Android apps by using:

- Application specific on-memory data structures
- Android specific data structures such as the `LruCache` [240]
- Retrofit and Volley to enable HTTP requests caching



(a) Spotify *(b)* Google Play Store *(c)* TikTok *(d)* Youtube

Figure 9.14. Examples of expressive messages reporting connectivity issues.

- the Picasso[241] and Glide[242] libraries for image caching
- Firebase framework [243] or Realm [244] to enable caching of non-relational collection
- SQLite for local storage of relational data

To enable a local actions queue, the `WorkManager` [245] and `RequestQueue` [246] APIs can be used. In addition, as part of the JetPack architecture components, Google suggest the usage of the `Repository` design pattern [247] that allows retrieval of cached data when connection is not available.

Finally, a widely used strategy in web apps is to display "generic fallbacks" when data is neither available from cache not network. A fallback is a local resource (*e.g.,* image) that is displayed to the user when the expected data is not available, with the purpose of improving the user experience or avoiding layout issues.

9.5.6 Delegating Actions to Other Apps

When an app must perform an action already implemented in pre-installed apps (*e.g.,* sending an email), they can delegate the execution of such a feature. However, this means that the main application loses control of behavior and an error in the delegated app might lead the user to think that the main app has failed too. Knowing that the delegated actions require connectivity should push the app's developers to properly handle failing cases in order to communicate to their uses the issue experienced with the delegated app.

9.5.7 Summing Up

While we tried to summarize in this chapter the most important lessons learned from our study, our online appendix [199] provides developers with an extensive catalogue of "bad-practices" that must be avoided when handling eventual connectivity scenarios. Those bad-practices and the aforementioned recommendations could be used by researchers to design detectors and tools for automated refactoring/patching of eventual connectivity issues.

For instance, static analyses could be used to detect code statements invoking network operations and then (i) identify the existence of connectivity status checks and statements for catching connectivity errors such as time-outs or null responses, and (ii) verify the proper execution of the network operations off-the-main-thread. Such analyses will help practitioners to mitigate several of the reported issues such as *BA*, *LC*, and *RDAC*. An interesting avenue for research here is the automated detection of *NERN* and *NIM*; this requires more specialized static analyses and automated test cases generation that are able to execute the features related to the network operations and check that there are visual hints when connectivity is disabled in the device. Finally, recommender systems and tools for automated refactoring could be designed to analyze source code and suggest the usage of good implementation practices (*e.g.,* generic fallback, repository pattern, request queue, images caching, etc), and also automatically apply the corresponding refactorings.

10

Prevention of Connectivity Issues

Android is the most popular mobile operating system in the world, running on more than 70% of mobile devices. This implies a gigantic and very competitive market for Android apps. Being successful in such a market is far from trivial and requires, besides the tackling of a problem or need felt by a vast audience, the development of high-quality apps. As recently showed in the literature, connectivity issues (*e.g.,* mishandling of zero/unreliable Internet connection) can result in bugs and/or crashes, negatively affecting the app's user experience. While these issues have been studied in the literature, there are no techniques able to automatically detect and report them to developers. We present **CONAN**, a tool able to detect statically 16 types of connectivity issues affecting Android apps. We assessed the ability of **CONAN** to precisely identify these issues in a set of 44 open source apps, observing an average precision of 80%. Then, we studied the relevance of these issues for developers by (i) conducting interviews with six practitioners working with commercial Android apps, and (ii) submitting 84 issue reports for 27 open source apps. Our results show that several of the identified connectivity issues are considered as relevant by practitioners in specific contexts, in which connectivity is considered a first-class feature.

Structure of the Chapter

- Section 10.1 provides motivation for this chapter.
- Section 10.2 contextualizes the reader with instrumental background concepts.
- Section 10.3 presents the architecture of the tool created for the study.
- Section 10.4 presents the design of our study, as well as the data extraction procedure and analysis methodology.
- Section 10.5 discusses our results and findings.
- Section 10.6 presents the threats that affect the validity of our work.

Supplementary Material

All the data used in this chapter are publicly available. More specifically, we provide the following items:

Replication Package[199] The replication package includes the following material:

- **Survey & Interviews.** Survey and a set of interviews gathering practitioners' opinions concerning connectivity aspects in an Android context.
- **List of Connectivity Issues.** Detailed list of the issues addressed by CONAN.
- **Applications & Issues.** List of: (i) the set of analyzed applications by CONAN, (ii) theissues detected by CONAN within such a set; and (ii) the reported GitHub Issues.
- **Source Code.** The source code of CONAN together with its Jar file and running instructions.

Publications and Contributions

Detecting Connectivity Issues in Android Apps[17]
Mazuera-Rozo, A., **Escobar-Velásquez, C.**, Espitia-Acero, J., Linares-Vásquez, M., and Bavota, G. In *29th IEEE International Conference on Software Analysis, Evolution and Reengineering , 2022.*

CONAN: Statically Detecting Connectivity Issues in Android Applications
Mazuera-Rozo, A., **Escobar-Velásquez, C.**, Espitia-Acero, J., Linares-Vásquez, M., and Bavota, G. **UNDER REVIEW** in *22nd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2023.*

CONAN → Fig. 6.4
We present and discuss a taxonomy of changes and bugs we built based on the results obtained as part of the study conducted in the previous article.

10.1 Introduction

The always increasing popularity of mobile devices such as smartphones has triggered a strong growth of the mobile apps market in recent years. These apps support everyday activities such as online shopping, social networking, banking, *etc.* It comes without surprise that the research community put the attention on studying mobile apps from different perspectives [248, 249]. For example, several studies highlighted the role played by the apps' code quality (*e.g.,* usage of robust APIs) on their success [250, 251, 252, 253]. Among the several quality attributes that are relevant for the user experience, those related to Internet connectivity are becoming more and more important[1]. Indeed, as it has been shown in the literature [178, 179, 254, 255], network permissions are the most requested by Android apps, showing the central role Internet access plays for their functioning.

A recent work by Escobar-Velásquez *et al.* [256] studied connectivity issues affecting Android apps, namely suboptimal implementation choices that affect the user experience in *zero/unreliable connectivity scenarios*. For example, an application triggering a network operation without firstly checking the availability of Internet connection can crash in case of no connectivity [256]. The reported findings confirmed the prevalence of these issues in 50 open source apps analyzed by the authors, who reported a total of 304 connectivity issues present in them.

Building on top of the work presented in Chapter 9, we present in this paper **CONAN**, the first approach able to automatically detect **CON**nectivity issues in **AN**droid apps. We designed **CONAN** in multiple steps. First, we studied the related literature (*e.g.,* [245, 256, 257, 258, 259, 260, 261, 262, 263, 264]) and online resources such as the official Android development guidelines [265] to define a catalogue of connectivity issues to target. From this analysis, we distilled 16 connectivity issues that **CONAN** is currently able to detect by using static analysis.

Then, we defined for each of these issues CI_i a detection strategy, namely a set of rules that can be checked via static code analysis to identify CI_i's instances. The definition of these rules was not straightforward. Indeed, we soon realized that mobile apps mostly rely on external libraries (*e.g.,* OkHttp [266], Volley [237]) to handle their network (HTTP) requests. This means that the actual implementation of a detection strategy must be instantiated to specific libraries we aim at supporting. Indeed, the implementation patterns change from library to library, with different APIs involved. For this reason, we ran a survey with 98 Android developers asking which HTTP libraries they mostly use when building apps. Based on the results, we implemented in **CONAN** full support for Retrofit [238], OkHttp [266], Volley [237], and HttpURLConnection [267], which were the main libraries reported by the surveyed developers.

Once we built our tool, we ran it on a set of 31 open source apps, identifying 898 candidate connectivity issues in them. Two of the authors manually inspected a statistically significant sample (389 instances) of these candidates, to classify them as true or false positives. This evaluation provided us with information about the precision of the detection rules we implemented that, on average, resulted to be 80%.

[1]Note that in this work we use the term "connectivity" to refer to the Internet connectivity. Other types of connectivity (*e.g.,* bluetooth, NFC) are out of the scope of this paper.

Finally, we assessed the relevance of the issues identified by **CONAN**. We did this in two steps. First, we interviewed six practitioners and collected their opinions about the relevance of the 16 issue types that **CONAN** detects. Note that the focus here was on the type of issue in general rather than on specific concrete instances identified by **CONAN**. Second, we opened 84 issue reports for 27 open source apps, reporting some of the true positive instances identified in our manual analysis. The study participants indicated that the relevance of the identified issues strongly depends on the specific type of app in which they are identified. In addition, the results of the two studies highlight that, despite that all 16 issues we support represent sub-optimal implementation choices for handling connectivity in Android apps, some of them are not perceived as relevant by practitioners. **CONAN** implementation and all data used in our study are publicly available [268].

10.2 Prevention of Connectivity Issues: Background

Despite the fact that Internet connectivity is becoming increasingly important for mobile apps, quality issues related to it have not been addressed by any of the approaches presented in the literature [269]. However, related to the "connectivity topic", previous works investigated issues related to network profiling [179, 186, 187, 188], network privacy concerns [192, 193, 194, 195], and network exploitability [196, 197, 198]. Also, while no tool has been specifically designed for detecting *connectivity issues*, some of the techniques proposed in the literature to identify apps' crashes, can detect crashes caused by a mishandling of the lack of connectivity. For instance, Moran *et al.* introduced CrashScope [160, 217], a tool to test mobile apps by employing (i) static and dynamic analysis, and (ii) automatically generated gestures and contextual inputs. Similarly, Adamsen *et al.* [218] and Azim *et al.* [270] proposed tools for the automated exploration of apps, with the goal of identifying crash events. The aforementioned approaches systematically explore the app to test and generate events like intermittent network connectivity. However, these tools are limited in the type of connectivity issue they investigate (*e.g.,* intermittent connectivity).

Liang *et al.* presented Caiipa [271], a mobile app testing framework employing contextual fuzzing methods to test Android apps over an expanded context space. Caiipa is able to simulate a variety of contexts observed in the wild including diverse network-related factors, for example, the throughput and loss rates observed for Wi-Fi and 3G networks. Similarly, JazzDroid [272] is an automated fuzzer which injects into applications various environmental interferences, such as network delays, network traffic interference and HTTP requests manipulation. While these tools address a variety of bugs, they are not specialized to *connectivity issues*.

More broadly, there are methods for diagnosing poor responsiveness in Android apps given some connectivity shortcomings. Relevant tools under this category are Monkey [273], AppSPIN [274], TRIANGLE [275] and the approach proposed by Yang *et al.* [276]. These tools could eventually test Android apps looking for *connectivity issues*, by assuming that random events could lead to *connectivity issues*.

CONAN differs from current approaches since our line of action is aligned to static analysis mechanisms. **CONAN** does not need the application to be run nor the simulation of flaky

Internet connectivity. **CONAN** allows practitioners to spot potential *connectivity issues* already in the early stages of the development life cycle, since all it needs is the source code of the app to analyze.

10.3 Approach overview

CONAN is a static analysis tool built on top of Android Lint [277] and aimed at identifying *connectivity issues* in Android apps. **CONAN** is able to detect these issues for apps that use the libraries described in Section 10.3.2. The list of *connectivity issues* that **CONAN** supports has been defined based on: (i) official resources for Android developers (*i.e.,* Google developer guidelines); (ii) library-specific documentation; and (iii) research articles on this topic.

10.3.1 Android Lint Overview

Besides ensuring that an app meets its functional requirements, it is also important to guarantee its quality (*e.g.,* the absence of bugs). Therefore, the Android SDK provides a tool named Android Lint that it is able to identify code quality issues via static analysis of the app's code. Android Lint is capable of identifying over 400 issue types [278] corresponding to a variety of quality attributes such as accessibility, correctness, internationalization, performance, security, and usability. Android Lint follows the open-closed principle, allowing to add custom rules that can detect quality issues of interest.

Android Lint supports the official programming languages used in Android, *i.e.,* Kotlin and Java. While Java has been historically the official programming language for creating Android apps, Kotlin is currently the recommended language for native Android apps. By exploiting an universal abstract syntax tree (UAST), Android Lint can analyze source code written both in Java and Kotlin, since both of them are represented using the UAST while running the static analysis. Thus, it is sufficient to write the detection rule for a specific quality issue and Lint provides support for both languages. Additionally, the Android Lint is also IDE-independent, meaning that it can be run through the command line either as a standalone tool or by employing the Gradle wrapper [279] present in Gradle-based Android projects. Given these characteristics, we integrated the detection of *connectivity issues* in Android Lint.

10.3.2 Supported libraries

As explained before, the implementation of the detection strategies strongly depends on the network libraries used by developers. To decide the libraries for which **CONAN** should provide support, we ran a survey investigating which libraries Android developers use to handle network requests. The survey has been designed to last at most 5 minutes to maximize the completion rate and it was structured as reported in Table 10.1. We collected background information about participants (Q_1-Q_4). If a participant answered "zero" to Q_4 (*i.e.,* no experience with native Android apps) the survey ended and the participant was excluded from the study (4 cases).

Table 10.1. Survey on network libraries used in Android apps

BACKGROUND QUESTIONS
Q_1: In which country do you live?
Q_2: What is your current job position?
Q_3: How many years of programming experience do you have?
Q_4: How many years of programming experience do you have concerning native Android apps?
ADOPTED NETWORK LIBRARIES
Q_5: Which libraries do you use to implement network requests in Android apps?

Q_5 aimed to collect information about the libraries developers use to implement network requests in Android apps. We provided a predefined list of existing libraries, with the possibility of specifying additional libraries. The predefined list included, among others, well-known libraries such as: *Retrofit*, *OkHttp*, and *Volley*.

To avoid a leading answer bias, each participant was presented with a shuffled version of the predefined list. In addition, an option stating "*I have never used an API for network request*" was included.

Besides sharing the survey with developers from companies we know, it has also been shared in social media and within the Google Android developers group [280]. We collected answers for two months, with a total of 98 participants that completed our survey from 41 countries (*e.g.,* Germany, USA, Ukraine, *etc.*). On average, participants had ∼10 years of programming experience and ∼5 years of Android development experience. Regarding their job position, 7% of participants are B.Sc. students, 16% M.Sc. students, 2% Ph.D students, 1% Postdoc and 74% professional software engineers having different specializations (*e.g.,* CTO, CoFounder, Developer, *etc.*).

The achieved results reported five libraries as the ones used by developers for network requests: (i) `Retrofit`, mentioned by 65 participants; (ii) `OkHttp`, 61 participants; (iii) `HttpURLConnection`, 23; (iv) `Volley` 20; and (v) `Apache HTTP client`, with 19 participants. Since from Android 6.0 `Apache HTTP` is no longer recommended and its employment has been discouraged by Google, we decided to exclude it from the final list of libraries to be supported in **`CONAN`**. Concerning the rest of the libraries, these fell under a *long-tail* area, thus we included only the first four listed by developers. The complete view of the libraries mentioned by the surveyees is available in our replication package [268].

10.3.3 Connectivity Issues Supported by `CONAN`

In the context of our work, a *connectivity issue* is a suboptimal implementation choice that can affect the correct behavior of an Android app when used in *zero/unreliable connectivity scenarios*. As shown in the literature, these issues can cause crashes in apps leading to users experience issues [256]. In this section we describe the six categories of issues we support in **`CONAN`**, while the complete list of 16 issue types (belonging to these six categories) is presented in Table 10.2. The specific detection strategies are described in Section 10.3.4.

Connect to network permissions (CNP)

When performing network operations within an Android app, the manifest file (a configuration file in an app) must include some essential permissions: (i) `INTERNET`, allowing full network access to the app, thus being able to open network sockets and use custom network protocols; and (ii) `ACCESS_NETWORK_STATE`, allowing an app to access information about network connections such as which networks exist and to which ones the device is connected [257].

Despite the fact that both permissions are *normal permissions*, which means they are granted at install time and do not need to be requested at runtime, they must be explicitly included in the manifest [258]. Since the Android manifest file is manually written by developers, misconfigurations may happen, such as the lack of permissions that are actually required by the app [259].

Connection Status Check (CSC)

Before performing network operations within an app it is recommended [257, 261] to validate the state of the network connectivity, and in particular: (i) the type of network (*e.g.,* `WI-FI`, `MOBILE`, `ETHERNET`) that would allow the developers to check, for example, whether a *Wi-Fi* network is available in case large amounts of data must be downloaded (to avoid the user to incur in unexpected expenses); (ii) its availability, as a user could be experimenting flaky mobile networks, airplane mode, and restricted background data; and (iii) its Internet capabilities, in consideration of a scenario in which a user is connected to a certain network not providing Internet access. These validations should be implemented globally in the project and before triggering a network request, thus preventing an app from inadvertently using wrong configurations. Also, if a network connection is unavailable, the application should be able to handle such a case via the implementation of *offline* practices [256, 260, 264].

Network usage management (NMG)

Fine-grained control over the usage of network resources should be provided to the user if an application constantly performs network operations [281]. The user should be able to modify such settings, *e.g.,* the frequency at which the app syncs data or whether to trigger network operations only when the device is connected to *Wi-Fi*. To provide the user with such a possibility, an `Activity` offering options to control data usage should be defined within the Android manifest.

Response handling (RH)

Libraries providing HTTP services use callback functions for response and error handling. Regarding "successful" `onResponse()` callbacks and notably when back-end services are not under the app developers control, it is critical to interpret the data provided by the external service (*e.g.,* HTTP message body data and HTTP status codes) since the lack of routines

Table 10.2. List of *connectivity issues* detected by **CONAN** and their respective coverage given a network library

Category	ID	Issue	Retrofit	OkHttp	Volley	HttpURLConnection	Resource
CNP	INP	No `INTERNET` Permission	✓	✓	✓	✓	[255, 257, 258, 259]
	ACP	No `ACCESS_NETWORK_STATE` Permission	-	-	-	-	
CSC	NP	No Network connection check in Project	✓	✓	✓	✓	[256, 257, 260, 261, 263, 264]
	NM	No Network connection check in Method	✓	✓	✓	✓	
	TP	No Network type check in Project	✓	✓	✓	✓	
	TM	No Network type check in Method	✓	✓	✓	✓	
	IP	No Internet availability check in Project	✓	✓	✓	✓	
	IM	No Internet availability check in Method	✓	✓	✓	✓	
NMG	AMN	No `ACTION_MANAGE_NETWORK_USAGE`	✓	✓	✓	✓	[281]
RH	RI	No `Response` implementation	✓	✓	✓	N/A	[256, 261, 263, 284, 285, 286]
	RB	No `Response` body verification	✓	✓	✓	✓	
	RC	No `Response` code verification	✓	✓	✓	✓	
	OF	No `onFailure` implementation	✓	✓	✓	✓	
TS	SYN	Avoid *synchronous* calls	✓	✓	N/A	✓	[245, 256, 261, 262, 287]
	WM	No `WorkingManager` employment	-	-	-	-	
LBS	OK	No more than one `OkHttp` Constructor	N/A	✓	N/A	N/A	[266]

for handling exceptional scenarios can cause broken data manipulation, thus leading to a crash. Therefore, it is important to validate possible errors on the response in order to make the app react properly in such scenarios. Concerning the `onFailure()` callbacks, it is indispensable to employ the error callback to (i) handle unexpected behaviors and (ii) improve the user experience by presenting *informative messages* aimed at properly conveying the errors [256, 261].

Task scheduling (TS)

It is not recommended to implement network operations performing *synchronous* requests.

Indeed, the latter blocks the execution of code which may cause an unresponsive behavior from the app. Note that this is not an Android-specific issue [282, 283]. Avoiding *synchronous* requests reduces the chance of running network operations in the main thread, something that it is discouraged and also reported with the throwing of a `NetworkOnMainThreadException` [256, 261, 262].

Complementarily, to reliably schedule asynchronous tasks that are expected to be properly executed even under flaky circumstances the `WorkManager` API [245] is the recommended scheduling API in Android, and should be preferred to older managers (*e.g.,* `FirebaseJobDispatche` `GcmNetworkmanager`, and `Job Scheduler`).

Library specific issues (LBS)

While the above-presented categories of connectivity issues can affect network operations implemented using different libraries, we also identified a well-known anti-pattern concerning the *OkHttp* library. According to its documentation [266]: *"OkHttp performs best when you create a single OkHttpClient instance and reuse it for all of your HTTP calls. This is because each client holds its own connection pool and thread pools. Reusing connections and threads reduces latency and saves memory. Conversely, creating a client for each request wastes resources on idle pools..."*.

10.3.4 Detection Strategies used by CONAN

The majority of the issues discussed in Section 10.3.3 are related to network operations triggered by the libraries used for handling network requests. Therefore, to detect potential connectivity issues it is crucial to identify in the app's code the methods generating network requests. We refer to the points in code in which a network operation is triggered as a "network trigger" (*NT*). The identification of a *NT* is supported by the built-in call graph and interprocedural analysis provided by Android Lint that eases the identification of method calls raised within an app. This allows to easily identify (i) the *NT*, since for each library we support we know what the API methods used to start a network operation are (see next paragraph); and (ii) specific methods from the Android APIs that allow to, *e.g.,* check connectivity, verify the internet access, *etc.* These will be used in some of the detection rules described in the following.

For both *Retrofit* and *OkHttp* the network requests can be triggered by invoking `enqueue()` and `execute()`. The former follows an *asynchronous* approach whilst the latter employs a *synchronous* mechanism. *HttpURLConnection* opens a connection to a remote object given a URL by employing the *synchronous* `openConnection()` method and it lacks of an *asynchronous* counterpart. Oppositely, *Volley* only operates *asynchronously* through the `add()` method.

The current section discusses every detection strategy we implemented to detect the *connectivity issues*. A summary of the 16 supported *connectivity issues* is presented in Table 10.2, which reports: (i) the category to which each issue type belongs (among the ones described in Section 10.3.3); (ii) the name we assigned to the connectivity issue; (iii) the libraries in which it can manifest and for which we provide detection support; and (iv) the resources (*e.g.,* articles, official Android guidelines, *etc.*) we used as references to define such a connectivity issue.

1) Connect to network permissions (CNP)

No `INTERNET` Permission: If a *NT* exists within the app analyzed by **CONAN**, our tool parses the Android manifest to iterate over the declared permissions. If no `android.permission.INTERNET` element is found, **CONAN** raises a warning.

No `ACCESS_NETWORK_STATE` Permission: If an app queries (i) the state of network connectivity or (i) the status of a network interface, both `ConnectivityManager` and `NetworkInfo` packages must be imported. If these imports exist, **CONAN** parses the permissions in the manifest and raises a warning if no `android.permission.ACCESS_NETWORK_STATE` element is found.

2) Connection status check (CSC)

No Network connection check (Project & Method): **CONAN** verifies whether a *NT* exists and, if this is the case, it checks if the `isConnected()` or `onAvailable()` methods are invoked at least once in the project. If such methods have been invoked at least once but they are not being invoked in a specific method including a *NT*, **CONAN** notifies of the need for invoking these methods to check the connection state before the *NT*. If, instead, these methods have never been used in the project but *NT* exist, **CONAN** reports such a lack highlighting all code locations including a *NT*. Note that `isConnected()` has been deprecated in API level 29, with the recommendation of using instead `onAvailable()`. However, **CONAN** still considers

both methods as a viable solution to check the network status in order to be able to support also apps using an API level lower than 29.

No Network type check (Project & Method): Similarly to the above-described check for the network status, assuming at least a *NT* exists in the app, **CONAN** verifies whether a call to either `getType()` or `hasTransport()` is present in the scope of the *NT* defined in the project. Both methods allow the developer to know the type of a network (*e.g.,* `WiFi`). If these methods have never been invoked in the whole project, **CONAN** will notify the developer of such a lack at project-level reporting all *NT* in which the network type is not verified. Instead, if a specific method using a *NT* is not invoking one of those methods, a method-level warning is raised. The method `getType()` has been deprecated in API level 28 in favor of hasTransport(). Also in this case, we preferred to support both of them in **CONAN** to favor compatibility with older apps.

No Internet availability check (Project & Method): For each *NT*, **CONAN** checks if `hasCapability()` is being called within its scope with either `NET_CAPABILITY_INTERNET` or `NET_CAPABILITY_VALIDATED` as arguments. The former checks if the network has Internet capabilities while the latter coveys that Internet connectivity was successfully detected. If `hasCapability()` is never invoked, **CONAN** raises a project-level warning and all the methods including *NT* will be flagged.

Otherwise, **CONAN** only locates the methods with *NT* not having the Internet availability validation. Note that there are other non-standard ways to validate the Internet access; however, those are not currently supported in **CONAN**.

3) Network usage management (NMG)

No `ACTION_MANAGE_NETWORK_USAGE`: If **CONAN** identifies at least one *NT*, it verifies whether the application declares in the Android manifest an activity offering options to the user to control the app's data usage. Such an activity is identified by looking for the declaration of an intent filter for the `ACTION_MANAGE_NETWORK_USAGE` action. If such an intent is not present in the Android manifest, a warning is reported to the developer.

4) Response handling (RH)

No `Response` implementation: Three of the four libraries we support (*i.e.,* all but *HttpURLConnection*) allow the implementation of a `Response` callback through which it is possible to check that a response has been successfully returned to the HTTP request. **CONAN** verifies if such a response callback is implemented and extend the scope of this information by using it as a preliminary check for the detection of the following issue (*i.e.,* No `Response` body verification). If this is not the case, a warning is raised. The detection of this issue is not supported for apps using *HttpURLConnection*.

No `Response` body verification: Given a *NT* performed through *Retrofit*, *OkHttp* or *Volley*, **CONAN** checks if the `body` of their respective `Response` object is being processed through a decision-making statement including a `null` validation. For *HttpURLConnection*, such validation is being addressed with the methods `getResponseMessage()` and `getInputStream()`, since they are the ones containing the content load.

No `Response` code verification: Similarly to the previous check, **CONAN** looks for HTTP status code verifications within the `Response` object of each supported library. This can be done: in *Retrofit*, through the method `code()`; in *OkHttp*, using `code()` or `isSuccessful()`; in *Volley*, by invoking `statusCode()`; and in *HttpURLConnection* through `getResponseCode()`.

CONAN verifies if these methods are invoked within a decision-making statement looking for a `null` validation.

No `onFailure` implementation: In *Retrofit* and *OkHttp*, the error callback is called `onFailure()`, while *Volley* handles the errors with the `ErrorListener`. Lastly, *HttpURLConnection* does not have callbacks, and similarly to the cases in which the developers have implemented *synchronous NT* related to the rest of the libraries, **CONAN** verifies if they are surrounded by a `TRY-CATCH` clause. Additionally, the `CATCH` statement is expected to not be empty. **CONAN** reports a warning if (i) the method needed for the error callback is not implemented (in the case of the first three libraries), or (ii) errors are not correctly handled in *HttpURLConnection*.

5) Task scheduling (TS)

Avoid synchronous *calls*: **CONAN** detects each occurrence of *synchronous NT*, namely invocations to the `execute()` method for both *Retrofit* and *OkHttp*. Concerning *HttpURLConnection*, `openConnection()` itself is *synchronous*. Google used to suggest that these operations should be executed with an *asynchronous* workaround by using `AsyncTask`, which is a `Class` intended to enable proper and easy use of the UI thread, widely employed for this aim before Android API level 30. Currently, it is recommended to use the standard `java.util.concurrent` package or the Kotlin concurrency utilities (depending on the language used for development) or co-routines. Since this advice is recent and eventually `AsyncTask` will no longer be supported, **CONAN** pinpoints every *synchronous NT* in the app under analysis. Note that the *Volley* library does not provide any *synchronous* operation.

No `WorkingManager` employment: `WorkingManager` is an API intended to ease the implementation of reliable asynchronous tasks scheduling. **CONAN** detects in the source code files the importing of old APIs that have been replaced by `WorkingManager`, namely `FirebaseJobDispatcher`, `GcmNetworkmanager`, and `JobScheduler`, reporting such an issue to the developers.

6) Library specific issues (LBS)

No more than one `OkHtpp` Constructor: **CONAN** analyzes the entire project looking for constructor calls of `OkHttpClient()`. In case more than one instance is defined within the application a warning is raised.

10.3.5 CONAN Report

CONAN adopts the same output format of Android Lint describing the identified *connectivity issues* in a structured HTML file. For each identified issue **CONAN** provides the snippet of code in which it has been identified marking its category as *connectivity issues*. Figure 10.1 shows an example of report for the issue type *No `onFailure` implementation*. Android Lint, and consequently our tool, can also be used directly in the IDE. In this case, identified issues will be marked directly in the code editor.

10.4 Study Design

The *goal* of this study is to assess (i) the precision of **CONAN** in detecting *connectivity issues* in Android apps, and (ii) the relevance of the identified issues from the practitioners' perspective. The *context* is represented by (i) 31 open source apps on which we run **CONAN** looking

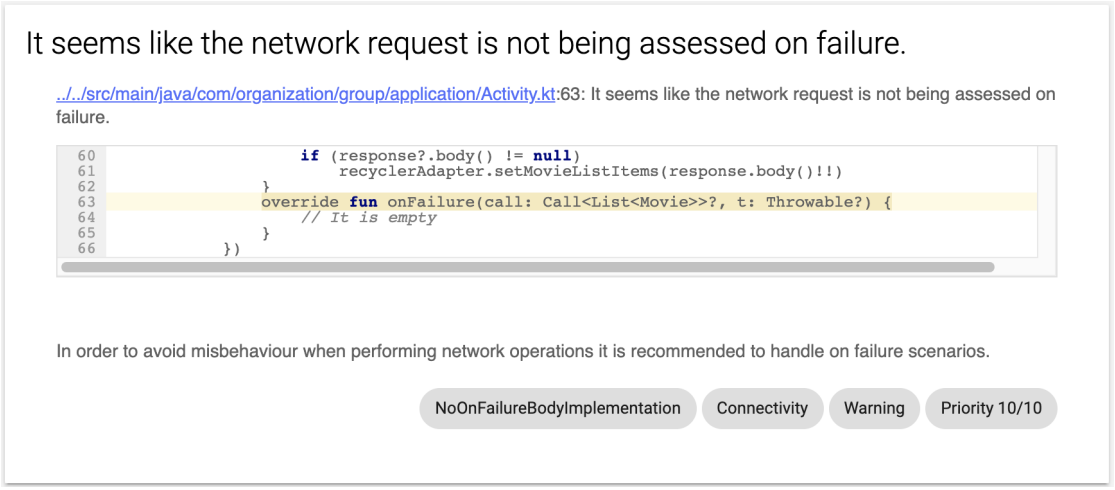


Figure 10.1. No `onFailure` implementation report

for *connectivity issues*; (ii) six interviews conducted with practitioners working with Android apps; and (iii) 84 issues we opened in 27 open source apps to report the *connectivity issues* identified by **CONAN**.

We aim at answering the following research questions:

RQ_1: *What is the precision of* ***CONAN*** *in identifying* connectivity issues *in Android apps?* We run **CONAN** on 31 open source apps, and we manually validated the reported *connectivity issues*, classifying them as true or false positives. We discuss **CONAN**'s accuracy for the different types of issues it is designed to detect (see Section 10.3).

RQ_2: *What is the relevance of the* connectivity issues *identified by* ***CONAN*** *for Android practitioners?* We answer this research question in two steps. First, we conducted six interviews with Android practitioners asking them the relevance of the *connectivity issues* issues that **CONAN** supports. Note that in these interviews practitioners did not inspect actual instances detected by **CONAN**, but were rather asked for their opinion about the *types* of issues we support (Table 10.2). Second, we opened 84 in the issue tracker of 27 of the 31 apps analyzed in RQ_1 to collect developers' feedback about the *connectivity issues* we identified. In this case, specific issue instances are studied.

10.4.1 Data Collection and Analysis

To answer our research questions, we selected 31 Android open source apps as subject systems. As a starting point we adopted the two following datasets:

Geiger *et al.* [288]. Composed of 8,431 real-world open-source Android apps. It combines source and history information from GitHub with metadata from Google Play store.

Coppola *et al.* [289]. Composed of 1,232 Android apps. The authors of this dataset mined all projects hosted on F-Droid[290], a repository for free and open source Android apps.

After removing duplicates among both datasets we obtained 8,157 unique open source Android apps, from which we excluded the ones no longer available on GitHub, leaving 8,003

apps. Then, we filtered out those not written in Java and/or Kotlin, obtaining 6,847 apps. This new dataset was then analyzed to identify repositories that could support an automated running of **CONAN**. This basically translates in the requirement that the repositories must use Gradle as their project-building tool. Thus, we identified repositories that: ① contain the source code of more than one Android app (70); ② do not use Gradle (1,902); ③ use Gradle, without however adopting the default name (*i.e., app*) for the main module to build (3,522); and ④ use Gradle, by adopting a customized name for the main module (1,353). We discarded repositories from ① and ②, since they do not support the automated running of **CONAN**. Also, we excluded all repositories not using a version of Gradle equal or newer than 6.7.1. Indeed, accordingly to the Android Lint documentation [291, 292], the version in which we integrated **CONAN** (30.0.0-alpha10) works properly on projects adopting those Gradle versions.

After these filters, we obtained a sample of 62 repositories belonging to ③ and 160 repositories belonging to ④. For the projects in ④ (*i.e.,* not using the default name for the main module), we manually identified such a name. The final set of 31 open-source Android apps was obtained as the set of projects successfully compiling and notifying at least one incident (*i.e.,* a warning raised by **CONAN**) when running **CONAN** on their latest snapshot (as of July 19th, 2021). On average, said apps have 1.6k commits, 36 releases, and 520 issues.

Table 10.3. Number of candidate *connectivity issues* identified by **CONAN** in the 31 subject apps

ID	Issue Type	#Affected Apps	#Candidate Instances	#Manually Validated	True Positives	False Positives	Precision
NP	No Network connection check in Project	16	16	16	16	0	100%
NM	No Network connection check in Method	26	163	50	50	0	100%
TP	No Network type check in Project	20	20	20	20	0	100%
TM	No Network type check in Method	26	166	50	50	0	100%
IP	No Internet availability check in Project	25	25	25	25	0	100%
IM	No Internet availability check in Method	27	191	50	50	0	100%
AMN	No `ACTION_MANAGE_NETWORK_USAGE`	42	42	42	42	0	100%
RI	No `Response` implementation	1	1	1	0	1	0%
RB	No `Response` body verification	9	16	16	1	15	6.25%
RC	No `Response` code verification	8	15	15	6	9	40%
OF	No `onFailure` implementation	13	45	45	10	35	22.22%
SYN	Avoid *synchronous* approach	23	189	50	41	9	82%
WM	No `WorkingManager` employment	3	3	3	3	0	100%
OK	No more than one `OkHttp` Constructor	6	6	6	0	6	0%
Total			**898**	**389**	**314**	**75**	

RQ_1: What is the precision of CONAN in identifying Connectivity issues in Android apps?

CONAN detected 898 candidate issues, distributed by app and type of issue as reported in the first three columns of Table 10.3. Note that for two *connectivity issues* (*i.e.,* those belonging to the CNP category in Table 10.2) we did not find any instance in the analyzed apps, excluding them from RQ_1.

Two authors performed a manual validation on a sample of the 898 candidate instances. We performed a stratified sampling: For each *connectivity issues* type T_i we targeted the random selection of 50 of its instances by considering the analyzed apps as strata. This

means that the number of T_i instances we selected from each app is proportional to their prevalence in the app. For example, if T_i affects three apps with 70, 20, and 10 instances, respectively, when extracting a random sample of 50 instances we picked 35, 10, and 5 instances from the three apps. There are two exceptions to this procedure: (i) when an app is only affected by a single T_i instance, we always selected such an instance for manual inspection; (ii) if T_i had less than 50 instances across all apps, we picked all its instances for manual analysis.

Overall, we manually analyzed 389 instances, ensuring a significance interval (margin of error) of ±5% with a confidence level of 99% [293]. The two authors involved in the manual validation independently inspected each instance, classifying it as a true positive (TP) or false positive (FP). This was done by inspecting the source code of the apps. Once completed, the two authors together inspected the 44 cases of conflict (11.31%), solving them through open discussion. The inspected instances together with the output of the manual validation are available in our replication package [268].

Using the results of our manual analysis we computed the precision, *i.e.,* TP/(TP+FP), of **CONAN** for each of the supported issue types, thus answering RQ_1. Note that we did not assess **CONAN**'s recall since this would require the manual analysis of the entire code base of all subject apps.

RQ_2: What is the relevance of the *connectivity issues* identified by CONAN for Android practitioners?

We answer RQ_2 in two steps. First, we conduct six interviews with Android practitioners in our contact network asking them about the relevance of the issue types currently supported in **CONAN**. The first part of the interview asked participants a few demographics questions including (i) the country in which they were located; (ii) their role in the company (*e.g.,* developer, tester); (iii) their programming experience and Android programming experience (in years). Then, we asked a few questions related to whether (i) they have experienced connectivity issues as app users; (ii) they adopt connectivity strategies (*i.e.,* code aimed at ensuring the proper working on the app in different connectivity scenarios) in their apps; (iii) they think connectivity strategies are part of the prioritized tasks when developing an app. After that, we asked the participants to assess their perceived relevance of the 16 *connectivity issues* that **CONAN** is able to detect. For each issue type, we showed the participant a description of the issue and asked their perceived relevance of this issue on a five-point scale: *Very Irrelevant*, *Irrelevant*, *Neutral*, *Relevant*, *Very Relevant*. The last question of the interview was: *What kind of support would you like to have from an automated tool that helps to detect these issues?* We summarize in Section 10.5 the participants' perceived relevance of the issue types we support.

Second, using the true positive instances (*i.e.,* actual *connectivity issues*) manually identified in RQ_1, we opened 84 issues in 27 of the 31 apps involved in our study (some were excluded due to the lack of true positive instances to report). When the same issue type affected the same app in multiple locations (*e.g.,* several methods of the app used network connectivity without first checking for the status of the network), we only opened one issue

summarizing all of them. For this reason, the 84 reports concern a total of 150 *connectivity issues* detected by **CONAN**. We discuss in Section **??** the number of issue reports in which developers (i) reacted by posting a comment to our issue; (ii) confirmed the issue by just commenting; or (iii) confirmed the issue and also took proper action to fix it. We also report qualitative examples of issues confirmed and "discarded" by developers.

10.5 Results

We discuss the achieved results accordingly to the formulated RQs. We start by discussing the results related to RQ_1 (Section 10.5.1) focusing on the data reported in Table 10.3 and complementing its discussion with qualitative examples. The results achieved for RQ_2 are instead presented in Section 10.5.2 (relevance of the issue types identified by **CONAN** as perceived by the six interviewed practitioners) and Section 10.5.3 (perceived relevance of the issue instances identified by **CONAN** in open source projects). We provide in our replication package [268] (i) the **CONAN** tool, which is available as a JAR file with instructions on how to run it; (ii) the list of 898 *connectivity issues* detected by **CONAN**; (iii) the transcriptions of our interviews (translated from Spanish to English); (iv) the list of opened issues; and (v) the list of analyzed apps.

10.5.1 RQ_1: Precision of CONAN

Looking at Table 10.3, it is first important to notice the lack of specific *connectivity issues*, for which no instances have been identified in our study. These are the issues belonging to the network permissions, and in particular "No `INTERNET` permission" and "No `ACCESS_NETWORK_STATE` permission".

We believe that this result is a consequence of our experimental design, in which we run **CONAN** on the latest available snapshot of the subject apps. Indeed, issues related to permissions are more likely to affect the app in the early stages of the software lifecycle, since those issues are easy to spot, not allowing the app to properly work.

Concerning the *connectivity issues* within the Connection Status Check (CSC) category, this block of issues comprises most of the identified instances, with a total of 581 candidate instances, accounting for 64% of total instances. Based on our manual validation, all the issues in the CSC category were detected with a 100% precision (see Table 10.3).

The CSC issue depicted in Fig. 10.2a is a representative example of a network operation performed without verifying the *network connection*, *network type* nor the *Internet availability*. The presented snippet of code is in charge of opening an `InputStream` to the HTML content of the given URL. The *NT* in this scenario is `openConnection()`, which returns a `URLConnection` instance that represents a connection to the remote object. Similarly, in Open Song Tablet (see Fig. 10.2b), the developers are downloading songs from a URL, also in this case implementing a *NT* with `openConnection()`. Despite the downloading operation is performed within a `TRY-CATCH` block that can handle eventual exceptions, the lack of specific checks for network connection (NM), network type (TM), and Internet availability (IM) does not allow to notify the user about specific errors occurring when running the *NT*,

leaving the user with a generic error. The lack of informative messages in case of connectivity errors is one of the main issues recently highlighted in the literature [256].

Project: mtg-familiar | File: FamiliarActivity.java

```
InputStream getHttpInputStream(URL url …) {
        ...
        HttpURLConnection.setFollowRedirects(true);
        HttpURLConnection connection = (HttpURLConnection) (url).openConnection();
        ...
        connection.setRequestProperty(...);
        connection.setConnectTimeout(5000);
        connection.setInstanceFollowRedirects(true);}
```

(a) Lack of connection status check in **mtg-familiar**

Project: OpenSongTablet | File: DownloadTask.java

```
protected String doInBackground(String… sUrl) {
InputStream input = null;
OutputStream outputStream = null;
HttpURLConnection connection = null;
    try {
          URL url = new URL(address);
          connection = (HttpURLConnection) url.openConnection();
          connection.connect(); … }}
```

(b) Lack of connection status check in **OpenSongTablet**

Figure 10.2. Missing connection status verification

Out of the 44 inspected apps, 42 of them do not implement the intent filter `ACTION_MANAGE_NETWORK_USAGE` (AMN issue), meaning that they do not allow their user to have fine-grained control over the app's usage of network resources. Also in this case all detected instances we manually validated were classified as true positives. Differently, for issues related to the *response handling*, **CONAN**'s precision is not consistent across the different issue types.

Concerning *RI* (*i.e.,* the lack of `onResponse` implementation), **CONAN** only reported a single instance that was classified as a false positive, since developers used a non-standard mechanism to handle the request. Also in this case, it is likely that this type of issue affect the app in the early development stages. In fact, in mature projects the `onResponse` implementation is expected. Thus, the identification of this type of issue is probably more useful when using **CONAN** inside the IDE rather than when running it on newer versions of the app.

Similarly, the identification of *RB* (no response body verification) and *OF* (no `onFailure` implementation) also resulted in low precision values, being 6.25% and 22.2%, respectively. This suggests that the detection rules we defined must be revised and refined to handle special cases we did not consider. For example, in case of *RB*, among the cases highlighted as false positive, we noticed that developers were assigning the `body()` content to a variable and then a `null` check was performed. However, **CONAN** checks if the `body` given by a `Response` object is being processed by the app through a **direct** decision-making statement such as a `null` validation, meaning that this heuristic must be enhanced in future releases of our approach. Moreover, concerning *OF*, in some false positives we found the presence

of logging elements (*e.g.,* `Log.d()`) printing the errors in response to a failure. While these are "borderline" strategies to handle errors, we still decided to consider them as a false positives. A better precision (40%) was achieved by **CONAN** when detecting instances of *RC* (no response code verification). Still, also here margins for improvement are present.

Moving to the issues related to the *task scheduling*, *i.e.,* "avoid *synchronous* approach" (*SYN*) and "no `WorkingManager` employment" (WM), the precision is quite high, with 82% and 100% of correctly detected instances, respectively. However, for the "no `WorkingManager` employment" issue we only assessed the three candidate instances **CONAN** identified. Several examples of *SYN* issues within the **Ultrasonic** app are shown in Fig. 10.3. Developers are using `OkHttp`'s `execute()` method in order to synchronously retrieve some resources from an external server. This issue was detected in an Kotlin file.

Project: Ultrasonic | File: RESTMusicService.kt

```
val response = API.getSongsByGenre(...).execute().throwOnFailure()
val response = API.getUser(username).execute().throwOnFailure()
val response = API.getBookmarks().execute().throwOnFailure()
val response = API.getShares().execute().throwOnFailure()
```

Figure 10.3. Several *synchronous* operations in **Ultrasonic**

Lastly, for the issue specific to `OkHttp`, *i.e.,* no more than one `OkHttp` Constructor (*OK*), **CONAN** found six instances all labeled as false positives (*i.e.,* 0% of precision). This is due to the fact that **CONAN** was run not only on the source code of the subject apps, but also on the code of their dependencies. This means that it identified several instances of `OkHttp` clients that, however, were not due to multiple instances declared within the app, but in the app and in its dependencies.

This can be avoided by simply setting an Android Lint property, indicating that it must only be run on the app's code. Note also that such a setting only affected the detection of this specific type of issue (*i.e.,* all other reported instances were related to the app's source code).

10.5.2 RQ_2: Relevance of connectivity issues (Interviews)

We focus our discussion on the relevance of the types of *connectivity issues* **CONAN** supports as perceived by the six interviewed Android practitioners (Fig. 10.4), while the answers they provided to all our questions are available in the replication package [268]. Participants have, on average, ∼7 years of overall programming experience and approximately ∼3 years concerning Android engineering tasks. The rows in the heat map represent the six interviewed practitioners (indicated with I*n*, with *n* going from 1 to 6), while the columns are the types of issues we investigated. The rectangle at the intersection participant/issue-type can have different colors based on the specific answer we got. First, we use a 5-step color scale from blue to red representing the reported relevance: dark blue for *very irrelevant*,

light blue for *irrelevant*, white for *neutral*, light red for *relevant*, and dark red for *very relevant*. Additionally, there are 2 marks that are used in the figure to represent: (i) the cases in which a participant did not answer the question (*i.e.,* a **NA** is shown in the rectangle); and (ii) cases in which the interviewee stated that the relevance of the issue depends on the app analyzed (*e.g.,* by how important is connectivity for that app)—an asterisk inside the rectangle is shown in this case. In the latter case, Fig. 10.4 reports the highest relevance perceived by the interviewee (*e.g.,* the answer "it can be very relevant but it depends on the connectivity usage of the app" is represented with a dark red rectangle with an asterisk inside).

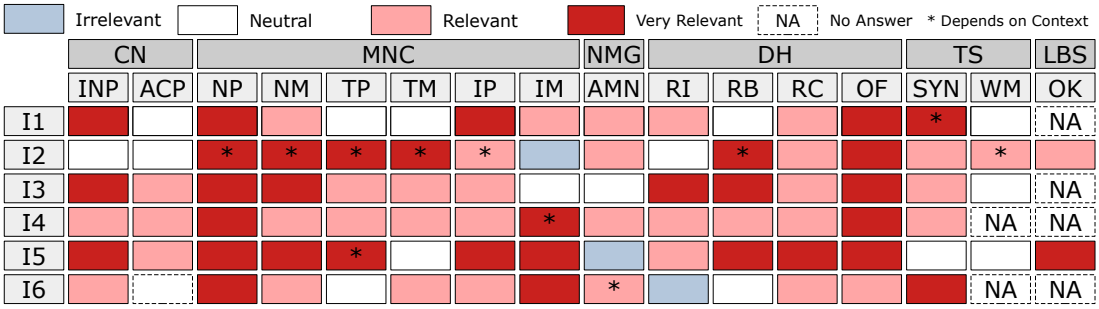


Figure 10.4. Issue relevance classification from interviews

As it can be seen from Fig. 10.4, none of the issues was tagged as *very irrelevant* by the interviewees. Nevertheless, three of the interviewees reported one case (each) as irrelevant. Specifically, I2 said that checking Internet availability at method level is irrelevant if there is already an existent check at project level; I5 claimed that allowing users to manage network usage is irrelevant since efficient design processes should contemplate that factor; finally, I6 said that checking the implementation of response handling is irrelevant since they never saw such a problem in any of the app they worked on. However, as it can be seen from Fig. 10.4, for each of these three issues there were other developers finding them relevant.

The issues perceived as more relevant by practitioners are: (i) *No Network connection check in Project*, (ii) *No onFailure implementation*, (iii) *No Network connection check in Method*, and (iv) *No Internet availability check in Project*. Basically, issues related to the lack of checks for connectivity and the lack of handling error conditions. For example, I1 talking about the *No Network connection check in Project* issue, said: "*In our case, it would be very relevant. In our apps we start by validating that a network connection exists; we have a background process for that that starts with the opening of the application, it is like a daemon continuously checking*".

I5 talking about the *No onFailure implementation* issue said:

"*Yes, I have seen it and I think it is very relevant. In fact, for me it is so relevant that I usually prefer to implement the error paths and then the success path, because in a distributed environment I do not know what is happening to the device*".

On the other side there are issue types that practitioners do not find relevant. This is the case of the *No* `ACCESS_NETWORK_STATE` *Permission* (ACP) that was marked by three interviewees as "neutral" in terms of relevance, since they claimed that the associated per-

mission is added along with the *`INTERNET` Permission*. Also, interviewees do not find the *No `WorkingManager` employment* (WM) as an actual issue, since the benefits provided by this component for asynchronous tasks are considered minor. Finally, four of the six participants do not use the *OkHttp* library and, thus, did not assess the relevance of the *No more than one `OkHttp` Constructor* (OK), that was marked as relevant and very relevant by the remaining two practitioners.

As a final note, it is worth noticing that five out of the six participants indicated that the relevance of at least one of the assessed issue types depend on the specific context (*i.e.,* app) in which it manifests and, more in particular, on the importance that connectivity has in the affected app.

To summarize, we found that: (i) for 13 out of the 16 issue types, there is a majority of participants (at least 4 out of 6) assessing them as *relevant* or *very relevant*; and (ii) the relevance of specific issues is influenced by the context in which they appear.

10.5.3 RQ_2: Relevance of connectivity issues (OSS projects)

We opened 84 GitHub issues in 27 of the 31 apps we studied. These issues represent a total of 150 *connectivity issues* since, as previously said, a single opened issue can concern multiple instances of the same issue type (*e.g.,* multiple instances of *No Network connection check in Method*). Once opened the issues, in the following six weeks we interacted with the developers who reacted by posting a comment and/or taking other types of action. After the six weeks, we summarized the status of the 84 issues.

Out of 84, 53 resulted in a reaction from the developers, which means that either they provided us a feedback by commenting or they modified the status of the issue (*e.g.,* by closing it, adding a tag). Out of these 53 issues, 15 have been recognized as *connectivity issues*, and for 8 of them the developers implemented a change to address the issue. For example, developers of *OpenSongTablet* commented: "*Many thanks for alerting me to this. This issue is addressed in an upcoming version that is still in early alpha [...]*".

Concerning the remaining 38 issues, in 14 cases the issue was just closed without any interaction with the developer. The remaining ones were tagged by the developers as false issues, leading to the their closure. Nevertheless, only on 24 of those, the developers provided us feedback regarding the issues. In one of those issues, the developers pointed to the low relevance of the type of issue (*i.e., No Network connection check in Method*) in *BinaryEye*.

Indeed, the developer explained that "*If this app would make many requests at once, then I would probably check the network connection before making all these requests. But for just one request per scan, this would be overkill*". In the other cases, the negative outcome (*i.e.,* the lack of fixing) was due to the fact that the developers did not recognize the reported issue as a potential point of failure in connectivity scenarios, does not justifying changes to the app's code.

Interestingly, we did not observe any specific pattern in terms of types of issue that were considered relevant/not relevant by developers. In other words, some issue types were considered relevant by some developers (with a consequent fixing) and irrelevant by others (with a closing of the issue).

These findings, that were quite surprising for us, point to three interesting observations. First, as already observed in our interviews, the relevance of the *connectivity issues* we identify is substantially impacted by the context in which they appear (*i.e.,* by the affected app). Second, it is possible that some of the issues identified by **CONAN**, while recognized in the literature or in the technical documentation, are unlikely to manifest in failures, thus being mostly considered as "good practices" rather than potential "bugs" to fix. This implies that a better reporting of the severity of the identified issue might be needed in **CONAN**, possibly exploiting an "adaptive" mechanism that can change the severity level depending on the app under analysis. Third, while it is clear that more work is needed on **CONAN**, we found as a positive result the identification of 26 confirmed *connectivity issues* (reported in the 15 confirmed issues) in open source apps.

10.6 Threats to Validity

Construct Validity. To assess the precision of **CONAN**, we relied on a manual analysis independently performed by two authors, reporting the number of conflicts raised in this process. Despite such a double-check, we are aware that imprecisions are still possible, and this is one of the reasons why we share in the replication package [268] the output of our manual validation. Concerning our second research question, we acknowledge that both in the interviews and in the study involving the opening of issues, the reported results represent the subjective perception of practitioners.

Internal Validity. The set of *connectivity issues* that **CONAN** supports was derived by the analysis of the literature, of the official Android developer guidelines, and of the documentation of the libraries used to establish HTTP connections.
We acknowledge that our contribution may be limited by the fact that **CONAN** is tied to specific versions of certain libraries and that other, and possibly more relevant *connectivity issues*, might have been missed in our study. Similarly, the detection strategies we defined *connectivity issues* were based on our intuition. Further research is needed in order to optimize/improve them (*e.g.,* by using data-flow analysis).

External Validity. In RQ_1 the assessment of **CONAN**'s precision is based on a set of 389 manually validated instances. Additional instances should be inspected for better generalization as we acknowledge that static analysis approaches are prone to introduce false positives. For RQ_2, we preferred interviews with a limited number of practitioners (six) rather than surveys with more participants, since interviews allowed to better understand and interpret the practitioners' perceptions of the *connectivity issues*. Finally, the number of issues we opened, while limited (84), still covered a total of 150 *connectivity issues*.

11

Connectivity Management for Android Apps: Discussion and Summary

We analyzed 50 popular open-source apps looking for bad practices. We executed 971 scenarios designed to test app functionalities that rely on Internet connection, with different connectivity scenarios. As a result, we found 320 issues grouped into ten categories. After categorizing the issues, we analyzed the source code of the apps to identify the reasons for the issues. We will be able to recognize and propose recommendations that practitioners and researchers can use to create new automated tools for detecting and preventing connectivity issues on Android apps. As an example of these new tools, we created **CONAN**, the first approach and tool in the literature designed to detect, using static analysis, *connectivity issues* affecting Android apps. The list of 16 *connectivity issues* supported by **CONAN** has been derived from the existing literature and by studying the technical documentation of Android and of libraries usually adopted to handle HTTP connections in Android apps (as a result of a survey with 98 Android developers). The precision of the detection strategies we implemented has been evaluated by manually inspecting the candidate instances identified by **CONAN**, showing that improvements are needed for detecting 5 of the 16 issue types. In contrast, excellent precision is achieved in the remaining 11 cases.

We then assessed the relevance of the issue types detected by **CONAN** by interviewing six practitioners, showing that they found most of the issues relevant, despite reporting their relevance as influenced by the specific context (app) in which they appear (*e.g.,* domain category). Such a finding has also been confirmed in the subsequent evaluation by opening 84 issues in open source projects, with contradicting observations provided by the developers who managed those issues. In 15 cases, they confirmed the issue (for a total of 26 actual code issues identified, since one issue report could concern multiple *connectivity issues*), while in 24 cases, they closed the issue as non-relevant for their app.

The results are not entirely positive nor negative, indicating the need for additional research on connectivity issues and practical detection tools considering different Android artifacts (*e.g.,* APK). Also, further studies with developers are needed to understand better the contexts in which connectivity issues are considered relevant.

Likewise, based on our results, practitioners and researchers could design and develop a

solution that enables *self-adaptness* of apps during runtime. In order to achieve this, a new approach using annotations could be designed to reduce the amount of boilerplate code required to cover the different states of the context regarding a quality attribute. For example, regarding connectivity, developers might need to specify particullar behaviours when device is connected to WiFi, connected to data, connected to a network without internet, when connection is unstable or when there is no connection. This case evaluation might require several condition statements that could be removed by using the proposed approach. This approach can also be extended by using APK modification to adapt applications right before being downloaded according to variables market places have access to like device information.These new approaches might improve the development process, reducing the current boilerplate code to manage all the context changes within an Android device.

Part V

Automated Testing of Android Apps

Structure of Part V: Automated Testing of Android Apps

- First section outlines the main concepts and previous work related to Automated Testing of Android Apps
- **Chapter 12** presents the details of Kraken-Mobile our platform-agnostic multi-device Interaction-based testing tool for Android and Web apps.
- **Chapter 13** presents our first approach towards applying DRL towards the generation of a systematic exploration approach for android Apps based on Kraken-Mobile.
- **Chapter 14** presents a summary of findings, contributions and open research opportunities created as result of our work regarding Interaction-based Testing for Android Apps.

In parallel to the wide adoption of mobile apps for human activities, a tremendous progress on tools and techniques for functional automated testing of mobile apps has been done [294, 295, 296]. Those efforts from both, researchers and practitioners, have lead to open source and closed tools that can use different testing techniques described by the following categories: (i) Automation API-based testing, (ii) record-and-replay execution, (iii) random input generation, (iv) systematic exploration and model-based testing, (v) search-based testing, (vi) machine learning-based testing, and (vii) hybrid approaches that combine any of aforementioned techniques.

An example of these techniques is End-to-End testing (E2E), defined as a technique to validate software quality at the system and acceptance levels by executing scenarios that combine several use cases. This implies that this testing technique does not restrict the test execution to a specific functionality and is more focused on complex test cases. For example, in an application for communication, an E2E scenario could include the following steps: a user login, retrieval of the list of contacts, open a chat, and finally sending of a message.

Several approaches and tools have been proposed for supporting E2E testing of web and mobile applications, but without enabling multi platform interaction. An example of the existing tools is Appium [53], that allows writing tests for web or mobile apps, by using specialized frameworks that transform Selenium[297] commands into web and mobile specific commands. This enables E2E testing for both platforms. Nevertheless, it does not provide support for the interaction across different devices and platforms. Another existing approach is BrowserStack[298] that offers the execution of Selenium[297] tests in parallel in a user-defined grid of browsers. BrowserStack also supports the execution in parallel of Android app tests written using Appium[53] or Espresso[52]. This tool allows for writing tests that are device-agnostic, however, it does not enable communication between platforms to complete the interaction circle for a multi-platform app.

In the case of mobile apps, there are tools such as Calabash [55], UIAutomator [51], and Espresso [52]. These tools enable E2E for within Android apps, but all present downsides to solve the platform-agnostic environment presented previously. For example, UIAutomator is capable of performing tests that involve several apps within the same device. Nevertheless, the main app is treated as a blackbox, limiting the processing of the generated result. However, none of the mentioned approaches enables the interaction between android devices.

One might think that this interaction problem can be solved by using existing communication protocols such as Bluetooth, Server Sockets or even NFC. Nevertheless, these approaches introduce new limitations such as no support for offline-mode scenarios for internet-based protocols, as well as low consistency due to specific implementation of those protocols between devices and platforms.

To the best of our knowledge the only approach that enables interaction between devices is Octopus[299], a private solution created by the Uber Engineering team. Octopus, generates a communication channel between mobile devices by creating a message based interaction that are stored and handle by a orchestrator device. Despite Octopus proposes a solution to the inter-communication testing problem, it is not available for users outside of Uber, and works only for mobile apps.

In parallel to the aforementioned efforts, several articles have presented the benefits, weaknesses and limitations of different testing techniques [131, 150, 300, 301, 302], such as high rate of invalid events in random input generators, lack of historic context in model-based input generation, flakiness in manually created or automatically recorded scripts, coverage-oriented testing vs. fault-oriented testing, among others. All in all, mobile app testing is an interesting avenue for research with many open challenges. One promising field that tries to solve some of those limitations, such as the lack of historic context in event sequences, is ML-based testing, and in particular Reinforcement Learning (RL) based testing, which recently has shown some benefits over other automated techniques; in fact, there is an increasing trend in the volume of papers describing RL-based approach for mobile app testing [303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313].

12

Enabling Automated Platform-Agnostic Multi-Device Interaction-based Testing for Android and Web Apps

Mobile devices and apps have a primordial role in daily life, and both have supported daily activities that involve humans interaction. Nevertheless, this interaction can occur between users in different platforms (e.g., mobile and web) and devices. Because of this, developers are required to test combinations of heterogeneous interactions to ensure a correct behavior of multi-device and multi-platform apps. Unfortunately, to the best of our knowledge, there is no existing open source tool that enables testing for those cases. In this chapter, we present Kraken 2.0, an open source tool that enables the execution of platform-agnostic interaction-based End-2-End tests for Android and web apps.

Structure of the Chapter

- Section 12.1 provides motivation for this chapter.
- Section 12.2 presents the architecture of Kraken 2.0.
- Section 12.3 shows the process to use Kraken 2.0.
- Section 12.4 discusses our results and findings.

Supplementary Material

All the data used in this chapter as well as our tool are publicly available. More specifically,we provide the following items:

Online Appendix[314] The online appendix includes the following material:

- **Source Code.** **`Kraken 2.0`** source code is available as a GitHub repository.
- **Installation.** Installation instructions and prerequisites are provided.
- **Usage.** Usage and parametrization instructions.
- **List of Applications.** List of Android applications assessed using **`Kraken 2.0`**

Publications and Contributions

Kraken-Mobile: Cross-Device Interaction-based Testing of Android Apps[315] Ravelo-Méndez, William and **Escobar-Velásquez, Camilo** and Linares-Vásquez, Mario. In *Proceedings of the 35th IEEE international conference on software maintenance and evolution (ICSME), 2019.*

Kraken: A Framework for Enabling Multi-Device Interaction-Based Testing of Android Apps[19]
Ravelo-Méndez, William and **Escobar-Velásquez, Camilo** and Linares-Vásquez, Mario. In *Science of Computer Programing*, 2021

Kraken 2.0: A platform-agnostic and cross-device interaction testing tool[20] Ravelo-Méndez, William and **Escobar-Velásquez, Camilo** and Linares-Vásquez, Mario. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.*

Kraken 2.0: A platform-agnostic and cross-device interaction testing tool[21] Ravelo-Méndez, William and **Escobar-Velásquez, Camilo** and Linares-Vásquez, Mario. In *Science of Computer Programing*, 2023

12.1 Introduction

Interaction with devices that have computing capabilities has become part of daily life tasks; it includes the usage of applications that are executed on desktops and laptops as well as mobile devices such as phones and tablets. Along with this plethora of available devices, users also have different options concerning the platforms used for executing and distributing applications, *e.g.,* web and mobile versions of the same app.

The availability of different app execution environment, have exerted a pressure over developers since users expect high quality interactions between their different devices when using an app. Additionally, developers must ensure the correct behavior of apps in scenarios where users interact on different platforms. An example of this are social media apps, that allow users to post content at web browsers and interact with it from a mobile device, or in the other way around (*i.e.,* mobile then web). Therefore, developers and practitioners need to test not only application scenarios executed on a specific platform but also interaction between users from different platforms or devices, and interactions of the same user but on different platforms. However, while testing of web and mobile applications are research fields that have proposed a plethora of approaches and tools, no current publicly available tool allows for creating testing scenarios that are executed on different platforms in an interactive way; automated testing of apps that provide cross-platform interactions is not a well explored field yet.

An example of this behavior can be an email communication between two people; at the beginning of the interaction both of them are using their browser to exchange emails, and at some point one person needs to change into the email android app, so she writes back from her cellphone. After answering a couple of the emails, the user that was still using the web browser decides to continue later on the mobile app, but she decides first to write a draft of its last email so he can finish it from its cellphone. At this point, these interactions have required different versions of an app to be capable of interact between them. Additionally, it has depicted not only interaction between different users, but also the interaction of the different environments of the same app . This type of interactions are common between people, but in order to test them, nowadays, practitioners and developers need to mock the interaction since there is no available approach that supports multi platform/device testing.

In this chapter we present **`Kraken 2.0`**, an open source automated End-2-End (E2E) testing tool for defining and executing scenarios that involve inter-communication between two or more browsers or mobile devices. **`Kraken 2.0`** is an enhanced version of **Kraken v1.0** [315], which (i) includes fixes to issues, (ii) improves user experience, and (iii) has new features to support web-mobile interaction and fuzzing. In order to show **`Kraken 2.0`** capabilities, we used the tool with 10 combinations of web and mobile apps, in which we created and executed cross-device E2E test scenarios.

12.2 The `Kraken 2.0` Tool

In this section we describe the main changes that **Kraken** undergo when going from its v1.0 to v2.0. The current version (2.0) can be considered as a new tool since it was migrated from ruby to typescript, leading to a set of changes such as the replacement of base test automation library, enhancement of the tool usage by easing its installation and operation, and extension of available features to provide the users with a more robust testing tool. However, **`Kraken 2.0`** maintains the original goal of the tool, enabling cross-device interaction-based testing of apps. After going through a robust experimentation with **Kraken v1.0** we identified the need to generate a new version to mitigate some limitations: First, because Ruby relies on UNIX tools, **Kraken v1.0** was particularly difficult to configure and use on Windows OS. Second, **Kraken v1.0** only allowed inter-communication between Android devices. Third, Calabash[55] stopped receiving support from the industry, leading

Table 12.1. Capabilities comparison for the different tools

Feature	**Kraken1.0**	**Kraken2.0**
Parallel Execution	✓	✓
Signaling	✓	✓
Mobile device support	✓	✓
Web Browser support		✓
Cross-platform testing		✓
Generate random events over full- and part-of-screen	✓	✓
Kraken monkey	✓	✓
DataPool definition	✓	✓
Random Data generation		✓
Deployment to cloud (CI/CD)		✓
Extract snapshot of device		✓
Report generation	✓	✓
Test specification protocols	Gherkin	Gherkin, TS, JS
Step definition language	Ruby	TS, JS

to its deprecation since Android Oreo, which endangered also the compatibility of **Kraken v1.0**. Fourth, developers were interested in having more options for scenarios specification. To solve these issues we built the new version on top of WebdriverIO [316] and Appium [53], and combine them with the specification-by-example testing and signaling proposed by **Kraken v1.0** and Octopus[299]. Table 12.1 presents the differences between the two versions of the tool. **`Kraken 2.0`** includes additional capabilities such as: (i) tests execution with mobile and web interaction; (ii) new languages available for the specification of tests and steps; (iii) deployment in the cloud for CI/CD pipelines; (iv) on-demand extraction of UI snapshots of an android application at a given moment; (v) random data generation.

Fig. 12.1 presents the architecture of **`Kraken 2.0`**. It consists of several modules aimed at launching required devices/browsers, creating artifacts for each device/browser, supporting/orchestrating the signaling protocol, executing steps and generating reports. Components that support the new features depicted in Table 12.1 are highlighted by using a red border.

12.2.1 Test Files and Static Analyzer

Originally, **Kraken v1.0** was built using Calabash[55] as framework for the test definition and execution. Because of this, the tests were required to be defined following a BDT-style specification. Additionally, in order to define custom test steps, the user must write them using ruby. As result of the migration to Appium, **`Kraken 2.0`** now allows users to additionally define tests using “native" typescript files. An example of a test written using typescript is presented in Listing 12.1. With this new capability, the *Static Analyzer* component was also modified to be capable to check the compliance of the test files with the syntax provided by Appium and **`Kraken 2.0`**. It is worth noticing that Appium also supports BDT-style specifications, providing the users with backward compatibility of the existent test scripts created for *Kraken v1.0*.

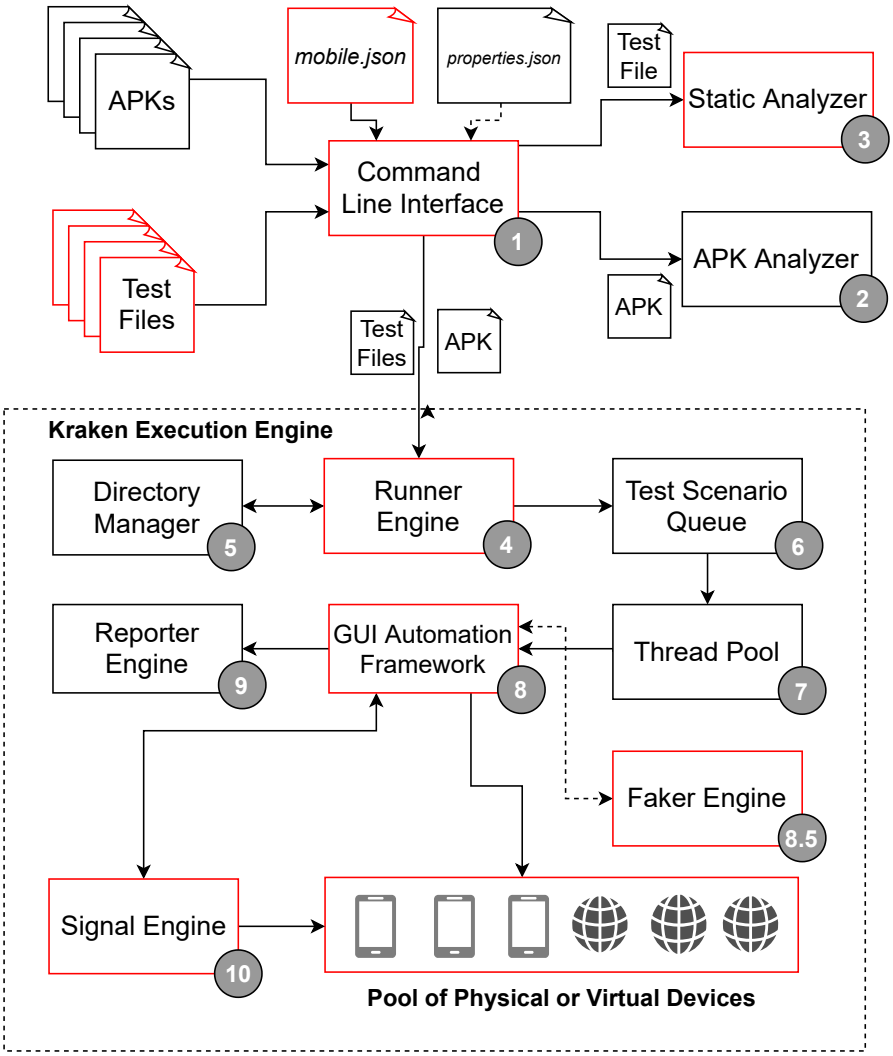


Figure 12.1. The **Kraken 2.0** architecture and workflow.

Listing 12.1. Kraken 2.0 scenario created with Typescript

```
let button = await client.$(‘android=new UiSelector().resourceId("es.usc.citius.servando.calendula:id/
    mi_button_skip")‘)
await button.click();//

let burgerMenu = await client.$(’//android.widget.ImageButton[@content-desc="Open"]’);
await burgerMenu.click();//

let settings = await client.$(‘android=new UiSelector().resourceId("es.usc.citius.servando.calendula:id
    /textView3")‘)
await settings.click();//

let back = await client.$(’//android.widget.ImageButton[@content-desc="Navigate up"]’)
await back.click();//
```

12.2.2 Cross-platform Interaction-based Multi-device testing

As part of the migration of the tool, **Kraken 2.0** now supports the definition and execution of tests over web browsers, since **Kraken v1.0** already performed signaling between devices, we extrapolated the signaling process to communicate two or more test execution processes regardless of the selected platforms. This was possible due to the capabilities provided by Appium, since the test definitions are platform-agnostic, and by using an internal framework **Kraken 2.0** translates the Selenium Webdriver commands into UIAutomator commands in the case of Android. Additionally, when it is used, Appium deploys an HTTP server that exposes a REST API that orchestrates the event transmission to the different devices and browsers.

12.2.3 Random Data Generation

In addition to the previously mentioned changes, **Kraken 2.0** allows for random data generation by using the FakerJS[317] library. This impacts the previous *(4) Runner Engine*, because when this mode is used within the test definition, a new task is created towards the *(8.5) Faker Engine* that is in charge of handling the library. This component generates the random data defined by the user and works as cache for the requested values in case a user decides to use them several times within the same test scenario. More detailed information regarding how to use this mode is presented in the following section.

12.3 Kraken 2.0 in Action

12.3.1 Console Interface

User interaction with **Kraken 2.0** is done via console commands. To start using **Kraken 2.0** it is only necessary to install the NPM package via the command `npm install kraken-node`; afterwards, users can (i) check if all prerequisites are fullfiled on the host machine, (ii) generate a test folder skeleton, and (iii) run the tests. Before running **Kraken 2.0** and depending on the type of devices that are going to be used, **Kraken 2.0** will require different prerequisites that are specified with the `kraken -node doctor` command; in addition, this command will display if the prerequisites are correctly configured as shown in Listing 12.2.

Listing 12.2. Kraken prerequisites

```
Checking dependencies...
Android SDK [Installed] (Required only for mobile testing - ANDROID_HOME)
Android AAPT [Installed] (Required only for Kraken's info command - ANDROID_HOME/build-tools)
Appium [Installed] (Required only for mobile testing)
Java [Installed] (JAVA_HOME)
Done.
```

After setting up the testing environment the first step to use **Kraken 2.0** is to generate a base project containing:

- A *.feature* test file (similar to the one in Listing 12.3)
- A *.json* file where the information of the APK under test will be specified
- A web subfolder containing a javascript file for the step definition
- A mobile subfolder with files required for **Kraken 2.0** execution

To do this, the user should call the `kraken-node gen` command.

Listing 12.3. Example test file

```
1  Feature: Example feature
2
3  @user1 @web
4  Scenario: As a first user I say hi to a second user
5  Given I navigate to page "https://www.google.com"
6  Then I send a signal to user 2 containing "hi"
7
8  @user2 @mobile
9  Scenario: As a second user I wait for user 1 to say hi
10 Given I wait for a signal containing "hi"
11 Then I wait
```

A test file should contain scenarios definition following a Gherkin+**Kraken 2.0** syntax. For example, the feature in Listing 12.3 contains two scenarios, one per each device involved in the test. As it can be seen in lines 3 and 8 of Listing 12.3, each scenario is linked to a specific user (*i.e.,* a device). The user tag follows a naming pattern: `@user(\d+)`. Also, each scenario can be executed on a different type of device such as a web browser or an Android device; to specify what type of device a scenario requires, the user should add the @web or @mobile tag.

Before executing the test and due to the usage of Appium as part of the architecture, **Kraken 2.0** requires the launcher activity name and package name of the APKs under test. This information must be specified in the *mobile.json* (Listing 12.4) file at the root directory.

Listing 12.4. mobile.json file

```
1  {
2          "type": "singular",
3          "apk_path": "<APK_PATH>",
4          "apk_package": "<APK_PACKAGE>",
5          "apk_launch_activity": "<APK_LAUNCH_ACTIVITY>"
6  }
```

In the case of specifying different APKs for each user then the tester should modify the *mobile.json* file to include the APK information of each user.

Listing 12.5. mobile.json file for multiple APKs

```
1  {
2          "type": "multiple",
3          "@user1": {
4                  "apk_path": "<APK_PATH>",
5                  "apk_package": "<APK_PACKAGE>",
6                  "apk_launch_activity": "<APK_LAUNCH_ACTIVITY>"
7          },
8          "@user2": {
9                  "apk_path": "<APK_PATH>",
10                 "apk_package": "<APK_PACKAGE>",
11                 "apk_launch_activity": "<APK_LAUNCH_ACTIVITY>"
12         }
13 }
```

When testing applications that are external to the organization or that the source code is not publicly available, it may happen that the APKs launcher activity name is not known for the tester, that is why **Kraken 2.0** offers the *apk-info* command that given an APK file path will retrieve this information by using the Android debug bridge (ADB). To run this command the tester should execute `kraken-node apk-file <APK_PATH>`, and **Kraken 2.0** will display information such as the one on

Listing 12.6

Listing 12.6. APK info retrieved by the `kraken-node apk-file command`

```
1 Launch activity: es.usc.citius.servando.calendula.activities.StartActivity
2 Package: es.usc.citius.servando.calendula
```

Once the user has accessed to this information, she must update the *.feature* file provided by **Kraken 2.0**, to correctly configure the test execution. It is worth remembering that **Kraken 2.0** uses the Gherkin + **Kraken 2.0** syntax, thus, the tester can create new steps and use already defined Appium and WebdriverIO helper functions. The **Kraken 2.0** specific steps are:

Send signal

Kraken 2.0 provides a step that sends a signal to a device. This step has two parameters: the user (*i.e.,* tag) that will receive the signal and the content of the signal. The structure of this step is:

Listing 12.7. **Kraken 2.0** send signal

```
1 I send a signal to user (\d+) containing "([^\"]*)"
```

Read signal

This step can be used in two different ways: first, to set the expected content (see line 1 in Listing 12.8); and third, to define the expected content and timeout (see line 2 in Listing 12.8)

Listing 12.8. **Kraken 2.0** wait signal

```
1 I wait for a signal containing "([^\"]*)"
2 I wait for a signal containing "([^\"]*)" for (\d+) seconds
```

Random Events Scenario

Kraken 2.0 provides a scenario step that allows users to generate GUI-based random inputs by following the syntax presented in Listing 12.9. This step can execute a given number of random events over the full screen (*line 1 in Listing 12.9*) or on a specific region defined by the user as a portion of the screen (*line 2 in Listing 12.9*).

Listing 12.9. **Kraken 2.0** random step

```
1 I start a monkey with (\d+) events
2 I start a monkey with (\d+) events from height (\d+)% to (\d+)% and width (\d+)% to (\d+)%
```

Kraken Monkey

Kraken 2.0 introduces another step that also sends and reads signals randomly, in addition to other input events (*e.g.,* text input, tap, etc.). The structure of this step is: `I start a kraken monkey with (\d+)events`

On-demand snapshot extraction

Kraken 2.0 allows users to extract a GUI snapshot of Android applications by using a predefined step. This feature can be used by practitioners to understand the UI components hierarchy (along with its content). The structure of this step is: `I save device snapshot in file with path "([^\"]*)"`

Properties file

Kraken 2.0 uses properties files to store data such as passwords or API keys that should be used in your test cases. A properties file should be a manually created JSON file with the structure presented in the following snippet:

```
{
    "@user1": {
        "PASSWORD": "test"
    }
}
```

Note that the key-value pairs are organized by user tags. In order to use the values defined in the properties file, the property should be invoked in a *.feature* file using "< ... >" as in the following: `I see the text "<PASSWORD>"`.

Finally, after writing the scenarios, the last step for using **Kraken 2.0** is running it with the `kraken-node run` command; it must be called in the folder that contains the *.feature* files. In case the user wants to use a property file, then the command must include the flag `--properties=<properties_path>`.

12.3.2 Fuzzing

Kraken v2.0 offers a fake string generator thanks to the library FakerJS; the list of supported faker types are: *Name*, *Number*, *Email*, *String*, *String Date*.

Kraken v2.0 keeps a record of every fake string generated; to this, each string must have an id. To generate a Faker string you need to follow the structure "$FAKERTYPEs_ID" as presented in the following example:

```
@user1
Scenario: As a user
    Then I enter text "$name_1" into field with id "view"
    Then I enter text "$date_1" into field with id "form_date"
```

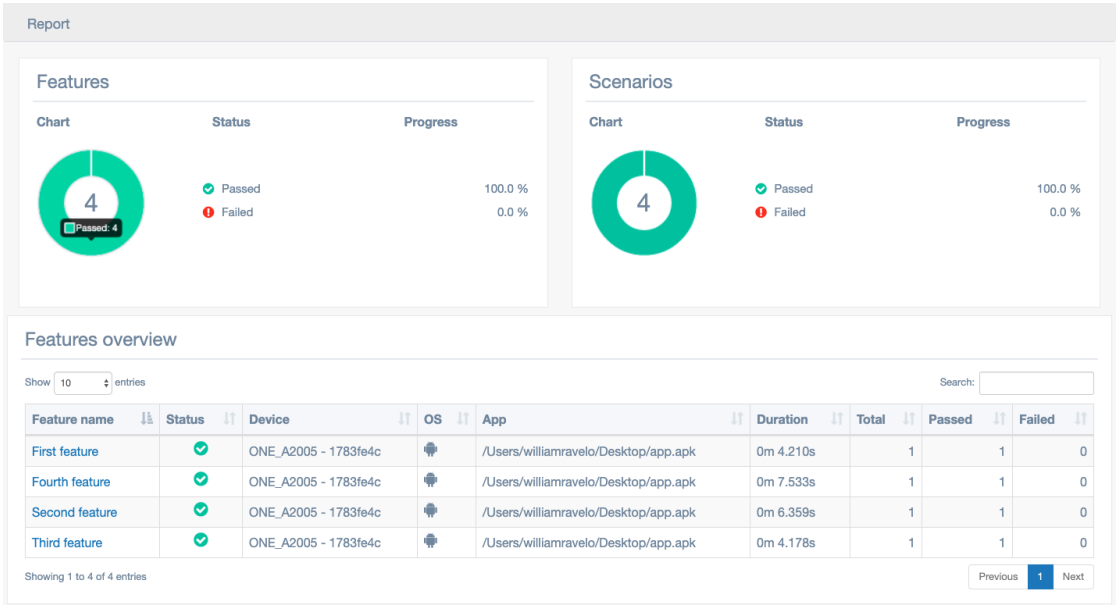
As mentioned before, **Kraken 2.0** keeps record of every string generated with an given id, this provides users with the possibility of reusing this string later in test scenarios. To reuse a string, the user needs to append a $ character to the fake string as in line 6 of the following example:

```
@user1
Scenario: As a user
        Given I wait
        Then I enter text "$name_1" into field with id "view"
        Then I press "add_button"
        Then I should see "$$name_1"
```

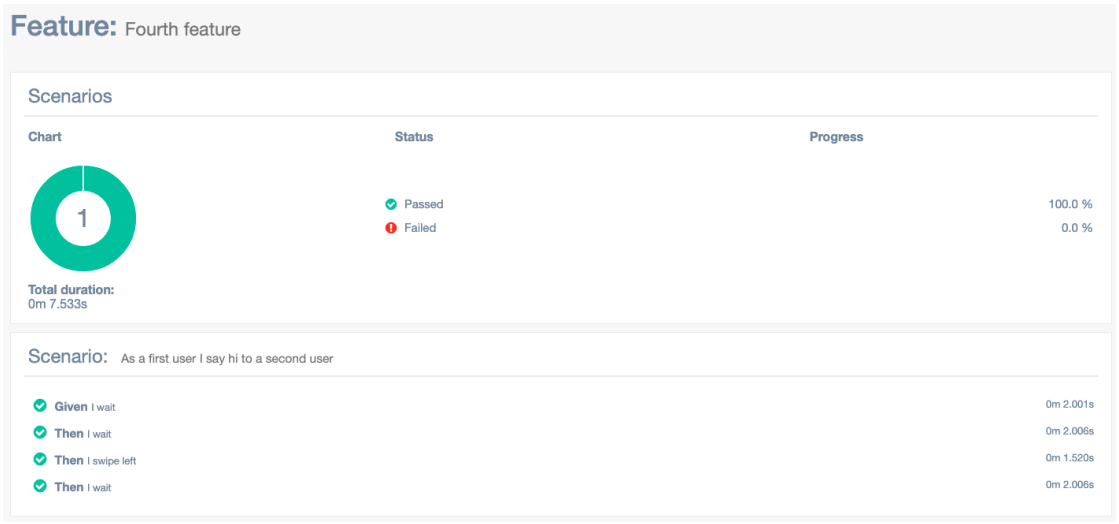
12.3.3 Web Reports

When finishing the execution of the test files, **Kraken 2.0** will generate a directory (in the current folder) that contains three reports: (i) general report, (ii) execution report by device, and (iii) execution detail by device.

(a) General report - sankey diagram



(b) Features report by device



(c) Feature details by device

Figure 12.2. Examples of reports generated by **`Kraken 2.0`**.

General report: In the top section of this report **Kraken 2.0** provides metadata of all the devices used in the execution of the test files. A Sankey diagram is included with the purpose of showing the actions flow of each device in every feature, as in a timeline (Fig. 12.2a). In this type of graph, the nodes will represent an action performed by a device; when the viewer hovers over the node it shows the executed step action as well as a thumbnail showing the GUI state after the action was completed. If the color of the node is green it represents that the action passed, otherwise, if the node is red then it indicates an error in the execution. Each path, composed of edges with the same color, represents a different device.

Execution report (by device): In this report, the tester can see the results of the tests, given a specific device. It includes the number of tests failed/passed, APK path, execution time, device name, and feature name (Fig. 12.2b).

Execution detail (by device): This final report presents the results of a specific test for a given device. The top section presents the number of steps passed and failed, in addition to the list of steps executed in either case. Finally, in the bottom section of the report, every step is presented with a screenshot of the device after the action was executed (Fig. 12.2c).

12.4 Usage Examples

To show **Kraken 2.0** capabilities we created and executed test scenarios for 10 different apps that involved the interaction of two or more users/applications and also included web interaction. The capability of the framework to coordinate the signaling protocol between two or more devices running the same application is illustrated with (i) social and messaging applications such as AskFM or Facebook were one user shares information and then other users can access it; (ii) trivia games such as Kahoot where two or more users compete by answering a list of questions; and (iii) delivery apps such as Pibox were a delivery company requests a delivery guy for sending a package. Finally, we used **Kraken 2.0** to test scenarios that involved the coordination of two or more services by testing a scenario where one user plays a song with an audio streaming service app such as Spotify, and then another user listens to the song and recognizes it with a song identifier app such as Shazam.

One of the examples is presented as follows: Listing 12.10 presents the steps for a test of an email app from a mobile app point of view (`user 1`). From lines 4 to 10, the user logins into the app, then in line 12 it starts composing and email, then from line 14 to 16 it fills out the email form, and finally the form is submitted (line 17). It is worth noticing that at line 20 the test sends a signal informing to `user 2` that the email was sent.

Complementary, Listing 12.11 presents the steps to check the email in a browser (`user 2`). For this, in line 3 the test starts by visiting the email app home page, then from line 4 to 7 the scenario fills out the login form and enters the app. At this point, since no signaling related step has been executed yet, both scenarios will be executed in parallel, nevertheless, in line 8 of Listing 12.11, the scenario specifies that `user 2` needs to wait for a signal with the label "`email-sent`", which was sent by `user 1` in line 20 of Listing 12.10. This allows the test execution to understand where it should wait for the other users. Finally, at line 11 it checks if there is a new email with the subject used in Listing 12.10.

Listing 12.10. Kraken 2.0 mobile scenario for email app

```
@user1 @mobile
Scenario: Send email from mobile app
    Given I wait
    When I click email input
    When I enter text "kraken@gmail.com"
    And I wait
    And I click password input
    And I enter text "kraken2020@"
    And I wait
    And I click on button with text "Login"
    And I wait
    And I click on button with text "Compose"
    And I wait
    And I enter text "krakentest2020@gmail.com" into field with id "destination_email"
    And I enter text "Kraken test subject" into field with id "compose_subject_field"
    And I click on screen 50% from the left and 50% from the top
    And I enter text "My message"
    And I press view with id "action_compose_send"
    And I wait
    Then I send a signal to user 2 containing "email-sent"
```

Listing 12.11. Kraken 2.0 web scenario for email app

```
@user2 @web
Scenario: Check email from browser
    Given I navigate to page "https://gmail.com"
    When I enter "krakentest2020@gmail.com" into input field having id "identifierId"
    And I click on element having id "identifiedNext"
    And I enter "kraken2020@" into input field having css selector "#password > div.aCsJod.oJeWuf > div
          > div.Xb9hP > input"
    And I click on element having id "passwordNext"
    And I wait for a signal containing "email-sent" for 60 seconds
    And I navigate to page "https://gmail.com"
    And I wait for 5 seconds
    Then I should see text "Kraken test subject"
```

13

Using DRL for Automated Testing of Android Apps

Mobile app testing is a time-consuming and critical task necessary to guarantee high-quality applications. Due to the constant market pressure on developers, new efforts have been made to exploit the capabilities of new technologies to automate and improve the testing process. Reinforcement Learning (RL), for example, is a machine learning technique that uses positive and negative rewards to find an optimal solution. Specifically for mobile apps, RL approaches can use different aspects of the mobile environment, such as fault-triggering, code coverage, and GUI changes, to guide the learning process towards enhanced exploration and testing of apps. However, state-of-the-art public approaches have focused on testing the individual impact of the aforementioned aspects on the learning process. In this article, we present `MutAPK` , a DeepRL approach that uses a compound reward function to enhance the exploration/testing of mobile apps. We conducted a study to compare the performance of `MutAPK` using two different algorithms with baseline approaches (*i.e.,* ARES and Monkey) and study the execution results to identify orthogonality. Our results show an increase of approximately 10% in the accumulated coverage achieved after exploration using SAC algorithm, while maintaining a similar behaviour of baseline approaches when using the DDPG algorithm.

Structure of the Chapter

- Section 13.1 provides motivation for this chapter.
- Section 13.2 contextualizes the reader with instrumental background concepts.
- Section 13.3 provides motivation for this chapter.
- Section 13.4 presents the design of our study, as well as the data extraction procedure and analysis methodology.
- Section 13.5 discusses our results and findings.
- Section 13.6 presents the threats that affect the validity of our work.

13.1 Introduction

Mobile devices have a important role in daily life of people, enabling the access to services and providing capabilities that users were used to have in computers. According to recent studies[318], Android and iOS are in the top 3 OS's used by people around the world. This increasing usage of devices have come along with a continous increase in the amount of apps available in marketplaces.

Unfortunatelly for the developers, this increase in the amount of apps represent also an increase in the amount of similar apps that could replace their position within users devices. Therefore, developers need to ensure high quality applications with frequent releases to keep and increase the amount of users. This is achieved by the improvement of the quality assurance process that starts with the feature definition, constant feedback analysis and testing of the app.

However, a recent studies has shown that only 14% of over 600 open-source Android applications contain test cases and circa 9% have executable test cases with coverage over 40% [6]. Showing that despite testing is a critical part of the development process, not all the projects use and apply them. Additionally, previous studies have shown that test are executed fequently by humans [150].

To enhance the testing capabilities, different strategies and approaches has been proposed and studied, such as random testing strategies [61, 319] that exercise the app under test by producing pseudo-random events; model-based strategies [320, 321] extract test cases from navigation models built employing static or dynamic analysis. Some of the proposed approaches have focused on testing the Graphical User Interface (GUI) [80, 320, 322, 323], trying to simulate a real user, that interacts with the app. Despite all the efforts for automating the testing of mobile app, there is still oportunities to improve the testing process. As a solution to this, new approaches has been proposed using reinforcement learning (RL), a ML technique that uses reward-based learning to optimize the generation of events. This technique has potential within the mobile app environment, due to the event-based mechanisms used within apps.

Nevertheless, recently proposed approaches use this technique by selecting a single metrics as input. Common examples of this metrics are: amount of issues/errors identified, GUI change rate after the execution of an event, and amount of new elements in the GUI after the execution of an event. As part of the evaluation of this approaches, the coverage of the tools are computed and presented along with the studies results. However, to the best of our knowledge, there is no approach that uses a compound reward function and additionally uses the coverage as part of metrics to oriented the learning process.

Based on this, in this article we present a study in which we evaluate the performance of an DeepRL approach that uses a compound reward function based on the amount of identified errors, the achieved coverage and the GUI change rate. We create a tool called **AgentDroid**, that was executed over 11 open source android apps. We compared the performance of this approach against state-of-the-art solutions and analyze the results in terms of accumulated coverage and amount of identified errors. Finally, we analyzed the results to identify if there were orthogonal results between the execution of the tools.

13.2 Mobile App Testing and Reinforcement Learning

13.2.1 Reinforcement Learning

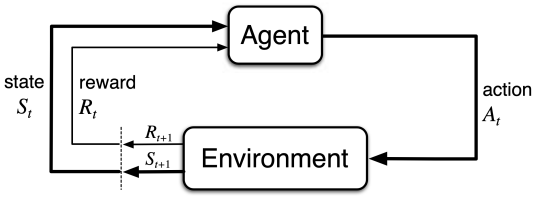


Figure 13.1. Agent-environment interaction in an MDP model in time step t [1]

Reinforcement Learning (RL) is a branch of Machine Learning that draws from psychology and neuroscience theories of learning [1]. It basically consists of learning by interaction to achieve a goal. The learner is referred to as the ***agent***, and what it interacts with is the ***environment***. Taking chess as an example, the player (*agent*) interacts with the board and pieces (*environment*) to win the game (*goal*). Formally, the reinforcement learning problem is modeled as a finite Markov Decision Process (MDP). An MDP model has four basic components: a set of states $\mathscr{S}$, a set of actions $\mathscr{A}$, a set of rewards $\mathscr{R} \in \mathbb{R}$ (or *reward function*), and the environment dynamics represented as the probabilities of a state and reward occurring given a particular preceding state and action:

$$p(s',r|s,a) \doteq P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{13.1}$$

Here, the *agent* and *environment* interact at discrete time steps, as shown in Figure 13.1. Specifically, at a time step, t, the *agent* observes the *environment* represented as a ***state*** $S_t \in \mathscr{S}$, *e.g.*, a chess board arrangement. With that information, the agent chooses and performs an ***action*** A_t that can be applied on the *environment*, such as moving a certain chess piece. This leads the *environment* changing to a new *state* ($S_{t+1} \in \mathscr{S}$) in the next time step $t+1$ and also responding with a ***reward*** $R_{t+1} \in \mathscr{R}$ [1].

Having said that, the goal of the *agent* is to maximize the cumulative *reward* received by the *environment* (not just the immediate reward at each time step). Thus, the *reward signal* or ***reward function*** must be aligned with the final objective is, not how it should be achieved. For instance, following the chess example, an agent should not be rewarded for taking the opponent's pieces but rather for winning the game. This cumulative reward is often represented as the *return*, which is defined as the sum or discounted sum (for continuing tasks) of the rewards in each time step. Because not all the steps implies learning, tasks can be divided into *episodic tasks* and *continuing tasks*. The former can be divided into sub-sequences or ***episodes*** that end in a particular *terminal state*, such as the task of playing chess, while the latter goes on indefinitely, such as the task of controlling a continuous reactor.

In essence, the learning process consists of mapping situations to actions. This mapping is referred to as the ***policy***, which is the center of the learning process as it defines an agent's

behavior. A *policy* π defines the probabilities of selecting each action given a state (equation 13.2), and the learning method defines how the agent's policy is updated through experience. Policies could be deterministic, *e.g.,* the probability of choosing an action is 1 for the preferred action and 0 for the other possible actions, or non-deterministic.

$$\pi(a|s) \doteq P\{A_t = a|S_t = s\} \tag{13.2}$$

There are two main approaches to implement RL: *tabular* and *deep RL*. The most straightforward consists of representing the possible states and actions as tables, *i.e.,* the **tabular** approach. The second approach, known as **Deep Reinforcement Learning** (DRL), uses deep neural networks to choose agent actions and evaluate states. Also, reinforcement learning methods are commonly classified as ***action-value methods*** and ***policy gradient methods***. On the one hand, *action-value methods* depend on the estimation of a ***value function*** used to evaluate and update policies seeking to maximize the value of all states. In this sense, a state-value function $v_\pi(s)$ basically indicates how good a state is. This is formally defined as the expected return when starting in state s and following a policy π. Also, the estimation of an action-value function comes in handy. Similar to $v_\pi(s)$, an action-value function $q_\pi(s,a)$ is defined as the expected return of taking action a when starting in state s and afterwards following policy π. Common examples of these methods are SARSA and Q-Learning[1]. On the other hand, in *policy gradient methods*, the policy update does not rely on the estimation of a value function. Instead, policies are parameterized ($\pi(a|s,\boldsymbol{\theta}), \boldsymbol{\theta} \in \mathbb{R}^d$) and updated with the objective of maximizing their performance, represented as a function $J(\boldsymbol{\theta})$, by using an approximation of gradient ascent. Common examples of these methods are the *actor-critic methods*, which learn both a policy (actor) and a parameterized value function (critic), like SAC and DDPG.

Furthermore, a common challenge that RL methods should treat is the *exploration-exploitation dilemma* (*i.e.,* use what the agent has learned or continue exploring new actions). This arises since, to maximize the reward, the agent should take actions that it has learned to be worthwhile, but to learn worthwhile actions, it should try yet unknown actions. A common way to approach this challenge in some action-value methods (though not the only one) is using an $\epsilon - greedy$ action selection, in which a random action is chosen with probability ϵ while the greedy action for estimated actions' values is chosen with probability $1-\epsilon$.

13.2.2 Automated GUI testing with Reinforcement Learning

Both tabular and deep RL have been applied to mobile app testing. Regarding the tabular approaches, [304, 305, 306, 309, 312] have explored using tabular Q-learning to test Android applications, while [303] and [308] have used SARSA and Double Q-Learning algorithms. In [305, 306, 309, 312], the states were modeled by extracting UI snapshots dynamically, meaning the states were modeled either by activity names and UI events available on each screen, the XML layout that allowed agents to discover GUI components, or combined states of components. In general, actions were defined mainly by event type, GUI component/widget, and other varying attributes such as on-screen coordinates [305] or text held by widget [306]. The possible event types were quite varied between the approaches, ranging from

simple GUI events such as taps and scroll gestures to more elaborate events such as physical and virtual device button presses or system events. Regarding the reward functions, most of the approaches uses the amount of available events in the GUI as parameter, leading to the exploration of new views and activities to increase this metric. When compared to other techniques, despite being relatively lightweight, the tabular RL approaches have achieved similar results, and in some cases better, for instance (i) [303, 304, 305, 306, 308, 309, 312] versus random testing, (ii) [303, 304, 309, 312] versus search-based testing , (iii) and model-based approaches such as [304, 305] versus Dynodroid, [309, 312] vs Stoat, and [304, 305] versus PUMA.

Furthermore, other studies have proposed Deep Reinforcement Learning (DRL) approaches. For instance, Vuong *et al.* [307] expanded on their previous work [305] by implementing a Deep Q-Network that uses the change in the amount of available GUI events as metric for the reward function. Collins *et al.* [313], proposed DeepGUIT, a Deep Q-Learning (DQL) approach that using trial-and-error and focusing in avoid event repetition generates automatically a preliminar set of test to reduce the load in the testing procees. Romdhana *et al.* [311] present ARES, a tool that evaluates three policy gradient algorithms (DDPG, TD3, and SAC) These DRL approaches formulate MDPs similar to the tabular solutions, however, DRL outperformed tabular the Q-learning approaches in most scenarios, as well as other automated test generation tools.

In general, most of the aforementioned studies (tabular and DRL) used different levels of code coverage as a metric for the evaluation (*e.g.,* statement, activity, line, block, or method coverage) [303, 304, 305, 306, 307, 309, 311, 312, 313]. Excluding [306], these studies also included fault detection of crashes or AUT package exceptions, and just a few of them had speed or number of events as an efficiency objective [307, 309, 311, 312]. **However, when looking into the details of the reward functions, none of the studies combined both coverage, and fault detection capabilities**. Different types of reward functions have been used such as coverage improvement, activities exploration, events execution, unfamiliar features execution.

On the other hand, a high proportion of the models use $\epsilon - greedy$ for the exploration-exploitation, changing the way it decreases. In some cases, *e.g.,* [304], ϵ is multiplied by a factor after each iteration (e.g., $\epsilon \times 0.995$) to exponentially decrease the exploration tendency of the agent. The motivation for this is to quickly start exploiting the states space while still having a small chance to explore after being trained over all AUTs, or to had a uniform decrease during the first hundred episodes [303, 305, 309, 311, 312, 313].

Alternative tools have implemented RL algorithms to explore Android applications, not necessarily for testing purposes. Examples of this are DeepApp, a DRL tool presented in [324] that attempts to predict application usage. Eskonen et al. [310] and Toyama et al. [325] developed image-based DRL tools; they differ from other tools by using GUI screenshots as the source of information instead of the logs and UI layout files: [310] uses the A3C algorithm for testing purposes, and [325] proposes AndroidEnv, a Python library focusing on applying DRL to execute tasks.

Lately, more studies have focused on problems in which competition or collaboration between multiple actors arises, such as multiplayer games, self-driving cars, package deliv-

eries, analysis of social dilemmas, and many more [326, 327]. These problems have led to the scaling of RL problems to multiple agents: Multi-Agent Reinforcement Learning (MARL), however, **while these approaches have proven to be powerful, no applications of MARL in testing have been implemented to this date.**

13.3 Proposed Approach

In this section, we describe **AgentDroid**, an RL approach for exploring and testing Android applications using APKs as input. **AgentDroid** was designed to prioritize maintainability and extensibility. In contrast to other approaches[80, 311, 328], **AgentDroid** uses a modular architecture that allows the decoupling of the learning environment from end devices, instrumentation methods, etc.

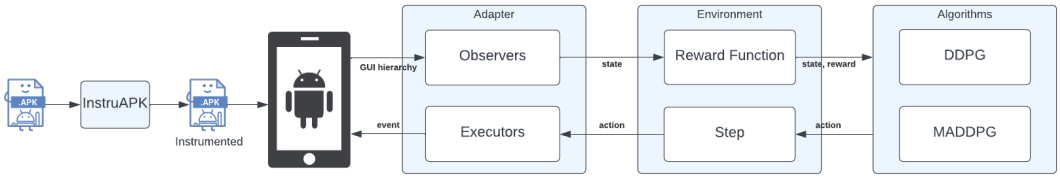


Figure 13.2. **AgentDroid** Workflow Overview

Fig. 13.2 presents an overview of **AgentDroid**'s workflow. It consists of 3 main phases: (i) app preprocessing, (ii) Environment setup, and (iii) Approach Execution. First, **AgentDroid** preprocess the APK to set it up for its execution and extracts required information for its usage (*i.e.,* Activity list, package, and main activity); this is done using `androguard` Python library. Second, based on the user-provided parameters and the previously extracted information, **AgentDroid** creates a set of adapters that (i) enable the interaction with the devices/emulators and (ii) provide mechanisms to extract and process information generated by the devices and as result of the exploration/test. Then, the resulting adapters are used to initialize the learning environment, specifically a custom OpenAI Gym Environment[329]. The selection of this environment allows us to formalize an observable Markov Decision Process for an episodic task. Lastly, **AgentDroid** creates an instance of the desired algorithm's wrapper class and generates a thread-based execution process that performs the exploration/test.

Following subsections present the definition of *States (S)*, *Actions (A)* and *Reward Function (RF)*. Then, we present the definition of the *Environment (E)* based on the aforementioned components.

13.3.1 State

We define our states using a layout-based representation. This decision is based on the additional effort associated to reduce the search space when using a image-based representation[310,

Table 13.1. Summary of RL models applied to Android testing

Tool/study	Type	Type of actions	Reward function (R)	Policy	Number of apps tested	Tools compared	Evaluated objectives
AimDroid [303]	tabular	• UI Events: Touch, Flip, Trackball, Drag, Keyboard, type	Discrete Rewards for: Activity Transition, Crash Penalizes for: App Transition	Outer states: explored with BFS Inner states: explored with RL using SARSA	50	Sapienz, Monkey	**Code coverage:** Instruction, Method, Activity **Faults:** Crashes
QBE [304]	tabular	• UI Events: click, long click, swipe • Device events: menu button, back button • System Events: connectivity toggle, bluetooth toggle, location mode switch, airplane mode, screen on/off • Other: reinitialization (reinstall and start)	2 reward functions Rewards for: Activity Transition, Crash	Q-Learning	100	Monkey, PUMA, SwiftHand, Sapienz, Dynodroid, A3E (Automatic Android App Explorer), AndroFrame (propietary) (Random and depth-first exploration)	**Code Coverage:** Activity, Instruction **Faults:** Crashes
Vuong [305]	tabular	• UI Events: Click, Long-click, Check, Text Input, Scroll • Device events: menu button, back button	Continuous Rewards for: degree of change (relative num of new events), inverse of event execution frequency	Q-Learning –> Behavioral Model	7	Android Monkey, Dynodroid, PUMA	**Coverage:** Line **Faults:** Analyzed, but not compared to other tools
Adamo [306]	tabular	• UI Events: tapping a widget, text input, etc • Device Events: home button, back button	Discrete Rewards for: inverse of event execution frequency	Q-learning	8	random test generation	**Coverage:** Block
QDroid [307]	deep	• UI Events: Click, Long-click, Scroll up, Scroll down, Swipe left, Swipe right, Text Input • Device Events: menu button, back button	Continuous Rewards for: degree of change (relative num of new events), inverse of event execution frequency	Deep Q-Network –> Behavioral Model	12	ClassicQ (vuong2018),Dynodroid, PUMA, A3E, GuiRipper	**Coverage**: Line, Method **Faults** **Efficiency:** Speed
FARLEAD-Android [308]	tabular	• UI Events: click, text • Device events: back button	LTL-based Reward Shaping Reward Prediction Rewards for: satisfying test case specification Penalizes for: not satisfying specification in X steps	Double Q-Learning	2	Random, Monkey, Q-Learning based exploration optimized for activity coverage (QBEa)	To generate satisfying tests
Pan [309]	tabular	• UI Events (not specified) • Device events (not specified) • System events: user actions (e.g., screen rotation, phone calls), broadcast messages(e.g. switch into or out of airplane mode), application-specific events which can be extracted from Android manifest files	Discrete Determined by Neural Network Rewards for: leading to unfamiliar functionalities Penalizes for: similarity of states	Q-Learning Curiosity driven Q initialization	50	Android Monkey, Stoat, Sapienz	**Coverage:** Instruction **Faults**: Crashes, AUT package exceptions **Efficiency**: Speed, Num of events
Eskonen [310]	deep	Only interacts with links, buttons and text boxes • UI Events: clicking a location, typing text	Discrete Rewards for: finding something new in the GUI	A3C algorithm	1	Random	Maximize the number of unique GUI states
ARES [311]	deep	• UI Events (not specified) • System events: toggle internet connection, rotate screen	Discrete Rewards for: visiting new activities, fault detection Penalizes for: strongly for leaving AUT, slightly otherwise	SAC Also implements Q-Learning, DDPG, TD3	41-68	Android Monkey, Sapienz, TimeMachine, Q-Testing	**Coverage:** Line (source code level) **Faults:** Crashes **Efficiency:** Speed
DroidbotX [312]	tabular	• UI Events: click, long-click, scroll, swipe left/right/up/down, input text data • Device events: menu button, back button, home button	Every state and action reward an agent with +1. Policy is not updated based on reward function, rather by priority	Q-Learning	30	Android Monkey, Humanoid, Stoat, Sapienz, Droidbot	**Coverage:** Instruction, Method, Activity **Faults:** Crashes **Efficiency:** Speed, Num of events
DeepGUIT [313]	deep	• UI Events: click, select, long click, text input, scroll, check • Device events: home, back, volume control, menu • System events: rotate	Continuous Rewards for: improving coverage, visiting new activities Small reward for: no change in state, leaving AUT	Deep Q-Learning	15	Monkey, Q-Testing	**Code coverage:** Instruction, Branch, Line, Method **Faults:** Failures, Crashes

NA: Not applicable. NS: Not specified.

325], such as reducing the image's resolution [310] and possibly losing valuable information in the process.

Hence, states are based on the current GUI hierarchy of the AUT extracted in XML format using the Android testing framework `uiautomator`. This hierarchy is transformed into a vector representation compatible with the learning algorithm similar to the one used in ARES [306]. Concretely, a state $s \in S$ consists of appending a one-hot-encoding vector representing the current activity and a vector encoding of its widgets (Eqs. 13.3 and 13.4). In order to encode the activity, the APK is statically explored beforehand, and all activities identified are added into the first vector. On the other hand, widget identifiers are added to a list when they are first found during dinamic exploration. This list is updated in each step, and the vector encoding is generated according to Eq. 13.5. This shows an improvement over ARES, as the model is able to distinguish between states in which the same widgets are enabled or disabled.

$$S = \{s | s = (a_0, a_1, ..., a_n, w_0, w_1, ..., w_m)\} \tag{13.3}$$

$$a_i = \begin{cases} 1 & \text{If activity } i \text{ is the current activity} \\ 0 & \text{Otherwise} \end{cases} \tag{13.4}$$

$$w_i = \begin{cases} 1 & \text{If widget } i \text{ is present but has no interaction} \\ 2 & \text{If widget } i \text{ is present, has disabled interaction} \\ 3 & \text{If widget } i \text{ is present, has enabled interaction} \\ 4 & \text{If widget } i \text{ is present, has enabled interaction, is text field} \\ 0 & \text{Otherwise} \end{cases} \tag{13.5}$$

13.3.2 Actions

To define the actions available for execution by our approach, we rely on "user interaction events" as a guide. Our approach will only execute events users might perform when using/-exploring the app. To achieve this, we process the XML extracted previously and identify the elements that have properties that enable the interaction by users. Examples of the properties mentioned above are: clickable, long clickable, and writable. Therefore, for each *State (s)*, we identify the widgets and events available as a set of tuples, where the first element of each tuple is the *Widget (w_i)* and the second element the *Event (e)* that can be performed over w_i.

13.3.3 Reward function

AgentDroid uses a reward function that combines code coverage, fault detection, and state change (see Equation 13.6), where $\Delta(coverage)$ measures the change in the code coverage after an action is executed by the agent; $\Delta(state)$ measure the change in the UI; and $|error|$ is the number of triggered errors.

$$R_t = \Delta(coverage) + \Delta(state) + |error| \tag{13.6}$$

Coverage Computation

As it was mentioned before, our approach is based on the usage of an APK as input and the first step performed is the instrumentation task that decodes the APK into SMALI files (intermediate representation available to process Android apps), and adds a log statement at the start of each method. This log adding process is performed only on methods belonging to the app code, therefore, methods belonging to libraries are not instrumented. Since this instrumentation process requires our approach to visit the whole app code representation, at the end of this process we know the exact amount and location of methods within the APK code. It is worth noticing that this process requires the computation of the code AST, which gives us the opportunity of extracting the call-graph of the app and allows us to identify the dead code. With the call-graph, **AgentDroid** skips dead/unreachable methods from the coverage computation.

Once **AgentDroid** have gone through the instrumentation process, the app will then log into the Logcat a string that contains a representation of the called method. Based on this log activity, we can then recognize the called methods once an action has been triggered. In order to measure the impact of an agent action in terms of coverage, **AgentDroid** keeps a register of the methods that have been called during the whole exploration/test of the app and then we measure the change of method coverage ($\Delta(coverage)$) after each action. Since **AgentDroid** knows the complete amount of methods within the app, $\Delta(coverage)$ is computed as a ratio of the number of unique methods called as result of an action executed by the agent.

State Change

In order to measure the effect of executing an action, **AgentDroid** extracts the XML representation of the states before and after the action execution. Then it computes the Jaro similarity [330] using the XMLs as strings. The Jaro similarity returns a value between 0 -no similarity- and 1 -identical strings-. In order to select this measure (*i.e.,* Jara similarity), we extracted manually a set of screenshots of the apps under study an compute the difference of these using different measures from past studies: Levenshtein distance, Hamming distance, Damerau-Levenshtein distance, Cosine distance, Jaro-Winkler similarity and Cosine similarity. After analyzing the results, we identified that the Jaro similarity, first, provide us a value between 0 and 1, allowing us to use the final results directly as parameter of our reward function, and second, provided the best fidelity of results. Equation 13.7 presents the Jaro similarity function, where m is the number of matching characters and t is half the number of transpositions.

$$d_j = \begin{cases} 0 & \text{If m} = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{Otherwise} \end{cases} \tag{13.7}$$

Triggered Errors

AgentDroid detects the triggered errors by processing the Logcat console from the Android OS of the testing device. To this, **AgentDroid** filters all the messages (in the Logcat) related to the application under test, by using the application `PID` (Process ID), then, from that initial list all the messages tagged as Errors or that contain the words "`Error`," "`Exception`," or "`FATAL EXCEPTION`" are selected. Since the Logcat presents the errors in more than 1 line, once we identify an error, **AgentDroid** analyzes the following lines to gather all the information logged as part of the error and make sure those are from the same source. Once the error has been processed, a Hash value is computed with the error header and message tuple; this hash allows **AgentDroid** to create an unique identifier for each new error in the console. The amount of identified errors is then used as the $|error|$ component of the reward function.

13.3.4 Environment

Given we are using a custom Gym environment, we must define: (i) the action and state spaces,(ii) the environment's behaviour at each step, and (iii) the environment's reset policy after each episode. Algorithm 1 presents the pseudo-algorithm of the stepping function.

ALGORITHM 1: Stepping function of our custom Gym environment

```
def step(action):
    executeAction(action);
    observation = observeEnvironment();
    reward = calculateReward(lastState, currentState);
    done = shouldEndEpisode();
    return observation, reward, done;
```

Additionally to perform an action, the *executeAction* function takes care of the limit of actions the agent has performed outside the application under test (AUT). The number of actions performed outside is kept in an attribute named `state`. If this number exceeds the limit, then it uses the device adapter to relaunch the AUT.

The *observeEnvironment* function is also at the beginning of each episode. This function is mainly in charge of calculating the state vector that is returned at each step, but it also updates helper structures like the `state attribute` and the list containing the possible actions. First, in order to calculate the vector describing the state, this function uses the GUI adapter to read the device's GUI. With this information, it proceeds to update the list of widget identifiers that are kept to calculate the state vector following the structure presented in previous sections. Additionally, the environment observation function is in charge of reading coverage reports using the coverage processor and reading the log's faults using the log reader.

The reward calculation computes the values of coverage increase, number of faults identified, and state difference according to the defined equation previously. Finally, **AgentDroid**

checks whether the episode should end or a new action must be performed. The episode is finalized when the time step or the amount of events/actions reaches the established maximum episode length.

13.4 Study Design

The *goal* of this study is to assess the performance of the proposed approach and identify the differences in terms of coverage and fault-detection capabilities, when compared to competitive baselines. This study is conducted from the *perspective* of researchers interested in improving current tools and approaches for automated testing of Android apps. As for the context, we focus on open source Android apps and used two existing approaches, ARES [311] and Monkey [61] as baselines. In particular, with this study we aim at answering the following research questions (RQs):

- ***RQ*$_1$*:*** *What is the performance of* ***AgentDroid*** *when compared with baselines approaches?* Because the RL model used in our approach has some differences when compared to state-of-the-art tools (*e.g.,* ARES), we want to evaluate whether those differences (*e.g.,* reward function) lead to a better performance in terms of code coverage and detected faults.

- ***RQ*$_2$*:*** *What are the characteristics of the faults detect and coverage achieved by our approach?* With the RQ we expect to understand if there are orthogonal or complementary results across the three analyzed approaches (*i.e.,* **AgentDroid**, ARES, and Monkey).

13.4.1 Context of the study

In order to answer the proposed research questions, we selected a dataset of 14 Android apps as follows. First, we selected a list of Android Apps available in the FDroid repository [290]. We selected apps from FDroid since it allows to download APK files directly without having a Google Play account. In addition, we filtered apps based on their level of interaction complexity, *i.e.,* we discarded apps that based on the app screenshots and description were trivial or did not provide more than 5 views. This decision was based on the fact that our objective was to test the capabilities of the analyzed tools to explore apps as thoroughly as possible. Also we wanted to avoid scenarios in which high coverage is achieved but because of the apps simplicity.

After conducting the aforementioned process, we had 42 apps in our dataset. The second step to filter this apps was based on the usage of the different tools required for our study. As it was mentioned in Section 13.3, in order to compute the coverage of the performed test we had to instrument the apps to identify the called methods during apps' exploration. In order to do this, we used a tool to modify the APKs of the apps to insert log statements at the start of each method; since we were working with APKs directly, there were 12 apps that after its modification could not be launched in an emulator, therefore, those apps were excluded

Table 13.2. List of analyzed applications

App Name	Date	Version	Package Name	Amount Methods	Category
Sipdroid	22/Sept/2022	4.4	org.sipdroid.sipua	599	Communication
APG	22/Sept/2022	1.1.1	org.thialfihar.android.apg	1512	Communication
Budget	22/Sept/2022	4.4	com.notriddle.budget	399	FInance
Twik	22/Sept/2022	1.3.10	com.reddyetwo.hashmypass.app	501	Communication
Apollo	22/Sept/2022	1.1	com.andrew.apollo	1777	Music
Mileage	22/Sept/2022	3.1.1	com.evancharlton.mileage	1141	Transportation
Radio Reddit	22/Sept/2022	0.7	com.radioreddit.android	296	Communication
IFixit	21/Sept/2022	2.4.1	com.dozuki.ifixit	1821	Books and Reference
HydroMemo	19/Sept/2022	1.0.12	de.boesling.hydromemo	80	Health & Fitness
Hydrate	19/Sept/2022	1.5	com.frankcalise.h2droid	168	Health & Fitness
TomDroid Notes	19/Sept/2022	0.7.5	org.tomdroid	745	Productivity

from the data set. The main reason for these errors was based on the usage of a library that restricts its functionality to apps build using a targetSDk below than the Android API 28.

Third, we executed **AgentDroid** and ARES using the remaining 30 apps; 19 of those apps exhibited issues when executed with the tools, *i.e.,* the errors do not show up when executed manually. This represents a shortfall for our approach and it is discussed in Section 13.6. As a result, our dataset contains 11 open source Android apps, available at FDroid that run with **AgentDroid**, ARES[311] and the Monkey Exerciser. The characteristics of the apps are listed in Table 13.2.

13.4.2 Analysis method

To answer $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ we selected ARES [311] and Monkey [61] as baseline approaches. In the case of the the UI/APP Exerciser Monkey, it is a random event generation tool, created by Google, which has been widely used in the testing community as baseline. Some studies have shown that Monkey is able to outperform non-trivial and more complex approaches in terms of coverage and fault-detection capabilities.

Regarding ARES, we selected it from the RL-based existing approaches (see Section 13.2.2), since it is one of the first publicly available approaches to use Deep Reinforcement Learning to test Android apps; moreover, according to a previous study [311], the Deep RL approach proposed within ARES achieves higher coverage and fault revelation than other similar approaches such as Sapienz [80], TimeMachine[328], Monkey[61]. Also, thinking on the DRL model implemented in ARES, **AgentDroid** has several differences such as a custom definition of the MDP, usage of a compound reward function, the evaluation process of achieved coverage, and the identification of similar error. All in all, ARES and Monkey, are two competitive baselines that allowed us to analyze **AgentDroid** capabilities in a comparative way.

ARES uses two RL algorithms (*i.e.,* DDPG and SAC), therefore, in order to provide a comprehensive comparison, we conducted our experiments using two "versions" of ARES, each one using a different algorithm. For this reason, we also used two versions of **AgentDroid** to analyze the performance of our approach in a fair way.

Having said that, in this study we used 5 tools: **AgentDroid** $_{DDPG}$, **AgentDroid** $_{SAC}$, ARES_{DDPG}, ARES_{SAC} and Monkey. We executed each tool on the 11 apps listed in Table 13.2. For each app, we executed the DRL-based approaches with 16 episodes, each

with 250 events. The first episode served as an exploration phase and involved random actions, reason why the first episode was excluded from the results. In the case of Monkey 15 executions of the latter were carried out, each with 250 actions to ensure comparability with the results obtained by ARES and **AgentDroid**. The parameters used for both ARES and AgentDroid are the same, with a fixed number of 4000 *timesteps* and 250 of *max_steps* or *episode_length*, resulting in a total of 16 episodes being executed. On the other hand, the parameters for the Monkey tool included the *ignore-crashes* setting to prevent premature termination in case of error detection, as well as a 300 millisecond *throttle* between each action.

Regarding $\mathbf{RQ}_1$, we computed class coverage, method coverage, and counted the number of exceptions/errors generated during the apps' execution. The aforementioned values were computed after each episode. Following a similar strategy used in previous studies (*e.g.,* [131, 331]), we analyzed the cumulative coverage to understand the exploration evolution and number of exceptions/error identified after the 15 episodes. We report the results using tables and box-plots.

For $\mathbf{RQ}_2$, we analyzed the achieved coverage across the 15 episodes (and the Monkey execution) in terms of apps' classes and methods that were visited by different tools, and identify those that were unique for each too, similarly to previous studies (*e.g.,* [131, 331]). We also conducted a similar analysis for the case of identified errors/exceptions. To explain the findings we provide qualitative examples.

13.5 Results

This section reports the results for the two research questions formulated in Section 4.

13.5.1 *RQ1.* How is the performance of AgentDroid when compared with baselines approaches

To answer this question, we present the results divided into two sections, associated with specific metrics, the cumulative coverage (CC), and the number of errors identified. For the first metric, we present a descriptive analysis of the results, followed by the results of executing statistical tests. Next, the descriptive statistics are presented with graphics (*e.g.,* boxplots). For the statistical analysis of CC, we executed a pairwise comparison for each distribution using two sided Mann-Whitney test [332], with Holms Bonferroni correction [333], and computed the Cliff's delta [334] for measuring the effect size. For the second metric, we present a descriptive analysis.

Cumulative coverage (CC). Fig. 13.3 depicts the distribution of the CC for all the combinations of $approach+algorithm$ when evaluated in the 11 aforementioned applications. In each boxplot, the third quartile was represented as a line that divided the boxplot into light gray and dark gray pieces, and a white circle was used to display the mean of each distribution. With this in mind, it is possible to identify that Monkey has the highest third quartile (Q_3), *i.e.,* 75% of the CC is above the other distributions; and the lowest Q_3 is present in the $Agentdroid + DDPG$. However, the highest CC was achieved by $Agentdroid + SAC$,

with a CC of 63.41% for the Budget app.The distribution of the CC can be seen in detail in Table 13.3, in which it is also possible to identify two particular cases for $Ares + DDPG$ with Twik and SipDroid apps, in which no value is presented. This was caused because both applications failed to execute in that configuration.

When comparing the results of each $approach + algorithm$ for the 11 applications, we identified that in five cases Monkey overruled the other approaches; in two scenarios $Ares + SAC$ had better results and in four cases $AgenDroid + SAC$ had the best results. Analyzing the cases in which Monkey outperformed the other approaches, it is possible to see that, in three cases, the difference between the CC and the second best CC is greater than 10%. And the second-best CC was achieved by $AgentDroid+SAC$ or $Ares+SAC$. In the other remaining cases, Monkey outperformed $AgentDroid+SAC$ by a difference between 5% and 9%, which had the second best CC. This tendency of Monkey to have a better performance could be related to the intrinsic nature of the reinforcement learning algorithms, in which they could need some time to learn or could be memorizing particular paths inside the app which are not applicable to navigating inside the entire app.

In the four cases in which $AgentDroid + SAC$ outperformed the other approaches, in all the cases, the second-best performance was achieved by Monkey. Finally, in the two cases in which $Ares+SAC$ outperformed the other results, in one case Monkey and $AgentDroid+SAC$ had the same CC (*i.e.,* for the app SipDroid), and in the other case, *i.e.,* with the Twik app, $AgentDroid + SAC$ outperformed Monkey by more than 20%.

After executing the statistical tests with the distributions, with a $p-value$ of 5%, we did not identified any statistical significant differences between distributions; therefore, we did not compute the cliff delta. With this we mean that we do not have enough data to claim that the performance of the five different approaches,*i.e.,* $Monkey$, $AgentDroid + SAC$, $Agendroid + DDPG$, $Ares + SAC$, $Ares + DDPG$ in the 11 apps have a difference on their performance.

Summary. When exploring the data at Table 13.3 and the boxplots Fig. 13.3, the different approaches and algorithms perform differently and there are some differences as in the case of CC for Budget app when executing the statistical tests we could not determine that any approach performed significantly different than any other. Based on these results, further analysis taking into account categories and application types might be performed to reduce the impact of certaint components in the analysis and execution of test.ß

Identified errors. When analyzing the number of *identified errors*, a maximum of 112 and a minimum of 1 error were identified. Table 13.3 depicts the number of identified errors per approach. It is also possible to see that the maximum number of identified errors was obtained by $AgentDroid+DDPG$ followed by $AgentDroid+SAC$ and $ARES+SAC$. The two highest numbers of errors were discovered in the same app. For that app, three out of the five possible approaches identified seven errors or more. When analyzing the great number of errors identified by $AgentDroid + DDPG$ and $AgentDroid + SAC$ in the Radio Reddit application, it was observed that the MediaPlayer component generated similar syntactical errors, which were distinguishable only by a hexadecimal identifier. According to the Android documentation, this identifier specifies the type of error thrown by the MediaPlayer component.

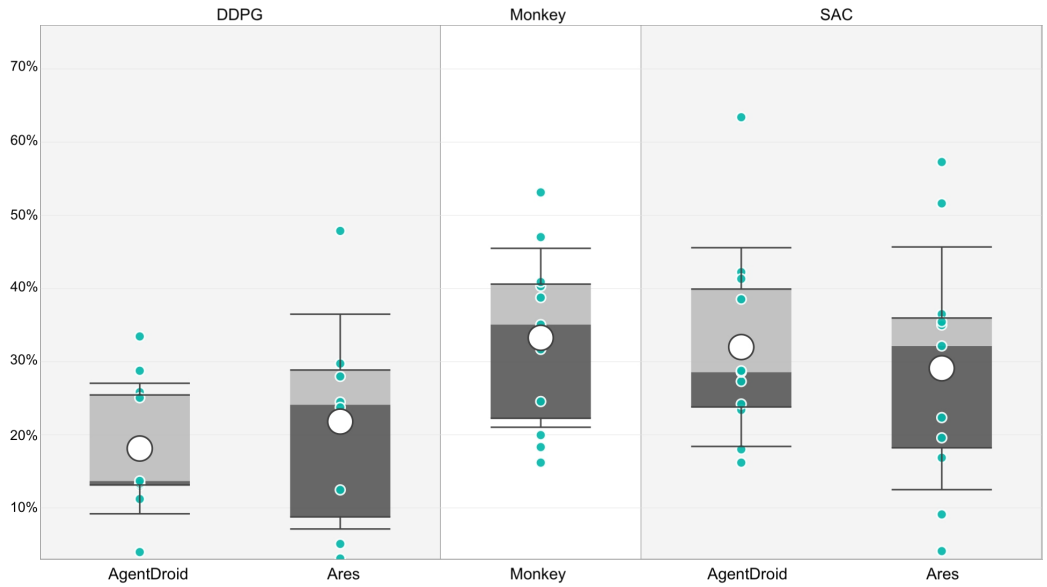


Figure 13.3. Cumulative coverage of the 11 applications by program and algorithm. Mean values marked by white circles

Table 13.3. Cumulative coverage expresed in percentage and the Number of errors found per each APK for each approach and each algorithm. The - means that the experiment failed to be executed. and the bolded numbers are used to identify the best result between $approach + algorithm$

	Number of errors identified					**Cumulative coverage in %**				
	ARES		**AgentDroid**		**Monkey**	**ARES**		**AgentDroid**		**Monkey**
APP	**DDPG**	**SAC**	**DDPG**	**SAC**	**Random**	**DDPG**	**SAC**	**DDPG**	**SAC**	**Random**
Apollo	1	1	1	**2**	1	24.48	19.58	25.83	27.29	**35.06**
iFixit	4	**14**	0	1	2	12.47	9.12	13.18	18.01	**24.55**
Mileage	0	0	0	0	0	5.08	22.35	11.22	23.40	**40.32**
Hydrate	0	0	0	0	0	27.98	32.14	13.10	28.57	**47.02**
Budget	0	0	0	0	0	47.87	51.63	25.06	**63.41**	53.13
Radio Reddit	0	7	**112**	32	0	29.73	36.49	33.45	**42.23**	40.88
Twik	-	**2**	0	1	1	-	**57.29**	17.76	41.32	19.96
HydroMemo	0	0	0	0	0	23.75	35.00	28.75	28.75	**38.75**
SipDroid	-	0	0	**1**	0	-	**16.86**	13.36	16.19	16.19
APG	0	2	0	**3**	0	2.05	4.10	3.97	**24.21**	18.32
Tomdroid	0	**1**	0	0	0	3.09	35.44	13.69	**38.52**	31.68

Summary. Each approach on the eleven analyzed applications showed variability in the number of errors identified. The faulty application "Radio Reddit" was found to have a higher number of errors compared to the other applications with a maximum of 112 errors

in a single execution. AgentDroid was able to identify errors in three applications, ARES in four, and Monkey in one. However, the use of the DDPG algorithm in AgentDroid proved to be particularly effective in detecting errors in the faulty app "Radio Reddit," surpassing the performance of the SAC algorithm. These results suggest the importance of selecting an appropriate approach and algorithm, as the characteristics of the application being tested can greatly impact the effectiveness of error detection. It is also important to note that additional data may be required to ensure the performance of the DDPG algorithm, as it only demonstrated promising results in the single case of the faulty "Radio Reddit" application.

13.5.2 *RQ2.* Are there ortogonal results within the executions of the selected approaches ?

In order to identify the existence of complementary results across the different analyzed methods, we gathered the records of the visited methods within the apps, via instrumentation, during the execution of the analyzed tools. These results, can be grouped in 5 main levels:

1. **Complete Study level:** visited methods can be analyzed as a whole group, *i.e.,* the analysis is performed using the entire set of methods as they were part of one system. This granularity level provides a general understanding of the capabilities of the tools over the complete set of apps.

2. **App-specific level:** visited methods are grouped using the name of the apps. This level of analysis enables the analysis of the behavior of each tool over a specific app. This level is not analyzed in this article due to space limits.

3. **App-specific Package Level:** Visited methods are grouped based on the app and package they belong. This level of granularity allows us to analyze the capabilities of the different tools to visit the existing packages within an app. By keeping this level of analysis, it is possible to understand the level of orthogonality the tools generate in their results. This level of granularity is not analyzed in this article due to space limits.

4. **App-specific Class Level:** Visited methods are grouped based on the app, the package, and the class they belong to. This level of granularity enables a more detailed analysis since, by having this information, practitioners can identify the sections of the app that are not visited and prioritize the execution of additional tests over these sections.

5. **Method-specific level:** Visited methods are analyzed as separated units. This level of granularity is the more detailed one since it considers the app, the package, and the class they belong to. By using this level of granularity in the analysis, researchers and practitioners can identify the specific methods that are not called. Due to the existing code analysis capabilities, at this granularity level, researchers can generate the app's call graph and identify the sequence of calls that are not being exercised to achieve higher coverage. This level of granularity is not analyzed in this article due to space limits.

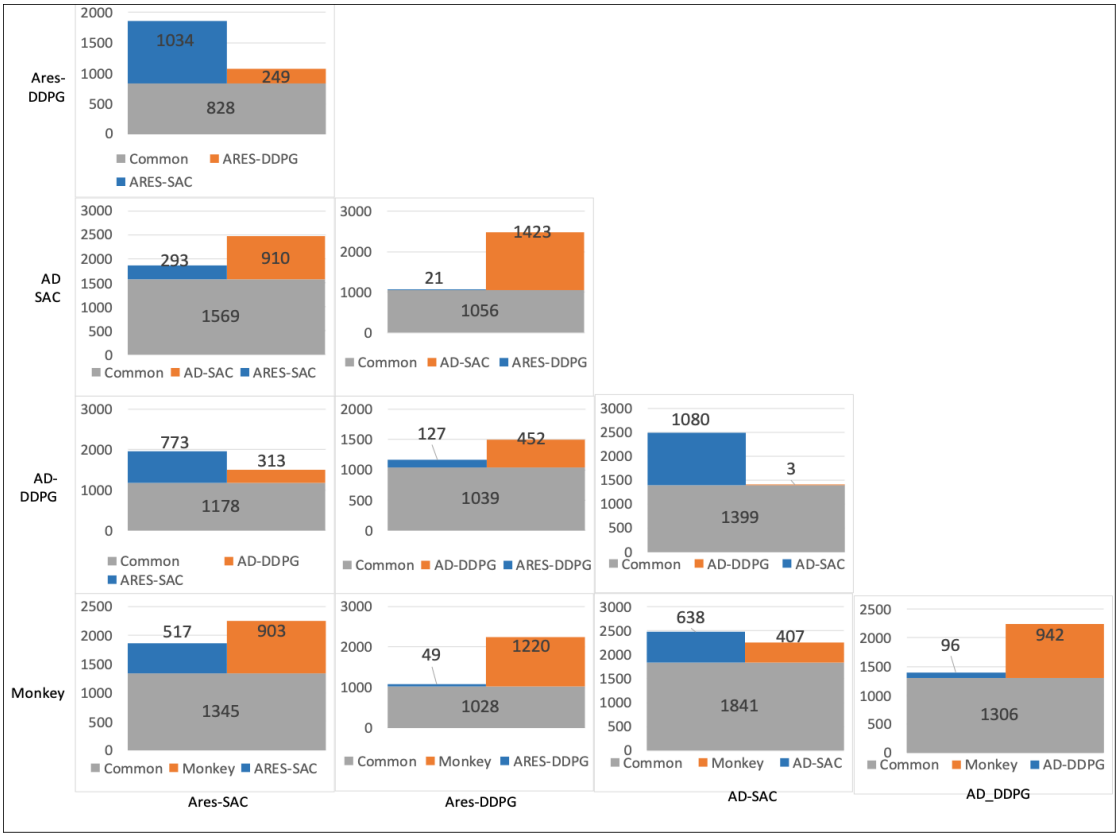


Figure 13.4. Comparison of amount of visited methods for each tool pairing using the complete study level of granularity.

Regarding the evaluation of our results, we present the analysis based on granularity levels (1) and (4). In order to achieve this, we computed the differences and similarities between each pair of tools. For example, when comparing Monkey and **AgentDroid** $_{DDPG}$ we need to compute: (i) the amount of methods visited by both tools, (ii) the amount of methods visited only by Monkey and (iii) the amount of methods visited only by **AgentDroid** $_{DDPG}$. Since we are comparing 5 tools, there are 10 comparisons and each comparison is based on 3 values.

Regarding the first granularity level, Fig. 13.4 presents the summary of the comparisons. The results are presented using a matrix layout, since the diagonal of the matrix is not required (*i.e.,* it would represent a comparison of each tool with itself), the figure presents graphically the comparison between each pair of tools in a 4x4 matrix. Each cell in the matrix represents a comparison between two tools, the gray bar represents the amount of methods that were visited by both tools, the orange bar represents the amount of methods visited only by the tool located in the Y axis and the blue bar represents the amount of methods visited only by the tool located in the X axis. For example, the case of the Monkey and ARES$_{SAC}$ comparison, located in the bottom left corner, $Monkey$ and $Ares + SAC$ have in

common 1345 method calls, and besides those calls, $ARES_{SAC}$ had 517 additional method calls (blue bar), and $Monkey$ had 903 in addition (orange bar).

In the case of ARES_{SAC} (first column of graph), we can see that there is a possitive difference when it is compared to the tools based on *DDPG* algorithm. Nonetheless, it falls short when compared with **AgentDroid** $_{SAC}$ and Monkey. Due to the limitations of this granularity level, we cannot conclude that ARES_{SAC} outperforms both tools using DDPG. However, taking into account second column and third row, both representing comparisons of DDPG-based tools against the remaining tools, it can be seen that DDPG-based tools present lower results regarding the amount of methods visited.

In the case of **AgentDroid** $_{SAC}$ (second row and third column) we can identify that in the 4 comparisons done, there is a positive difference with an average of 770 methods visited additionally when compared with the other tools. Additional information regarding these differences might be identified using a more detailed input (*i.e.,* deeper granularity level).

Finally, regarding Monkey (last row) there is a positive difference when compared to all other tools but **AgentDroid** $_{SAC}$, however, the results are really close and share a considerable amount of visited methods.

With this in mind, we are required to increase the level of granularity in order to get more information that allows us to study the level of orthogonality in the obtained results. For this purpose, we extended our results' analysis to granularity level (4), therefore, we analyzed the differences in terms of the visited classes within the apps. In order to do this, we identified and listed the classes containing the methods visited for each tool, then, we organize the list based on the app and packages. AS result of this, we are now able to identify the classes visited by each tool and the package each class belongs to. Finally, we study the package and class content to identify the corresponding section of the app.

In the case of ARES_{SAC}, when compared to the other tools, it is common to identify that the other tools visit more classes within packages related to *UI*, nevertheless, the amount of methods called in these classes is low, therefore, there is no significant differences in terms of new states and new activities visited by the other tools. On the other side, when analyzing the packages and classes visited only by ARES_{SAC}, it is more common to find classes in deeper locations in terms of folders. Therefore, ARES_{SAC} visits deeper locations when compared to **AgentDroid** and Monkey.

Regarding **AgentDroid**, as it was mentioned before, the main differences when compared to ARES approaches is the amount of classes and activities visited, nonetheless, it does not explores at the same depth. In constrast, when compared to Monkey the packages and classes vary on the purpose of these. This means that there is not a standard difference between **AgentDroid** and Monkey. However, the amount of classes and packages visited in common is high. Based on this, we can conclude that our approach has similar coverage in terms of packages and classes visited with Monkey.

Summary. In order to analyze the level of orthogonality within our results, we compared the achieved coverage at two different levels of granularity. First, taking into account only the method name, leading to a higher amount of visited methods by SAC-based approaches, while having a high amount of methods visited in common within approaches. Nevertheless, this level of granularity did not provide us enough information to evaluate the orthogonality,

based on this, we analyzed as second step the orthogonality taking into account the app, package and class information. As result of this analysis, we identify that ARES$_{SAC}$ visited less activities but went deeper in the exploration process, while **AgentDroid** and Monkey had a broader exploration of superficial classes. Therefore, we can conclude that ARES and **AgentDroid**/Monkey can be used together to obtain an enhanced result.

13.6 Threats to validity

For the purpose of our study to have validity, we had to face different threats, first, the amount of apps used for the study, that was mitigated by selecting a set of apps from different categories and different sizes to obtain a diverse set of results. Second, the rigurosity of the execution process, that was mitigated by using standarized parameters between approaches such as the amount ot episode and events. Additionally, we used a state-of-the-art and commonly used tool (*i.e.,* Monkey) to compare our results. In order to reduce the probability of having unreproducible results, we executed each tool more than 1 time and enhance the error identification process by generalizing the issues based on the error trace presented.

14

Automated Testing of Android Apps: Discussion and Summary

We presented **Kraken 2.0**, the first publicly available and open-source tool for platform-agnostic cross-device interaction-based testing of Android and web apps. **Kraken 2.0** allows developers/testers to write testing scenarios in an E2E fashion for applications that require the interaction of two or more users on different devices/browsers and between platforms. **Kraken 2.0** also allows developers/testers to combine E2E scenarios with random generation of events and data. As part of our research process results, we created examples of Kraken being used with different apps, showing the capabilities to test daily life apps such as email clients and entertainment apps such as Spotify and multiplayer games. Future work could be devoted to (i) supporting the execution of scenarios in iOS devices, (ii) including a systematic exploration mode to enable cross-device GUI ripping, and (iii) generating studies on the usage of the tool by practitioners in their daily life. We have made a first approach to the latter option by enabling a set of tutorials that students of our classes on "Automated Testing" and "Software Engineering for Mobile Apps Development" from the Masters in Software Engineering at Universidad de los Andes, Colombia might follow. Results on using Kraken in these classes have provided us feedback to improve **Kraken 2.0** continuously.

Regarding the usage of DRL-based techniques for automated testing, we have conducted a study that compares the behavior of AgentDroid, our approach that uses both faults and coverage to compute the reward, and baseline approaches in terms of the capabilities to identify faults and improve coverage. Our results show similar outcomes between using a composed reward function and single parameter functions. Nevertheless, the execution time was a factor that differentiated the execution of baseline approaches and our approach, providing an advantage to using single-parameter reward functions.

The usage of these automated techniques provides mechanisms for developers to reduce their interaction with the testing process; based on this, the existing state-of-the-art approaches, and also **AgentDroid**, can be extended to react according to the execution context, for example, by identifying the version of the OS to execute standard or newly added navigation gestures. Additionally, parameterized services can be added to provide custom app exploration/execution. For example, by using a text generation service that returns

strings that meet certain business rules or constraints.

Finally, DRL-based techniques might be used for the execution of multi-agent scenarios by exploiting the capabilities of **Kraken 2.0** for interaction-based testing. An initial approach might be to create a shared memory service, to be accessed by different devices via **Kraken 2.0** signaling capabilities; this with the objective of improving the app exploration during the training phase. Another approach could focus on generating interaction-oriented tests by relying on the execution engine and enable capabilities provided by **Kraken 2.0**. Lastly, multi-device execution could be used to perform an optimization of weights used within the combined reward function, with the objective of targeting any of the specific metrics with single- and multi-agent executions.

Part VI

Epilogue

15

Conclusion

The mobile app development environment is in constant change and exerts continous pressure over developers and practitioners to create quality apps in the minimum amount of time possible. This have lead to the automation and usage of external services to improve the outcome, *i.e.,* the application. Based on this requierement, in this thesis we present a set of approaches created to improve and enhance the development process of android apps by preventing, identifying and fixing issues related to different quality attributes and testing techniques. In order to achieve this, we conducted a set of studies that lead to the creation of several tools used to enable new capabilities in the process of android app development.

In order to achieve this, we analyzed the state-of-the-art approaches to identify the existing gaps in the development process. We first analyzed the approaches that used static analysis to identify issues within the development process. After this, we studied the existing approaches for automated testing of android apps, in order to understand the current behaviour of developers while assuring the quality of their apps via dinamic analysis. Finally, we investigated the state-of-the-art regarding automated fixing of issues for android apps, to understand what are practitioners and researchers doing in order to fix the identified issues prior to the publication of a new app version.

After the aforementioned review, we identified a research gap regarding 4 main topics around analysis and testing of Android Apps, based on these we conducted a series of studies and created tools to improve the state of the art. Therefore, we reduced the research gap by enabling a set of tools and studying the state of the practice regarding the selected topics. This studies and tools increase the capabilities available for practitioners and researchers to potentially create/mantain in less time quality apps that might fulfill the users requirements.

We presented at the end of each part of this document a discussion on the results of our proposed approaches along with an analysis of the future work that practitioners and researchers might follow to investigate new characteristics or extend our previous approaches. The following sections of this chapter present a summary of the contributions and future work that researchers and practitioners might tackle around software engineering for mobile apps.

15.1 Summary of Research Contributions

The contributions of this thesis provide new approaches, insights and new techniques for four topics related to analysis and testing of android apps. In the following we summarize these contributions, linking them to the respective chapters. Additionally, the concrete contributions to the scientific scholarly output performed by the author of this thesis can be found in Appendix B

15.1.1 Mutation Testing (Part II)

Mutant generation (Chapter 3)

We conceived and implemented the most comprehensive framework for mutant generation of Android apps, `MutAPK`, at APK-level. This framework presents the implementation of 38 mutant operators based on a taxonomy of android-specific faults. This framework, was then evaluated in terms of the number of non-compilable, trivial, equivalent, and duplicate mutants generated and their capacity to represent real faults in Android apps as compared to other well-known mutation tools. The result showed an improvement of 93.83% (*i.e.,* 4.32 from 4.61 seconds) in the mutation time and 87.05% (*i.e.,* 174.73 from 195 seconds) for compilation/assembling times. However, the source code-based mutation approach generates only 2.97% (*i.e.,* 263 of 8847 mutants) of non-compilable or trivial mutants as compared to the 6.8% (*i.e.,* 5105 of 75053 mutants) of the APK-based mutation approach.

Mutant Selection (Chapter 4)

We present `MutAPK 2.0`, an improved version of our open source mutant generation tool (`MutAPK`) for Android apps at APK level. To the best of our knowledge, `MutAPK 2.0` is the first tool that enables the removal of dead-code mutants, provides a set of mutant selection strategies, and removes automatically equivalent and duplicate mutants. We evaluated this new version, by using it with 10 open source android apps. We analyzed the results regarding the reduction of the mutant dataset when removing dead-code mutants, as well as equivalent and duplicate ones. In addition, we evaluated whether the implementation of the selection techniques produced balanced sets (in terms of the mutation operators).

15.1.2 Internationalization of Android Apps. (Part III)

Detection of internationalization issues. (Chapter 6)

We presented an empirical study on how i18n can impact the GUIs of Android apps. In particular, we investigated the changes, bugs and bad practices related to GUIs when strings of a given default language (i.e., English in this case) are translated to 7 different languages. To this, we created a source-codeless approach, ITDroid, for automatically (i) translating strings, and (ii) detecting bad practices and collateral changes introduced in the GUIs of Android apps after translation. ITDroid was used on a set of 31 Android apps and their

translated versions. Then, we manually validated the i18n changes that introduced bugs into the GUIs of the translated apps.

Prevention and fixing of internationalization issues. (Chapter 7)

Continuing with the developement process of ITDroid, we presented our efforts towards enhancing automated identification and enabling automated fixing of i18n bugs automatically identified and localized by ITDroid (*i.e.,* extracting hard-coded string and fixing RtL GUI components mirroring). Additionally, we enhanced the final report to provide a more comprehensive guide of the issues along with the improvement of the tool to enable the capability to automatically repair a subset of the identified issues. The main result of this process is the extension of our open source tool.

15.1.3 Connectivity Management for Android Apps. (Part IV)

Identification of Connectivity Issues on Android Apps. (Chapter 9)

We conducted the first study on Eventual Connectivity (ECn) issues exhibited by Android apps, by manually inspecting 971 scenarios related to 50 open-source apps. We found 304 instances of ECn issues (6 issues per app, on average) that we organized in a taxonomy of 10 categories. Based on our findings, we distill a list of lessons learned for both practitioners and researchers, indicating directions for future work. Our results in this matter was published in the following article:

Automated - Prevention, Detection and Repairment of Connectivity Issues. (Chapter 10)

Taking into account the identified errors on the previous chapter, we identified that connectivity issues (e.g., mishandling of zero/unreliable Internet connection) can result in bugs and/or crashes, negatively affecting the app's user experience. While these issues have been studied in the literature, there are no techniques able to automatically detect and report them to developers. In order to tackle this, we built CONAN, a tool able to detect statically 16 types of connectivity issues affecting Android apps. We assessed the ability of CONAN to precisely identify these issues in a set of 44 open source apps, observing an average precision of 80%.

15.1.4 Automated testing of Android Apps. (Part V)

Enabling Automated Platform-Agnostic Multi-Device Interaction-based Testing for Android and Web Apps. (Chapter 12)

We presented **`Kraken 2.0`**, an open source automated End-2-End (E2E) testing tool for defining and executing scenarios that involve inter-communication between two or more browsers or mobile devices. **`Kraken 2.0`** is an enhanced version of **Kraken v1.0** [315], which (i) includes fixes to issues, (ii) improves user experience, and (iii) has new features to support web-mobile interaction and fuzzing. In order to show **`Kraken 2.0`** capabilities, we used

the tool with 10 combinations of web and mobile apps, in which we created and executed cross-device E2E test scenarios.

Using DRL for Automated Testing of Android Apps. (Chapter 13)

We presented a study evaluating the performance of an DeepRL approach that uses a combined reward function based on the amount of identified errors, the achieved coverage and the GUI change rate. We created a tool called **AgentDroid**, that was executed over 11 open source android apps. We compared the performance of this approach against state-of-the-art solutions and analyzed the results in terms of accumulated coverage and amount of identified errors. Finally, we analyzed the results to identify if there were orthogonal results between the execution of the tools.

15.2 Future work

Considering the results presented in this thesis, we encourage practitioners and researchers to extend the work done to automate analysis and testing tasks at APK level. Using APKs as input provides several benefits, which can be exploited to improve the development process of Android Apps. In the case of researchers, the approaches presented for mutation testing and internationalization enabled the generation of models that could be used to study additional quality attributes of Android apps, like accessibility and compatibility. For example, researchers might use call graphs to analyze the coupling of software modules within closed-source apps. Likewise, practitioners might use the proposed approaches to enable new capabilities within their apps, such as support for commonly used languages and the adaptation of UI to i18n guidelines. Additionally, practitioners might design new testing protocols that rely only on using APKs, since these require less computational power and could provide rapid feedback.

Regarding mutation testing, researchers might follow several paths to extend testing capabilities. As mentioned in the corresponding chapter, one of the main fallbacks of performing mutation testing at the APK level is the understandability of the changes injected. Therefore, one of the leading research gaps that could be tackled is the generation of "translation" of changes that allow developers to understand the injected changes and improve the test suite capabilities. An existing restriction regarding this could be the obfuscation of code which reduces the probability of generating an understandable solution. Another approach researchers might consider is the extension of the proposed approaches to support a more extensive set of android-specific faults, define new selection techniques that rely on development patterns, or consider change history to generate only mutants on the recently modified code.

Regarding internationalization testing, researchers might create approaches based on the UI render available in Android Studio; this could allow the preliminary evaluation of UI behavior for the supported languages. Additionally, it might enable the preliminary evaluation of other graphical-related quality attributes, such as usability or accessibility. This approach might be added to the different IDEs like the approach proposed by Petrović and

Ivanković [335]. Researchers might also extend the current approach with additional features to enhance the analysis of commonly used natural languages, for example, when working with Right-to-Left languages (*e.g.,* arabic, hebrew, persian) certain type of multimedia content like graphs should be also mirrored. Finally, practitioners might extend the proposed approaches with custom validations to check design rules imposed by the business or by design guidelines. For example, ITDroid does not register the distance between elements only its positional relations, therefore, practitioners might extend the Layout Graph definition to register and check the distance between elements. In addition, researchers might propose approaches to support analysis and testing of Android applicationes built using Flutter, we proposed an initial approach to enable internationalization, nevertheless, there is no public available grammar to generate the parsers required to statically analyze the code. Likewise, the implementation of internationalization in flutter apps is not standarized and developers might use different libraries and mechanism to store and retrieve strings.

Regarding connectivity management, researchers might extend the capabilities available for Android linter to support additional quality attributes and enhance the issues identification process. At the same time, as mentioned in the corresponding chapter, researchers might evaluate the proposal of approaches that enable the self-adaptation of Android apps. The discussed approach based on code annotations could be use for other quality attributes like battery, energy consumption, current memory usage, among others. Taking into account that the changes would happen during execution, this approach should use dependency injection. Taking into account this, DI could be enhance by using decorators on a context handler that manage the instantiation of services according to the values of the different QA metrics during the execution.

Finally, it is worth discussing, due to its increasing adoption within developers, about automated code generation/recommendation based on AI techniques. Currently available solutions as Github copilot, ChatGPT or Tabnine provide developers with tools to generate code based on the current state of the project or specific questions. Nevertheless, this approaches are trained based on existing codebases on repositories like github, and as it was shown in this thesis, open source applications lacks of good practices regarding different quality attributes. Therefore, an approach researchers and practitioners might follow is the evaluation and enhancement of the parameterization in order to create/recommend code following better code practices and that avoid fault injections.

Part VII

Appendix

A

Additional Studies related to Android Apps

A.1 Investigating Metadata and Survivability of Top Android Apps

Hall-of-Apps: The Top Android Apps Metadata Archive. [336]
Bello-Jiménez, Laura and **Escobar-Velásquez, Camilo** and Mojica-Hanke, Anamaria and Cortés-Fernández, Santiago and Linares-Vásquez, Mario In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR), 2020.*

The amount of Android apps available for download is constantly increasing, exerting continuous pressure on developers to publish outstanding apps. Google Play (GP) is the default distribution channel for Android apps, which provides mobile app users with metrics to identify and report app quality such as rating, amount of downloads, previous users' comments, etc. In addition to those metrics, GP presents a set of top charts that highlight the outstanding apps in different categories. Both metrics and top app charts help developers to identify whether their development decisions are well valued by the community. Therefore, app presence in these top charts is valuable information when understanding the features of top-apps. In this paper, we present *Hall of Apps*, a dataset containing top charts' apps metadata extracted (weekly) from GP, for 4 different countries, for 30 weeks. The data is presented as (i) raw HTML files, (ii) a MongoDB database with all the information contained in the app's HTML files (*e.g.,* app description, category, general rating, etc.), and (iii) data visualizations built with the `D3.js` framework. A first characterization of the data along with the URLs to retrieve it can be found in our online appendix: **https://bit.ly/2uIkXs8**

Crème de la crème. Investigating Metadata and Survivability of Top Android Apps. [337]
Mojica-Hanke, Anamaria and Bello-Jiménez, Laura and **Escobar-Velásquez, Camilo** and Linares-Vásquez, Mario In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.*

Mobile apps are distributed via online markets allowing practitioners to reach users worldwide; on the other side, users select what apps are more suitable for their preferences from a large set of apps offering similar features and capabilities. To facilitate that selection process, the distribution markets have different mechanisms, such as comments, ratings and top listed apps included a curated list. As it is well known, apps stores metadata can provide insights for new, popular features or fixing existing bugs, as reported in previous works. However, to the best of our knowledge, app store data have not been used to identify possible predominant characteristics of successful apps using as a reference the aforementioned top lists. Thus, in this paper, we present a study that analyzes the metadata of apps belonging to Google Play top lists during 30 weeks in 4 countries to distill features of successful apps. Unfortunately, our results suggest that apps store data from top lists do not provide enough information to identify predominant characteristics for each top.

A.2 Quality Attributes of Android Apps

Taxonomy of security weaknesses in Java and Kotlin Android apps. [338]
Mazuera-Rozo, Alejandro and **Escobar-Velásquez, Camilo** and Espitia-Acero, Juan and Vega-Guzmán, David and Trubiani, Catia and Linares-Vásquez, Mario and Bavota, Gabriele In *Journal of Systems and Software), 2022.*

Android is nowadays the most popular operating system in the world, not only in the realm of mobile devices but also when considering desktop and laptop computers. Such popularity makes it an attractive target for security attacks, also due to the sensitive information often manipulated by mobile apps. The latter is going through a transition in which the Android ecosystem is moving from the usage of Java as the official language for developing apps, to the adoption of Kotlin as the first choice supported by Google. While previous studies have partially studied security weaknesses affecting Java Android apps, there is no comprehensive empirical investigation studying software security weaknesses affecting Android apps considering (and comparing) the two main languages used for their development, namely Java and Kotlin. We present an empirical study in which we: (i) manually analyze 681 commits including security weaknesses fixed by developers in Java and Kotlin apps, to define a taxonomy highlighting the types of software security weaknesses affecting Java and Kotlin Android apps; (ii) survey 43 Android developers to validate and complement our taxonomy. Based on our findings, we propose a list of future actions that could be performed by researchers and practitioners to improve the security of Android apps.

A Preliminary Study on Accessibility of Augmented Reality Features in Mobile Apps. [339]
Naranjo-Puentes, Sergio and **Escobar-Velásquez, Camilo** and Vendome, Christopher and Linares-Vásquez, Mario. In *Proceeding of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.*

The capabilities of Android applications have been increasing along with the advancements of the underlying mobile devices. This has allowed developers to include features that require more resources and provide more complex functionality inside mobile apps. An example of this is the usage of Augmented Reality (AR) in mobile applications. AR allows developers to experiment with new immersive experiences for their users in a wide variety of application areas. AR content is often visual and it represents a challenge for users with visual impairments, especially if these features are core to the underlying application. In this paper, we present a preliminary study aimed at investigating accessibility of AR functionalities of mobile applications that are not specifically designed for users with visual impairments (*i.e.,* applications of which these users could be part of the intended audience, but they are not the specific audience). To accomplish this, we conducted a case study with 49 individuals without visual impairment and 5 individuals that are visually impaired, who used 5 applications with AR features. Our results demonstrate that the analyzed apps lack accessibility mechanisms within their AR functionalities.

B

Individual Author Contribution

The CRediT (Contributor Roles Taxonomy) standard [340] is a mechanism to indicateand acknowledge researchers' contributions given a scholarly output. CRediT is a high-leveltaxonomy, including 14 roles, that can be used to represent the roles typically played bycontributors to scientific scholarly output as described in the following.

- **Conceptualization (C)**: Formulation or evolution of overarching research goals and aims.
- **Methodology (M)**: Development or design of methodology; creation of models.
- **Software (S)**: Programming, software development; designing computer programs; implementation of the computer code and supporting algorithms; testing of existing code.
- **Validation (VA)**: Verification, whether as a part of the activity or separate, of the overall replication/reproducibility of results/experiments and other research outputs.
- **Formal analysis (FO)**: Application of statistical, mathematical, computational, or other formal techniques to analyze or synthesize study data.
- **Investigation (I)**: Conducting a research and investigation process, specifically performing the experiments, or data/evidence collection.
- **Resources (R)**: Provision of study materials, reagents, materials, patients, laboratory samples, animals, instrumentation, computing resources, or other analysis tools.
- **Data Curation (D)**: Management activities to annotate (produce metadata), scrub data and maintain research data (including software code, where it is necessary for interpreting the data itself) for initial use and later reuse.
- **Writing - Original Draft (WO)**: Preparation, creation and/or presentation of the published work, specifically writing the initial draft (including substantive translation).
- **Writing - Review & Editing (WR)**: Preparation, creation and/or presentation of the published work by those from the original research group, specifically critical review, commentary or revision – including pre- or post-publication stages.
- **Visualization (VI)**: Preparation, creation and/or presentation of the published work, specifically visualization/data presentation.
- **Supervision (SU)**: Oversight and leadership responsibility for the research activity planning and execution, including mentorship external to the core team.

- **Project administration (P)**: Management and coordination responsibility for the research activity planning and execution.
- **Funding acquisition (FU)**: Acquisition of the financial support for the project leading to a publication.

In order to identify in a standardized manner the contributions performed by the authorof the present thesis across the chapters, Table A.1 highlights the contributions of the authorbased on CRediT.

Table B.1. Contributions of the author in the scientific scholarly output presented in this thesis

Study	Chapter	C	M	S	VA	FO	I	R	D	WO	WR	VI	SU	P	FU
Enabling mutant generation for open-and closed-source android apps.	Chapter 3	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	
MutAPK: Source-codeless mutant generation for android apps.	Chapter 3	✓	✓	✓	✓		✓	✓	✓	✓		✓		✓	
MutAPK 2.0: A tool for reducing mutation testing effort of Android apps.	Chapter 4	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	
An empirical study of i18n collateral changes and bugs in guis of android apps.	Chapter 6	✓	✓	✓	✓		✓	✓	✓	✓		✓		✓	
ITDroid: A tool for automated detection of i18n issues on android apps.	Chapter 6	✓	✓	✓	✓		✓	✓	✓	✓		✓		✓	
Studying eventual connectivity issues in Android apps	Chapter 9	✓	✓	✓	✓		✓	✓	✓	✓		✓		✓	
Detecting connectivity issues in android apps	Chapter 10	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	
Kraken-mobile: Cross-device interaction-based testing of android apps	Chapter 12	✓	✓	✓			✓	✓		✓		✓	✓		
Kraken: A framework for enabling multi-device interaction-based testing of Android apps.	Chapter 12	✓	✓	✓			✓	✓			✓		✓		
Kraken 2.0: A platform-agnostic and cross-device interaction testing tool	Chapter 12	✓	✓	✓			✓	✓			✓		✓		
Kraken 2.0: A platform-agnostic and cross-device interaction testing tool	Chapter 12	✓	✓	✓			✓	✓			✓		✓		

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[2] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013, pp. 15–24.

[3] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Crowdsourcing user reviews to support the evolution of mobile apps," *JSS*, vol. 137, pp. 143–162, 2018.

[4] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *ASE'17*, 2017, pp. 308–318.

[5] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *ASE'15*, 2015, pp. 429–440.

[6] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *ICST'15*, 2015, pp. 1–10.

[7] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *ICSME'17*, 2017, pp. 399–410.

[8] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *ICSE'16*, 2016, pp. 14–24.

[9] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *FSE'16*, 2016, pp. 499–510.

[10] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *IST*, vol. 88, pp. 67–95, 2017.

[11] C. Escobar-Velásquez, M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutant generation for open-and closed-source android apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 186–208, 2020.

[12] C. Escobar-Velásquez, M. Osorio-Riaño, and M. Linares-Vásquez, "Mutapk: Source-codeless mutant generation for android apps," in *2019 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

[13] C. Escobar-Velásquez, D. Riveros, and M. Linares-Vásquez, "Mutapk 2.0: A tool for reducing mutation testing effort of android apps," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1611–1615.

[14] C. Escobar-Velásquez, M. Osorio-Riaño, J. Dominguez-Osorio, M. Arevalo, and M. Linares-Vásquez, "An empirical study of i18n collateral changes and bugs in guis of android apps," in *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2020, pp. 581–592.

[15] C. Escobar-Velásquez, A. Donoso-Diaz, and M. Linares-Vásquez, "Itdroid: A tool for automated detection of i18n issues on android apps," in *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE, 2021, pp. 52–55.

[16] C. Escobar-Velásquez, A. Mazuera-Rozo, C. Bedoya, M. Osorio-Riaño, M. Linares-Vásquez, and G. Bavota, "Studying eventual connectivity issues in android apps," *Empirical Software Engineering*, vol. 27, pp. 1–43, 2022.

[17] A. Mazuera-Rozo, C. Escobar-Velásquez, J. Espitia-Acero, M. Linares-Vásquez, and G. Bavota, "Detecting connectivity issues in android apps," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 697–708.

[18] W. Ravelo-Méndez, C. Escobar-Velásquez, and M. Linares-Vásquez, "Kraken-mobile: Cross-device interaction-based testing of android apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 410–413.

[19] ——, "Kraken: A framework for enabling multi-device interaction-based testing of android apps," *Science of Computer Programming*, vol. 206, p. 102627, 2021.

[20] W. Ravelo-Mendéz, C. Escobar-Velásquez, and M. Linares-Vásquez, "Kraken 2.0: A platform-agnostic and cross-device interaction testing tool," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 839–843.

[21] W. Ravelo-Méndez, C. Escobar-Velásquez, and M. Linares-Vásquez, "Kraken 2.0: A platform-agnostic and cross-device interaction testing tool," *Science of Computer Programming*, vol. 225, p. 102897, 2023.

[22] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[23] Android. (2023) The activity lifecycle. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle

[24] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 403–414.

[25] Android. (2023) Security tips. [Online]. Available: https://developer.android.com/training/articles/security-tips?hl=en

[26] B. Boehm and V. R. Basili, "Defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[27] Oracle. The java® virtual machine specification. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se8/html/

[28] R. Tolksdorf. Programming languages for the java virtual machine jvm and javascript. [Online]. Available: http://vmlanguages.is-research.de/

[29] D. Bornstein. Dalvik vm internals. [Online]. Available: https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=1

[30] B. B. Ben Cheng. A jit compiler for android's dalvik vm. [Online]. Available: http://www.android-app-developer.co.uk/android-app-development-docs/android-jit-compiler-androids-dalvik-vm.pdf

[31] S. R. Group. Soot. [Online]. Available: https://sable.github.io/soot/

[32] B. Gruver. Smali. [Online]. Available: https://github.com/JesusFreke/smali

[33] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[34] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *PLDI'14*, 2014, pp. 259–269.

[35] L. Li, A. Bartel, J. Klein, and Y. Le Traon, "Automatically exploiting potential component leaks in android applications," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*. IEEE, 2014, pp. 388–397.

[36] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

[37] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*, 2013.

[38] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 77–88.

[39] J. Lin, B. Liu, N. Sadeh, and J. I. Hong, "Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings," 2014.

[40] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *MSR'14*, 2014, pp. 2–11.

[41] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *HotNets-X'11*, ser. HotNets-X. New York, NY, USA: ACM, 2011, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070567

[42] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 78–89.

[43] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[44] ——, "Andarwin: Scalable detection of semantically similar android applications," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 182–199.

[45] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *OOPSLA '13*, 2013, pp. 641–660.

[46] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 67–77.

[47] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber, "Cassandra: Towards a certifying app store for android," in *SPSM'14*, 2014, pp. 93–104.

[48] É. Payet and F. Spoto, "Static analysis of android programs," *Information and Software Technology*, vol. 54, no. 11, pp. 1192–1201, 2012.

[49] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling analysis and auto-detection of cryptographic misuse in android applications," in *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*. IEEE, 2014, pp. 75–80.

[50] Y. Arnatovich, H. B. K. Tan, S. Ding, K. Liu, and L. K. Shar, "Empirical comparison of intermediate representations for android applications." in *SEKE*, 2014, pp. 205–210.

[51] Android. (2019) Ui automator. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[52] ——. (2019) Espresso. [Online]. Available: https://developer.android.com/training/testing/espresso

[53] J. Foundation. Appium. [Online]. Available: http://appium.io/

[54] RobotiumTech. (2019) Robotium. [Online]. Available: https://github.com/RobotiumTech/robotium

[55] Calabash. (2019) Calabash-android. [Online]. Available: https://github.com/calabash

[56] P. L. Xtreme Labs and G. Inc. (2023) Roboelectric. [Online]. Available: https://robolectric.org/

[57] Google, "Create ui tests with espresso test recorder. https://developer.android.com/studio/test/espresso-test-recorder."

[58] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 72–81.

[59] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 349–366.

[60] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 215–224.

[61] Google, "Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey."

[62] ——. (2023) Run a robo test. [Online]. Available: https://firebase.google.com/docs/test-lab/android/robo-ux-test

[63] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[64] Q. Sun, L. Xu, L. Chen, and W. Zhang, "Replaying harmful data races in android apps," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2016, pp. 160–166.

[65] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 316–325, 2014.

[66] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating test cases to expose concurrency bugs in android applications," in *Proceedings of the 31st IEEE/ACM international Conference on Automated software engineering*, 2016, pp. 648–653.

[67] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in android apps," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 58–77.

[68] G. Suarez-Tangil, M. Conti, J. E. Tapiador, and P. Peris-Lopez, "Detecting targeted smartphone malware with behavior-triggering stochastic models," in *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I 19*. Springer, 2014, pp. 183–201.

[69] S. T. Ahmed Rumee and D. Liu, "Droidtest: Testing android applications for leakage of private information," in *Information Security: 16th International Conference, ISC 2013, Dallas, Texas, November 13-15, 2013, Proceedings*. Springer, 2015, pp. 341–353.

[70] J. Qian and D. Zhou, "Prioritizing test cases for memory leaks in android applications," *Journal of Computer Science and Technology*, vol. 31, pp. 869–882, 2016.

[71] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 2014, pp. 176–183.

[72] K. B. Dhanapal, K. S. Deepak, S. Sharma, S. P. Joglekar, A. Narang, A. Vashistha, P. Salunkhe, H. G. Rai, A. A. Somasundara, and S. Paul, "An innovative system for remote and automated testing of mobile phone applications," in *2012 Annual SRII Global Conference*. IEEE, 2012, pp. 44–54.

[73] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 2–11.

[74] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 425–436.

[75] A. R. Espada, M. del Mar Gallardo, A. Salmerón, and P. Merino, "Runtime verification of expected energy consumption in smartphones," in *Model Checking Software: 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. Springer, 2015, pp. 132–149.

[76] T. Zhang, J. Gao, J. Cheng, and T. Uehara, "Compatibility testing service for mobile applications," in *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE, 2015, pp. 179–186.

[77] S. Vilkomir and B. Amstutz, "Using combinatorial approaches for testing mobile applications," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 78–83.

[78] J.-f. Huang, "Appacts: Mobile app automated compatibility testing service," in *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2014, pp. 85–90.

[79] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE international conference on software testing, verification and validation (icst)*. IEEE, 2016, pp. 33–44.

[80] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[81] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[82] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 352–361.

[83] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 187–198.

[84] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[85] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 307–318.

[86] M. Martinez and S. Lecomte, "Towards the quality improvement of cross-platform mobile applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 184–188.

[87] Y. Liu, C. Tantithamthavorn, Y. Liu, P. Thongtanunam, and L. Li, "Autoupdate: Automatically recommend code updates for android apps," *arXiv preprint arXiv:2209.07048*, 2022.

[88] M. T. Azim, I. Neamtiu, and L. M. Marvel, "Towards self-healing smartphone software via automated patching," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 623–628.

[89] Y. Zhao, P. Liu, X. Sun, Y. Liu, Y. LIU, J. Grundy, and L. Li, "Autopatch: Learning to generate patches for automatically fixing compatibility issues in android apps," *Available at SSRN 4254659*.

[90] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 226–237.

[91] M. Mobilio, O. Riganelli, D. Micucci, and L. Mariani, "Filo: Fix-locus localization for backward incompatibilities caused by android framework upgrades," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1292–1296.

[92] W. Guo, Z. Dong, L. Shen, W. Tian, T. Su, and X. Peng, "Detecting and fixing data loss issues in android apps," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 605–616.

[93] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *ESEC/FSE'17*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 233–244. [Online]. Available: https://doi.org/10.1145/3106237.3106275

[94] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183492

[95] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 102–111.

[96] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: an input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 259–269.

[97] D. Rodríguez-Baquero and M. Linares-Vásquez, "Mutode: generic javascript and node. js mutation testing tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 372–375.

[98] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[99] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[100] H. Coles, "Pit. http://pitest.org/," 2017.

[101] I. Moore, "Jester - the junit test tester." 2017, http://goo.gl/cQZ0L1.

[102] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system," *Softw. Test., Verif. Reliab.*, vol. 15, no. 2, Jun. 2005.

[103] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: an efficient and extensible tool for mutation analysis in a java compiler," in *ASE 2011*, 2011.

[104] R. Two, "Jumble," 2017, http://jumble.sourceforge.net.

[105] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," ser. ESEM/FSE 2009, 2009.

[106] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 2015, pp. 1–10.

[107] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[108] Yuan-W. Mudroid. [Online]. Available: https://github.com/Yuan-W/muDroid

[109] A. C. Paiva, J. M. Gouveia, J.-D. Elizabeth, and M. E. Delamaro, "Testing when mobile apps go to background and come back to foreground," in *ICSTW 2019*. IEEE, 2019.

[110] J. Liu, X. Xiao, L. Xu, L. Dou, and A. Podgurski, "Droidmutator: an effective mutation analysis tool for android applications," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 77–80.

[111] "Android mutation web appendix https://thesoftwaredesignlab.github.io/android-mutation/," 2019.

[112] SEMERU, "Mdroid+ repo at github https://gitlab.com/SEMERU-Code-Public/Android/Mutation/MDroidPlus," 2019.

[113] M. L.-V. Camilo Escobar-Velásquez. Mutapk, enabling mutation testing at apk level. [Online]. Available: https://thesoftwaredesignlab.github.io/MutAPK/

[114] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," ser. ICSE 2015, vol. 1, May 2015.

[115] R. Wiśniewsky and C. Tumbleson. Apktool, a tool for reverse engineering android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[116] "Uber apk signer. https://github.com/patrickfav/uber-apk-signer."

[117] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, April 2015, pp. 1–10.

[118] R. Jabbarvand and S. Malek, "mudroid: An energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 208–219.

[119] S. Roy Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *ASE 2015*.

[120] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verif. Reliab.*, vol. 7, no. 3, 1997.

[121] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Does choice of mutation tool matter?" *Software Quality Journal*, vol. 25, no. 3, Sep. 2017. [Online]. Available: https://doi.org/10.1007/s11219-016-9317-7

[122] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.4370040303

[123] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, April 2018.

[124] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," ser. ICSE 2014. New York, NY, USA: ACM, 2014.

[125] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, 1979.

[126] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Eng.*, vol. 3, no. 4, Jul. 1977.

[127] J. Zhang, M. Zhu, D. Hao, and L. Zhang, "An empirical study on the scalability of selective mutation testing," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 277–287.

[128] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," ser. ICST 2013.

[129] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 435–444.

[130] "Mutapk. https://github.com/TheSoftwareDesignLab/MutAPK."

[131] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *ASE'15*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89

[132] S. Jeong, S. Cha, and W. J. Lee, "Usage log-based testing of embedded software and identification of dependencies among environmental components," *IEICE TRANSACTIONS on Information and Systems*, vol. 104, no. 11, pp. 2011–2014, 2021.

[133] H. N. Silva, J. Prado Lima, S. R. Vergilio, and A. T. Endo, "A mapping study on mutation testing for mobile applications," *Software Testing, Verification and Reliability*, vol. 32, no. 8, p. e1801, 2022.

[134] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Mutation testing in the wild: findings from github," *Empirical Software Engineering*, vol. 27, no. 6, p. 132, 2022.

[135] J.-W. Lin, N. Salehnamadi, and S. Malek, "Route: Roads not taken in ui testing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–25, 2023.

[136] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 957–969.

[137] M. Polo-Usaola and I. Rodríguez-Trujillo, "Analysing the combination of cost reduction techniques in android mutation testing," *Software Testing, Verification and Reliability*, vol. 31, no. 7, p. e1769, 2021.

[138] E. H. Marinho, F. Ferreira, J. P. Diniz, and E. Figueiredo, "Evaluating testing strategies for resource related failures in mobile applications," *Software Quality Journal*, pp. 1–27, 2023.

[139] X. Xia, D. Lo, F. Zhu, X. Wang, and B. Zhou, "Software internationalization and localization: An industrial experience," in *2013 18th International Conference on Engineering of Complex Computer Systems*, July 2013, pp. 222–231.

[140] A. Alameer, S. Mahajan, and W. G. Halfond, "Detecting and localizing internationalization presentation failures in web applications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 202–212.

[141] A. Alameer and W. G. Halfond, "An empirical study of internationalization failures in the web," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 88–98.

[142] C. Escobar-Velásquez, M. Osorio-Riaño, J. Dominguez-Osorio, M. Arevalo, and M. Linares-Vásquez, "Itdroid: Internationalization of android apps. https://thesoftwaredesignlab.github.io/ITDroid/."

[143] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 477–487. [Online]. Available: https://doi.org/10.1145/2491411.2491428

[144] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[145] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 291–300.

[146] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings for software internationalization," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 353–363.

[147] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings in web applications," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882306

[148] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 516–536, April 2013.

[149] X. Wang, C. Chen, and Z. Xing, "Domain-specific machine translation with recurrent neural network for software localization," *Empirical Software Engineering*, Apr 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09702-z

[150] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do Developers Test Android Applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 613–622.

[151] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

[152] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 599–609. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635896

[153] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1013–1024. [Online]. Available: http://doi.acm.org.ezproxy.uniandes.edu.co:8080/10.1145/2568225.2568229

[154] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, June 2017.

[155] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, Sep. 2015.

[156] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond, "Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 215–226.

[157] Y. Arnatovich, H. B. K. Tan, S. Ding, K. Liu, and L. K. Shar, "Empirical Comparison of Intermediate Representations for Android Applications," in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School, 2014, pp. 205–210.

[158] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation," *IEEE Access*, vol. 6, pp. 12 382–12 394, 2018.

[159] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings*

of the 27th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[160] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 33–44.

[161] S. Liñán, L. Bello-Jiménez, M. Arévalo, and M. Linares-Vásquez, "Automated extraction of augmented models for android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 549–553.

[162] K. Jamrozik and A. Zeller, "Droidmate: A robust and extensible test generator for android," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2016, pp. 293–294.

[163] Y. Cao, G. Wu, W. Chen, and J. Wei, "Crawldroid: Effective model-based gui testing of android apps," in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, ser. Internetware '18. New York, NY, USA: ACM, 2018, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/3275219.3275238

[164] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. L. Vásquez, G. Bavota, C. Vendome, M. D. Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," *CoRR*, vol. abs/1802.04749, 2018. [Online]. Available: http://arxiv.org/abs/1802.04749

[165] Google. Bidirectionality. https://bit.ly/3c61pyd.

[166] ——. Support layout mirroring. https://bit.ly/2yDkx8T.

[167] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, "Can everyone use my app? an empirical study on accessibility in android apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 41–52.

[168] Google. android:ellipsize. https://bit.ly/2XEmwSN.

[169] R. Smith, "Distinct word length frequencies: distributions and symbol entropies," *Glottometrics*, vol. 23, pp. 7–22, 2012.

[170] A. F. Donoso Díaz, "Improving automated i18n testing of android apps," 2021.

[171] M. S. Andrade Vargas, "Towards automated repairment of internationalization issues for android apps," 2022-10-25.

[172] G. Ben, *Smali Wiki*, 2021, found at: https://github.com/JesusFreke/smali/wiki.

[173] "Support different languages and cultures |android," https://developer.android.com/training/basics/supporting-devices/languages, 2022.

[174] Google, “Offline first. https://developer.chrome.com/apps/offline_apps.”

[175] J. Archibald, “Offline cookbook. https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook.”

[176] G. Machado, “Mobile apps offline support. https://www.infoq.com/articles/mobile-apps-offline-support,” 2015.

[177] Google, “Network and battery best practices. http://goo.gl/vbiiV7.”

[178] M. Frank, B. Dong, A. Porter Felt, and D. Song, “Mining permission request patterns from android and facebook applications,” in *2012 IEEE 12th International Conference on Data Mining*, 2012, pp. 870–875.

[179] S. Mostafa, R. Rodriguez, and X. Wang, “Netdroid: Summarizing network behavior of android apps for network code maintenance,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 165–175.

[180] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, “A first look at traffic on smartphones,” in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 281–287. [Online]. Available: https://doi.org/10.1145/1879141.1879176

[181] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat, and W. Dabbous, “Network characteristics of video streaming traffic,” in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2079296.2079321

[182] S. K. Baghel, K. Keshav, and V. R. Manepalli, “An investigation into traffic analysis for diverse data applications on smartphones,” in *2012 National Conference on Communications (NCC)*, 2012, pp. 1–5.

[183] Hyo-Sik Ham and Mi-Jung Choi, “Applicaion-level traffic analysis of smartphone users using embedded agents,” in *2012 14th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2012, pp. 1–4.

[184] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Profiledroid: Multi-layer profiling of android applications,” in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, ser. Mobicom ’12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 137–148. [Online]. Available: https://doi.org/10.1145/2348543.2348563

[185] K. Fukuda, H. Asai, and K. Nagami, “Tracking the evolution and diversity in network usage of smartphones,” in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 253–266. [Online]. Available: https://doi.org/10.1145/2815675.2815697

[186] W. Nayam, A. Laolee, L. Charoenwatana, and K. Sripanidkulchai, "An analysis of mobile application network behavior," in *Proceedings of the 12th Asian Internet Engineering Conference*, ser. AINTEC '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 9–16. [Online]. Available: https://doi.org/10.1145/3012695.3012697

[187] M. Conti, Q. Q. Li, A. Maragno, and R. Spolaor, "The dark side(-channel) of mobile devices: A survey on network traffic analysis," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 2658–2713, 2018.

[188] M. Rapoport, P. Suter, E. Wittern, O. Lhotak, and J. Dolby, "Who you gonna call? analyzing web requests in android applications," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 80–90.

[189] H. Kuzuno and S. Tonami, "Signature generation for sensitive information leakage in android applications," in *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, 2013, pp. 112–119.

[190] Y. Song and U. Hengartner, "Privacyguard: A vpn-based platform to detect information leakage on android devices," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 15–26. [Online]. Available: https://doi.org/10.1145/2808117.2808120

[191] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "Recon: Revealing and controlling pii leaks in mobile network traffic," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 361–374. [Online]. Available: https://doi.org/10.1145/2906388.2906392

[192] Z. Cheng, X. Chen, Y. Zhang, S. Li, and Y. Sang, "Detecting information theft based on mobile network flows for android users," in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, 2017, pp. 1–10.

[193] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[194] X. Huang, A. Zhou, P. Jia, L. Liu, and L. Liu, "Fuzzing the android applications with http/https network data," *IEEE Access*, vol. 7, pp. 59 951–59 962, 2019.

[195] P. Gadient, M. Ghafari, M. Tarnutzer, and O. Nierstrasz, "Web apis in android through the lens of security," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 13–22.

[196] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, Jan. 2011. [Online]. Available: https://doi.org/10.1007/s10844-010-0148-x

[197] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 123–134. [Online]. Available: https://doi.org/10.1145/2594368.2594391

[198] T. Wei, C. Mao, A. B. Jeng, H. Lee, H. Wang, and D. Wu, "Android malware detection via a latent network behavior analysis," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012, pp. 1251–1258.

[199] C. Escobar-Velásquez, A. Mazuera-Rozo, C. Bedoya, M. Osorio-Riaño, M. Linares-Vásquez, and G. Bavota, "Online appendix. https://thesoftwaredesignlab.github.io/android-eventual-connectivity."

[200] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015.

[201] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 389–398. [Online]. Available: https://doi-org.ezproxy.uniandes.edu.co:8443/10.1109/ASE.2013.6693097

[202] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 352–361.

[203] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 341–352. [Online]. Available: http://doi.acm.org.ezproxy.uniandes.edu.co:8080/10.1145/2635868.2635903

[204] A. Mazuera-Rozo, C. Trubiani, M. Linares-Vásquez, and G. Bavota, "Investigating types and survivability of performance bugs in mobile apps," *Empirical Software Engineering*, 2019.

[205] A. Agrawal, B. Sodhi, and P. TV, "A multi-dimensional measure for intrusion: The intrusiveness quality attribute," in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, ser. QoSA '13. New York, NY, USA: ACM, 2013, pp. 63–68. [Online]. Available: http://doi.acm.org.ezproxy.uniandes.edu.co:8080/10.1145/2465478.2465497

[206] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 468–471.

[207] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 95–109.

[208] M. E. Joorabchi, M. Ali, and A. Mesbah, "Detecting inconsistencies in multi-platform mobile apps," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 450–460.

[209] M. Ali, M. E. Joorabchi, and A. Mesbah, "Same app, different app stores: A comparative study," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 79–90. [Online]. Available: https://doi-org.ezproxy.uniandes.edu.co:8443/10.1109/MOBILESoft.2017.3

[210] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 308–318.

[211] I. T. Mercado, N. Munaiah, and A. Meneely, "The impact of cross-platform development approaches for mobile applications from the user's perspective," in *Proceedings of the International Workshop on App Market Analytics*, ser. WAMA 2016. New York, NY, USA: ACM, 2016, pp. 43–49. [Online]. Available: http://doi.acm.org.ezproxy.uniandes.edu.co:8080/10.1145/2993259.2993268

[212] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets-X. New York, NY, USA: ACM, 2011, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070567

[213] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307661

[214] J. Zhang, A. Musa, and W. Le, "A comparison of energy bugs for smartphone platforms," in *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*, May 2013, pp. 25–30.

[215] Y. Liu, C. Xu, and S. C. Cheung, "Where has my battery gone? finding sensor related energy black holes in smartphone applications," in *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2013, pp. 2–10.

[216] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

[217] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 15–18.

[218] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 83–93. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771786

[219] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2639108.2639131

[220] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli, "A graph-based dataset of commit history of real-world android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, 2018.

[221] R. Coppola, L. Ardito, and M. Torchiano, "Characterizing the transition to kotlin of android apps: A study on f-droid, play store, and github," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, ser. WAMA 2019, 2019.

[222] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang, "Ar-miner: Mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 767–778. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568263

[223] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 14–24. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884818

[224] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 291–300.

[225] ——, "Crowdsourcing user reviews to support the evolution of mobile apps," *Journal of Systems and Software*, vol. 137, pp. 143–162, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217302807

[226] (2017) List of free and open-source android applications. [Online]. Available: https://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications

[227] (2017) open-source-android-apps. [Online]. Available: https://github.com/pcqpcq/open-source-android-apps

[228] K. Krippendorff, *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[229] Google, "Connectivity manager. https://developer.android.com/reference/android/net/ConnectivityManager."

[230] Android, "Manage network usage. https://developer.android.com/training/basics/network-ops/managing."

[231] ——, "Monitor network connectivity while the app is running. https://bit.ly/3np3gDU."

[232] ——, "Kotlin io. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-i-o.html."

[233] ——, "Improve app performance with kotlin coroutines. https://developer.android.com/kotlin/coroutines-adv."

[234] ——, "Running android tasks in background threads. https://developer.android.com/guide/background/threading."

[235] ——, "Running android tasks in background threads - using handlers. https://bit.ly/3pTknPC."

[236] ——, "Asynctask class. https://developer.android.com/reference/android/os/AsyncTask."

[237] ——, "Volley. https://developer.android.com/training/volley."

[238] I. Square, "Retrofit. https://square.github.io/retrofit/."

[239] J. Archibald, "The offline cookbook. https://web.dev/offline-cookbook."

[240] Android., "Lrucache. https://developer.android.com/reference/android/util/LruCache."

[241] Square., "Picasso. https://square.github.io/picasso/."

[242] B. Ruth., "Glide. https://bumptech.github.io/glide/."

[243] Firebase, “Firebase - access data offline. https://firebase.google.com/docs/firestore/manage-data/enable-offline.”

[244] Realm, “Realm for android. https://realm.io/blog/realm-for-android/.”

[245] Android, “Schedule tasks with workmanager. https://developer.android.com/topic/libraries/architecture/workmanager.”

[246] ——, “Set up requestqueue. https://developer.android.com/training/volley/requestqueue.”

[247] ——, “Guide to app architecture - cache data. https://bit.ly/3nqTwc4.”

[248] M. Nagappan and E. Shihab, “Future trends in software engineering research for mobile apps,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 21–32.

[249] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, “A survey of app store analysis for software engineering,” *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[250] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, “The impact of api change- and fault-proneness on the user ratings of android apps,” *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[251] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “Api change and fault proneness: A threat to the success of android apps,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013.

[252] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, “What are the characteristics of high-rated apps? a case study on free android applications,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 301–310.

[253] E. Noei, M. D. Syer, Y. Zou, A. E. Hassan, and I. Keivanloo, “A study of the relation of mobile device attributes with the user-perceived quality of android apps,” *Empirical Softw. Engg.*, 2017.

[254] N. S. A. A. Bakar and I. Mahmud, “Empirical analysis of android apps permissions,” in *2013 International Conference on Advanced Computer Science Applications and Technologies*, 2013, pp. 406–411.

[255] K. Olmstead and M. Atkinson, “Apps permissions in the google play store,” 2015.

[256] C. Escobar-Velásquez, A. Mazuera-Rozo, C. Bedoya, M. Osorio-Riaño, M. Linares-Vásquez, and G. Bavota, “Studying eventual connectivity issues in android apps,” *Empirical Software Engineering*, 2022.

[257] Google, “Connect to the network.” [Online]. Available: https://developer.android.com/training/basics/network-ops/connecting

[258] ——, “Permissions on android.” [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[259] A. K. Jha, S. Lee, and W. J. Lee, “Developer mistakes in writing android manifests: An empirical study of configuration errors,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 25–36.

[260] J. Archibald, “The offline cookbook.” [Online]. Available: https://web.dev/offline-cookbook/

[261] C. Escobar-Velásquez, M. Linares-Vásquez, G. Bavota, M. Tufano, K. P. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, “Enabling mutant generation for open- and closed-source android apps,” *IEEE Transactions on Software Engineering*, 2020.

[262] Google, “Processes and threads overview.” [Online]. Available: https://developer.android.com/guide/components/processes-and-threads

[263] E. Hellman, *Android Programming : Pushing the Limits*. Chichester, West Sussex, United Kingdom: John Wiley & Sons Ltd, 2014.

[264] Google, “Connectivity for billions.” [Online]. Available: https://developer.android.com/docs/quality-guidelines/build-for-billions/connectivity#network-offline

[265] “Android development guides. https://developer.android.com/guide.”

[266] “Okhttp. https://square.github.io/okhttp/.”

[267] “Httpurlconnection. https://developer.android.com/reference/java/net/HttpURLConnection.”

[268] A. Mazuera-Rozo, C. Escobar-Velásquez, J. Espitia-Acero, M. Linares-Vásquez, and G. Bavota, “Replication package.” [Online]. Available: https://sites.google.com/view/conan-ecn

[269] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, “Static analysis of android apps: A systematic literature review,” *Information and Software Technology*, 2017.

[270] M. T. Azim, I. Neamtiu, and L. M. Marvel, “Towards self-healing smartphone software via automated patching,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 623–628. [Online]. Available: https://doi.org/10.1145/2642937.2642955

[271] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '14, 2014.

[272] W. Xiong, S. Chen, Y. Zhang, M. Xia, and Z. Qi, "Reproducible interference-aware mobile testing," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 36–47.

[273] Google, "Monkey," https://developer.android.com/studio/test/monkey, [Online; accessed 6-June-2019].

[274] W. Zhao, Z. Ding, M. Xia, and Z. Qi, "Systematically testing and diagnosing responsiveness for android apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 449–453.

[275] L. Panizo, A. Díaz, and B. García, "Model-based testing of apps in real network scenarios," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 2, pp. 105–114, Apr. 2019. [Online]. Available: https://doi.org/10.1007/s10009-019-00518-2

[276] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in android applications," in *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*, 2013, pp. 1–6.

[277] Google, "Improve your code with lint checks." [Online]. Available: https://developer.android.com/studio/write/lint

[278] ——, "Lint issue index," http://googlesamples.github.io/android-custom-lint-rules/checks/index.md.html.

[279] Gradle, "The gradle wrapper," https://docs.gradle.org/current/userguide/gradle_wrapper.html.

[280] Google, "Android developers," https://groups.google.com/g/android-developers.

[281] ——, "Manage network usage." [Online]. Available: https://developer.android.com/training/basics/network-ops/managing

[282] Microsoft, "Synchronous i/o antipattern - performance antipatterns for cloud apps." [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/antipatterns/synchronous-io/

[283] Mozilla, "Synchronous and asynchronous requests - web apis: Mdn." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

[284] MITRE, "Common weakness enumeration - error conditions, return values, status codes." [Online]. Available: https://cwe.mitre.org/data/definitions/389.html

[285] ——, “Common weakness enumeration - empty exception block.” [Online]. Available: https://cwe.mitre.org/data/definitions/1069.html

[286] ——, “Common weakness enumeration - empty code block.” [Online]. Available: https://cwe.mitre.org/data/definitions/1071.html

[287] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, “Efficiently manifesting asynchronous programming errors in android apps,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018.

[288] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli, “A graph-based dataset of commit history of real-world android apps,” in *Int. Conference on Mining Software Repositories (MSR)*, 2018, pp. 30–33.

[289] R. Coppola, L. Ardito, and M. Torchiano, “Characterizing the transition to kotlin of android apps: A study on f-droid, play store, and github,” in *Int. Workshop on App Market Analytics (WAMA)*, 2019, pp. 8–14.

[290] “F-droid. https://f-droid.org/.”

[291] “Android user guide.” [Online]. Available: http://googlesamples.github.io/android-custom-lint-rules/user-guide.html

[292] “Android lint api guide.” [Online]. Available: http://googlesamples.github.io/android-custom-lint-rules/api-guide.md.html

[293] B. Rosner, *Fundamentals of Biostatistics*, 7th ed. Brooks/Cole, Boston, MA, 2011.

[294] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, “Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 399–410.

[295] S. Zein, N. Salleh, and J. Grundy, “A systematic mapping study of mobile application testing techniques,” *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300140

[296] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino, “Automated functional testing of mobile applications: a systematic mapping study,” *Software Quality Journal*, vol. 27, no. 1, pp. 149–201, 2019. [Online]. Available: https://doi.org/10.1007/s11219-018-9418-6

[297] SeleniumHQ. Selenium. [Online]. Available: https://www.selenium.dev/

[298] BrowserStack. Browserstack. [Online]. Available: https://www.browserstack.com

[299] B. J. A. Chow. (2015) Octopus to the rescue: The fascinating world of inter-app communications at uber engineering. [Online]. Available: https://eng.uber.com/rescued-by-octopus/

[300] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *2012 7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 29–35.

[301] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: towards getting there in an industrial case," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 253–262.

[302] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130. [Online]. Available: https://doi.org/10.1145/3468264.3468620

[303] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 103–114.

[304] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 105–115.

[305] T. A. T. Vuong and S. Takada, *A Reinforcement Learning Based Approach to Automated Testing of Android Applications*. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–37. [Online]. Available: https://doi.org/10.1145/3278186.3278191

[306] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, *Reinforcement Learning for Android GUI Testing*. New York, NY, USA: Association for Computing Machinery, 2018, p. 2–8. [Online]. Available: https://doi.org/10.1145/3278186.3278187

[307] T. Vuong and S. Takada, "Semantic analysis for deep q-network in android gui testing," in *Proceedings - SEKE 2019*, ser. Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE. Knowledge Systems Institute Graduate School, Jan. 2019, pp. 123–128, 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019 ; Conference date: 10-07-2019 Through 12-07-2019.

[308] Y. Köroglu and A. Sen, "Reinforcement learning-driven test generation for android GUI applications using formal specifications," *CoRR*, vol. abs/1911.05403, 2019. [Online]. Available: http://arxiv.org/abs/1911.05403

[309] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[310] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 160–167.

[311] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *CoRR*, vol. abs/2101.02636, 2021. [Online]. Available: https://arxiv.org/abs/2101.02636

[312] H. N. Yasin, S. H. A. Hamid, and R. J. Raja Yusof, "Droidbotx: Test case generation tool for android applications using q-learning," *Symmetry*, vol. 13, no. 2, 2021. [Online]. Available: https://www.mdpi.com/2073-8994/13/2/310

[313] E. Collins, A. Neto, A. Vincenzi, and J. Maldonado, *Deep Reinforcement Learning Based Android Application GUI Testing*. New York, NY, USA: Association for Computing Machinery, 2021, p. 186–194. [Online]. Available: https://doi.org/10.1145/3474624.3474634

[314] W. Ravelo-Méndez, C. Escobar-Velásquez, and M. Linares-Vásquez, "Kraken online appendix https://thesoftwaredesignlab.github.io/android-mutation/," 2022.

[315] ——, "Kraken-mobile: Cross-device interaction-based testing of android apps," in *ICSME'19*.

[316] O. foundation. Webdriver.io. [Online]. Available: https://webdriver.io/

[317] Marak. faker.js. [Online]. Available: https://bit.ly/30KyBeX

[318] StatCounter, "Mobile operating system market share worldwide," 2023. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[319] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[320] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[321] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[322] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[323] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011, pp. 77–83.

[324] Z. Shen, K. Yang, Z. Xi, J. Zou, and W. Du, "Deepapp: A deep reinforcement learning framework for mobile application usage prediction," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[325] D. Toyama, P. Hamel, A. Gergely, G. Comanici, A. Glaese, Z. Ahmed, T. Jackson, S. Mourad, and D. Precup, "Androidenv: A reinforcement learning platform for android," *CoRR*, vol. abs/2105.13231, 2021. [Online]. Available: https://arxiv.org/abs/2105.13231

[326] J. N. Foerster, "Deep multi-agent reinforcement learning," Ph.D. dissertation, University of Oxford, 2018.

[327] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," 2017. [Online]. Available: https://arxiv.org/abs/1706.02275

[328] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, pp. 1–12.

[329] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. [Online]. Available: https://arxiv.org/abs/1606.01540

[330] W. W. Cohen, P. Ravikumar, S. E. Fienberg *et al.*, "A comparison of string distance metrics for name-matching tasks." in *IIWeb*, vol. 3, 2003, pp. 73–78.

[331] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130. [Online]. Available: https://doi.org/10.1145/3468264.3468620

[332] R. Shier, "Statistics: 2.3 the mann-whitney u test," *Mathematics Learning Support Centre. Last accessed*, vol. 15, p. 2013, 2004.

[333] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[334] M. R. Hess and J. D. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances," in *annual meeting of the American Educational Research Association*, vol. 1. Citeseer, 2004.

[335] G. Petrovic and M. Ivankovic, "State of mutation testing at google," in *Proceedings of the 40th International Conference on Software Engineering 2017 (SEIP)*, 2018.

[336] L. Bello-Jiménez, C. Escobar-Velásquez, A. Mojica-Hanke, S. Cortés-Fernández, and M. Linares-Vásquez, "Hall-of-apps: The top android apps metadata archive," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 568–572.

[337] A. Mojica-Hanke, L. Bello-Jiménez, C. Escobar-Velásquez, and M. Linares-Vásquez, "Crème de la crème. investigating metadata and survivability of top android apps," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 469–480.

[338] A. Mazuera-Rozo, C. Escobar-Velásquez, J. Espitia-Acero, D. Vega-Guzmán, C. Trubiani, M. Linares-Vásquez, and G. Bavota, "Taxonomy of security weaknesses in java and kotlin android apps," *Journal of Systems and Software*, vol. 187, p. 111233, 2022.

[339] S. Naranjo-Puentes, C. Escobar-Velásquez, C. Vendome, and M. Linares-Vásquez, "A preliminary study on accessibility of augmented reality features in mobile apps," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 454–458.

[340] NISO. (2023) Implementing credit. [Online]. Available: https://credit.niso.org/implementing-credit/