PhD-FSTM-2021-018
The Faculty of Science, Technology and Medicine

Dissertation

Presented on the 01/04/2021 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Pingfan KONG
Born on 5th March 1991 in Hefei, China

Taming Android App Crashes

Dissertation Defense Committee

Dr. Jacques Klein, Dissertation Supervisor
Associate Professor, Université du Luxembourg, Luxembourg

Dr. Tegawendé Bissyandé, Chairman
Associate Professor, Université du Luxembourg, Luxembourg

Dr. Li Li, Vice Chairman
Assistant Professor, Monash University, Australia

Dr. Leonardo Mariani
Professor, University of Milano–Bicocca, Italy

Dr. Ting Su
Professor, East China Normal University, China

Abstract

App crashes constitute an important deterrence for app adoption in the android ecosystem. Yet, Android app developers are challenged by the limitation of test automation tools to ensure that released apps are free from crashes. In recent years, researchers have proposed various automation approaches in the literature. Unfortunately, the practical value of these approaches have not yet been confirmed by practitioner adoption. Furthermore, existing approaches target a variety of test needs which are relevant to different sets of problems, without being specific to app crashes.

Resolving app crashes implies a chain of actions starting with their reproduction, followed by the associated fault localization, before any repair can be attempted. Each action however, is challenged by the specificity of Android. In particular, some specific mechanisms (e.g., callback methods, multiple entry points, etc.) of Android apps require Android-tailored crash-inducing bug locators. Therefore, to tame Android app crashes, practitioners are in need of automation tools that are adapted to the challenges that they pose. In this respect, a number of building blocks must be designed to deliver a comprehensive toolbox.

First, the community lacks well-defined, large-scale datasets of real-world app crashes that are reproducible to enable the inference of valuable insights, and facilitate experimental validations of literature approaches. Second, although bug localization from crash information is relatively mature in the realm of Java, state-of-the-art techniques are generally ineffective for Android apps due to the specificity of the Android system. Third, given the recurrence of crashes and the substantial burden that they incur for practitioners to resolve them, there is a need for methods and techniques to accelerate fixing, for example, towards implementing Automated Program Repair (APR).

Finally, the above chain of actions is for curative purposes. Indeed, this "reproduction, localization, and repair" chain aims at correcting bugs in released apps. Preventive approaches, i.e., approaches that help developers to reduce the likelihood of releasing crashing apps, are still absent. In the Android ecosystem, developers are challenged by the lack of detailed documentation about the complex Android framework API they use to develop their apps. For example, developers need support for precisely identifying which exceptions may be triggered by APIs. Such support can further alleviate the challenge related to the fact that the condition under which APIs are triggered are often not documented.

In this context, the present dissertation aims to tame Android crashes by contributing to the following four building blocks:

- *Systematic Literature Review on automated app testing approaches*: We aim at providing a clear overview of the state-of-the-art works around the topic of Android app testing, in an attempt to highlight the main trends, pinpoint the main methodologies applied and enumerate the challenges faced by the Android testing approaches as well as the directions where the community effort is still needed. To this end, we conduct a Systematic Literature Review (SLR) during which we eventually identified 103 relevant research papers published in leading conferences and journals until 2016. Our thorough examination of the relevant literature has led to several findings and highlighted the challenges that Android testing researchers should strive to address in the future. After that, we further propose a few concrete research directions where

testing approaches are needed to solve recurrent issues in app updates, continuous increases of app sizes, as well as the Android ecosystem fragmentation.

- *Locating Android app crash-inducing bugs*: We perform an empirical study on 500 framework-specific crashes from an open benchmark. This study reveals that 37 percent of the crash types are related to bugs that are outside the crash stack traces. Moreover, Android programs are a mixture of code and extra-code artifacts such as the Manifest file. The fact that any artifact can lead to failures in the app execution creates the need to position the localization target beyond the code realm. We propose ANCHOR, a two-phase suspicious bug location suggestion tool. ANCHOR specializes in finding crash-inducing bugs outside the stack trace. ANCHOR is lightweight and source code independent since it only requires the crash message and the apk file to locate the fault. Experimental results, collected via cross-validation and in-the-wild dataset evaluation, show that ANCHOR is effective in locating Android framework-specific crashing faults.
- *Mining Android app crash fix templates*: We propose a scalable approach, CraftDroid, to mine crash fixes by leveraging a set of 28 thousand carefully reconstructed app lineages from app markets, without the need for the app source code or issue reports. We develop a replicative testing approach that locates fixes among app versions which output different runtime logs with the exact same test inputs. Overall, we have mined 104 relevant crash fixes, further abstracted 17 fine-grained fix templates that are demonstrated to be effective for patching crashed apks. Finally, we release ReCBench, a benchmark consisting of 200 crashed apks and the crash replication scripts, which the community can explore for evaluating generated crash-inducing bug patches.
- *Documenting framework APIs' unchecked exceptions*: We propose Afuera, an automated tool that profiles Android framework APIs and provides information on when they can potentially trigger unchecked exceptions. Afuera relies on a static-analysis approach and a dedicated algorithm to examine the entire Android framework. With Afuera, we confirmed that 26 739 unique unchecked exception instances may be triggered by invoking 5 467 (24%) Android framework APIs. Afuera further analyzes the Android framework to inform about which parameter(s) of an API method can potentially be the cause of the triggering of an unchecked exception. To that end, Afuera relies on fully automated instrumentation and taint analysis techniques. Afuera is run to analyze 50 randomly sampled APIs to demonstrate its effectiveness. Evaluation results suggest that Afuera has perfect true positive rate. However, Afuera is affected by false negatives due to the limitation of state-of-the-art taint analysis techniques.

Have a greater perspective of things that are greater than myself.

Acknowledgements

This dissertation would not have been possible without the support of many people who in one way or another have contributed and extended their precious knowledge and experience in my PhD studies. It is my pleasure to express my gratitude to them.

First of all, I would like to express my deepest thanks to my supervisor, Assoc. Prof. Jacques Klein, who has given me this great opportunity to come across continents to pursue my doctoral degree. He has always trusted and supported me with his great kindness throughout my whole PhD journey.

Second, I am equally grateful to my daily advisers, Assoc. Prof. Tegawendé Bissyandé and Asst. Prof. Li Li, who have introduced me into the world of Android. Since then, working in this field is just joyful for me. They have taught me how to perform research, how to write technical papers, and how to conduct fascinating presentations. Their dedicated guidance has made my PhD journey a fruitful and fulfilling experience. I am very happy for the friendship we have built up during the years.

Third, I would like to extend my thanks to all my co-authors including Prof. Yves Le Traon, Dr. Jun Gao, Dr. Kui Liu, Dr. Kevin Allix, Dr. Médéric Hurier, Dr. Alexander Bartel, Mr. Timothée Riom, Ms. Yanjie Zhao, and Mr. Jordan Samhi for their valuable discussions and collaborations.

I would like to thank all the members of my PhD defense committee, including Prof. Leonardo Mariani, Prof. Ting SU, my supervisor Assoc. Prof. Jacques Klein, and my daily advisers Assoc. Prof. Tegawendé Bissyandé and Asst. Prof. Li Li. It is my great honor to have them in my defense committee and I appreciate very much their efforts to examine my dissertation and evaluate my PhD work.

I would like to also express my great thanks to all the friends that I have made in the Grand Duchy of Luxembourg for the memorable moments that we have had. More specifically, I would like to thank all the team members of TRuX and SerVal at SnT for the great coffee breaks and interesting discussions. I would also like to thank the team under Asst. Prof. Li Li in Monash University for the insightful discussions.

Finally, I would like to thank my wife and my daughter for bringing the everlasting joy and happiness to my everyday life.

Pingfan Kong
University of Luxembourg
April 2021

Contents

List of figures xi

List of tables xiii

Contents xiii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Challenges . . . 2
  - 1.2.1 Test Automation Challenges . . . 3
  - 1.2.2 Program Repair Challenges . . . 3
  - 1.2.3 Android Framework Documentation Challenges . . . 4
- 1.3 Contributions . . . 5
- 1.4 Roadmap . . . 6

2 Background 7
- 2.1 Android . . . 8
  - 2.1.1 Architecture . . . 8
  - 2.1.2 API Level Evolution . . . 9
  - 2.1.3 Manifestation . . . 10
- 2.2 App Crash . . . 10
  - 2.2.1 Android Debug Bridge . . . 10
  - 2.2.2 Logcat . . . 11
  - 2.2.3 Stack Trace . . . 11
- 2.3 Static Analysis . . . 12
  - 2.3.1 Call Graph Construction . . . 12
  - 2.3.2 Taint Analyzer . . . 13
- 2.4 Datasets . . . 13
  - 2.4.1 F-Droid . . . 13
  - 2.4.2 AndroZoo . . . 13
  - 2.4.3 Lineage . . . 13

3 Automated Testing of Android Apps: A Systematic Literature Review 15
- 3.1 Overview . . . 16
- 3.2 Methodology of This SLR . . . 18
  - 3.2.1 Initial Research Questions . . . 18
  - 3.2.2 Search Strategy . . . 19
  - 3.2.3 Exclusion Criteria . . . 20
  - 3.2.4 Review Protocol . . . 21
- 3.3 Primary Publications Selection . . . 21
- 3.4 Taxonomy of Android Testing Research . . . 23
- 3.5 Literature Review . . . 24
  - 3.5.1 What concerns do the approaches focus on? . . . 24
  - 3.5.2 Which Test Levels are Addressed? . . . 27
  - 3.5.3 How are the Test Approaches Built? . . . 28
  - 3.5.4 To What Extent are the Approaches Validated? . . . 32

3.6 Discussion . . . 38
3.6.1 Trend Analysis . . . 38
3.6.2 Evaluation of Authors . . . 39
3.6.3 Research Output Usability . . . 40
3.6.4 Open Issues and Future Challenges . . . 41
3.6.5 New Research Directions . . . 42
3.7 Threats to Validity . . . 43
3.8 Related Work . . . 43
3.9 Summary . . . 44

4 Anchor: Locating Android Framework-specific Crashing Faults 49
4.1 Overview . . . 51
4.2 Background . . . 52
4.2.1 Android App Crash Stack Trace . . . 52
4.2.2 Callback-based and Event-driven Mechanism . . . 53
4.2.3 Android APK File Format . . . 54
4.3 Motivating Example . . . 54
4.4 Empirical Study on Fault Locations . . . 56
4.4.1 Dataset Construction . . . 56
4.4.2 Ground Truth & Results . . . 57
4.4.3 Category A: in Stack Trace . . . 58
4.4.4 Category B: out of Stack Trace, in the Code . . . 58
4.4.5 Category C: out of Stack Trace, out of Code . . . 59
4.5 Ranking Suspicious Locations . . . 60
4.5.1 Phase 1: Categorization . . . 60
4.5.2 Phase 2: Localization . . . 60
4.6 Study Setup . . . 63
4.6.1 Research questions . . . 63
4.6.2 Metrics . . . 63
4.6.3 Cross-validation . . . 64
4.6.4 Feature Selection . . . 64
4.7 Study Results . . . 64
4.7.1 RQ1: Effectiveness of Categorization . . . 64
4.7.2 RQ2: Effectiveness of Localization . . . 65
4.7.3 RQ3: Overall Performance of Anchor . . . 65
4.7.4 RQ4: Performance in the Wild . . . 65
4.8 Disscussion . . . 66
4.8.1 Comparing Anchor with other Locators . . . 66
4.8.2 Developer Effort for Locating Bugs . . . 67
4.9 Threats to Validity . . . 67
4.9.1 Internal Threats . . . 67
4.9.2 External Threats . . . 67
4.10 Related Work . . . 67
4.11 Summary . . . 68

5 Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems 71
5.1 Overview . . . 72
5.2 Motivating Example . . . 73
5.3 Study Design . . . 74
5.3.1 Phase I: Fix Mining . . . 74
5.3.2 Phase II: Fix Grouping and Fix Template Abstraction . . . 77
5.3.3 Patching Crashed Apks . . . 78

5.4 Dataset and Statistics . . . 79
5.4.1 Crash Fixes from Lineages (from Phase I) . . . 79
5.4.2 Fix Buckets & Fix Templates (from Phase II) . . . 81
5.4.3 ReCBench for Evaluating Bug Patches . . . 82
5.5 Study Results . . . 83
5.5.1 Explored Crashes vs. Reported Crashes . . . 83
5.5.2 Benchmarks for Evaluating Patches . . . 83
5.5.3 Evaluating Fix Templates on ReCBench . . . 84
5.6 Threats to Validity . . . 86
5.7 Related Work . . . 87
5.8 Summary and Future Work . . . 88

6 Afuera: Automatically Documenting Android Framework APIs for Unchecked Exceptions 89
6.1 Overview . . . 90
6.2 Motivation and Background . . . 91
6.2.1 Motivation . . . 91
6.2.2 Background on Android Framework-specific Exceptions . . . 93
6.3 Study Design for `Afuera` . . . 94
6.3.1 Module I: Profile UE-API Methods . . . 94
6.3.2 Module II: Pinpoint Parameters . . . 97
6.4 Study Results . . . 99
6.4.1 Implementation . . . 99
6.4.2 Key Characteristics of UE-API Methods . . . 100
6.4.3 UE-APIs Usage in Real-world Android Apps . . . 101
6.4.4 Effectiveness of `Afuera` Module II . . . 103
6.5 Threats to Validity . . . 104
6.6 Related Work . . . 104
6.7 Summary . . . 105

7 Conclusions and Future Work 107
7.1 Conclusions . . . 108
7.2 Future Work . . . 108
7.2.1 Verification of Parameter Constraints . . . 109
7.2.2 Fuzzing for App Crashes . . . 109
7.2.3 Improve the Accuracy of Bug Locating . . . 109

Bibliography 113

List of figures

1 Introduction **2**
1.1 Roadmap of This Dissertation. 6

2 Background **8**
2.1 The Android Architecture. 8
2.2 The Android Version Evolution (Data updated in March, 2021). 9
2.3 The Formation of App Lineage. 14

3 Automated Testing of Android Apps: A Systematic Literature Review **16**
3.1 Process of testing Android apps. 17
3.2 Process of the SLR. 18
3.3 Word Cloud based on the Venue Names of Selected Primary Publications. 22
3.4 The number of publications in each year. 22
3.5 Distribution of examined publications through published venue types and domains. . 23
3.6 Taxonomy of Android App Testing. 23
3.7 Breakdown of examined publications regarding their applied testing types. 30
3.8 Venn Diagram of Testing Environment. 31
3.9 The distribution of the number of tested apps (outliers are removed). 33
3.10 Trend of Testing Types. 38
3.11 Trend of Testing Levels. 38
3.12 Trend of Testing Methods. 39
3.13 Trend of Testing Targets and Objectives. 40
3.14 Trend in community authors. “New Authors” and “Stayed Authors" indicate the number of authors that enter the field (no relevant publications before) and have stayed in the field (they will keep publishing in the following years). 40

4 Anchor: Locating Android Framework-specific Crashing Faults **51**
4.1 Crash Stack Trace of app Sailer’s Log Book. 53
4.2 Call Graph Comparison between General Java Program (left) and Android App (right), inspired from [244] 53
4.3 Crash of Transistor. 55
4.4 Localization Process for Category C. 63
4.5 F Measure v.s. Selected Features. 64

5 Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems **72**
5.1 Overview of CraftDroid. 75
5.2 Patching and Evaluation. 78
5.3 Distribution of Total, Installed and Crashed Numbers of Apks in Lineages. 80
5.4 Count of Lineages Crashes per Testing Strategy. 80
5.5 Bucket Count for Exceptions. 82

6 Afuera: Automatically Documenting Android Framework APIs for Unchecked Exceptions **90**
6.1 Motivating Example. 92
6.2 Java *Throwable* Type and its Sub-classes. 93
6.3 Checked Exception 94
6.4 Unchecked Exception 94

6.5 Workflow of `Afuera`. 95
6.6 Exceptional Control Flow. 96
6.7 Signalers and UE-API methods Statistics. 101
6.8 Distribution of the UE-API Usages per UE type (left) and per package (right) . . . 102
6.9 UE-API Usage Yearly Evolution. 102

List of tables

2 Background **8**
2.1 Contents in an APK File. 10

3 Automated Testing of Android Apps: A Systematic Literature Review **16**
3.1 Search Keywords 19
3.2 Summary of the selection of primary publications. 21
3.3 Test objectives in the literature. 25
3.4 Test targets in the literature. 34
3.5 Recurrent testing phases. 35
3.6 Test method employed in the literature. 36
3.7 Common test types. 36
3.8 Summary of basic tools that are frequently leveraged by other testing approaches. . 37
3.9 Assessment Metrics (e.g., for Coverage, Accuracy). 37
3.10 The Full List of Examined Publications. 45

4 Anchor: Locating Android Framework-specific Crashing Faults **51**
4.1 Categories of Fault Locations in Android apps 57
4.2 Crash Causes of Categorie C 60
4.3 Effectiveness of Categorization (Phase 1) 64
4.5 Localization Performance 65
4.6 Overall Performance of Anchor 65
4.7 Categorization on an independent dataset. 66
4.9 Recall@k and MRR on an independent dataset. 66

5 Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems **72**
5.1 Fix Templates 81
5.2 Buckets Count between Fan et al. and `CraftDroid` 83
5.3 Comparison among benchmarks 84
5.4 Patch Evaluation on RecBench 85

6 Afuera: Automatically Documenting Android Framework APIs for Unchecked Exceptions **90**
6.1 Confusion Matrix of Evaluation Results. 103

1 Introduction

In this chapter, we first introduce the motivation for taming Android app crashes. Then, we summarize the challenges for both researchers and developers in taming such crashes. Finally, we present the contributions and roadmap of this dissertation.

Contents

1.1 Motivation **2**
1.2 Challenges **2**
1.2.1 Test Automation Challenges 3
1.2.2 Program Repair Challenges 3
1.2.3 Android Framework Documentation Challenges 4
1.3 Contributions **5**
1.4 Roadmap **6**

1.1 Motivation

Android smart devices have become pervasive after gaining tremendous popularity in recent years. The app distribution ecosystem around the official store and other alternative stores further attract users to find apps. However, these apps do not always function correctly as designed. Among such malfunctioning cases, app crashes are a recurrent phenomenon in the Android ecosystem [240]. They generally cause damages to the app reputation and beyond that to the provider's brand [83]. Apps with too many crashes can even be simply uninstalled by annoyed users. They could also receive bad reviews which limit their adoption by new users. Too many apps crashes could also be detrimental to specific app markets that do not provide mechanisms to filter out low-quality apps w.r.t. proneness to crash.

It is thus of utmost importance to ensure that Android apps are sufficiently tested before they are released on the market. However, manual testing is often laborious, time-consuming and error-prone. Therefore, the ever-growing complexity and quantity of Android apps call for scalable, robust and trustworthy automated testing solutions. Despite the large set of testing approaches proposed over the past years, it takes a systematic review to find the most suitable approaches that may expose app crashes efficiently. Given the enormous number of apps and their quick evolution of version, these approaches also need to be great in code coverage, independent of app source code, and free from massive instrumentation.

If the apps crashes are successfully exposed before release, the natural follow-up objective would be to fix such crash-inducing bugs and provide the users with crash-free apps. This calls for a toolchain of actions. The starting action is to locate the bugs which caused the crashes. However, app crashing is a dynamic behavior which often leave nothing but traces of execution. Developers are still required to examine the app logic to understand the root cause of the crashes and consequently know which part of the apps caused the crashes. Automating this action and providing assistance to developers would boost the bug resolution process. After locating the crash-inducing bugs, next action is to suggest fixes for such bugs. However, apps may crash from various bad programming practices. It is often extremely time- and effort-consuming for developers to fix such bugs by hand. Recently, Automatic Program Repair (APR) techniques have also been proposed to automatically fix app crashes [223]. Modern APR techniques first apply different fixing templates to patch buggy programs and evaluate the patched candidates to select truly fixed programs. However, existing APR methods on Android app crashes are greatly limited by insufficient fixing templates. Therefore, generating these templates has become an imperative step in crash-inducing bug fixing. Unfortunately, there is a need for comprehensive datasets of true crash fixes to abstract such fixing templates. This is a major obstacle which has prevented extensive research on crash repair within the Android community.

Despite the curative toolchain consisting of testing for, locating and fixing such app crashes, it is equally important to prevent developers from unknowingly writing such bugs that will cause the app crashes. Intuitively, fewer bugs introduced to the apps means the burden of curative measures are eased. To make developers avoid writing crashing-prone apps, it is better to warn them about programming hazards that may cause app crashes. As existing studies pointed out, many crashes arise from exceptions signaled from the huge set of framework APIs. However, these potential exceptions are not well documented in the official Android API references. Thorough documentation of these programming hazards is needed to cut the likelihood of providing crash-inducing bugs.

1.2 Challenges

In this section, we present the technical challenges we face when taming Android app crashes. Specifically, we discuss the challenges imposed by evaluating test automation approaches, the

challenges faced to program repair, as well as the challenges for understanding Android framework documentation.

1.2.1 Test Automation Challenges

Android apps are often shipped to the users with potential crashes since they are not fully tested for crash-inducing bugs. Given the enormous total number of apps and the ever-growing size and more complex logic of each app, human based testing is challenged to find these bugs efficiently. Therefore, test automation for Android apps attracts more attention in recent years from the research community. However, test automation are still challenged to find all app crashes before releasing. In this subsection, we detail such challenges.

- **Instrumentation.** In order to handle massive number of apps under test, test automation solutions need to be light-weight. However, many existing test automation solutions require the instrumentation of Android apps and/or Android framework. These solutions insert logging statements into the original app and/or Android framework to understand the execution of the apps and adjust their test input generation strategy accordingly. Instrumentation often incurs these solutions inefficient and will not expose enough app crashes given limited time budget. Moreover, the instrumentation may even cause regression, i.e., the apps may crash from the instrumented code.
- **Code coverage.** Test automation tools are often challenged by low code coverage. Indeed, modern Android apps are often large in terms of lines of code. They often contain many components. Some components require specific conditions being satisfied in order to be started. It is challenging, if not impossible, for test automation tools to guarantee full coverage. Therefore, they could miss crash-inducing bugs hiding in uncovered code. These bugs will later be exposed when they are already shipped to the users, but already too late since they greatly hindered the user experience and fondness for the apps.
- **Source code dependence.** To automate the test sequence generation, many testing approaches require the app source code to be known. However, these approaches are challenged by the fact that most commercial Android apps do not disclose their source code. For example, app hosting platforms may wish to apply testing techniques to filter out crash-prone apps. However, since the hosted apps are often closed-source, such testing approaches are not usable.
- **Compatibility.** Android framework is a quickly evolving system with many customized distributions installed on different devices. Some crash-inducing bugs are only valid with specific distribution, running on specific devices. However, most test automation tools test Android apps on limited types of devices or API versions. Some testing tools only work on emulators instead of real devices. Therefore, such tools are challenged in exposing crash-inducing bugs closely related to incompatibility.

To summarize, the above-mentioned limitations of automated Android app testing tools challenge our task of exposing Android app crashes for analysis. Given the large selection of such testing tools, it is important, while equally difficult, to evaluate these tools and pick the most suitable ones for exposing Android app crashes.

1.2.2 Program Repair Challenges

After exposing Android app crashes, the succeeding step for taming them is applying program repair techniques to fix crashed apps. However, several facts challenge the process of program repair. First, repairing crash-inducing bugs requires their precise location in the code. Second, repairing bugs can be extremely time- and labor-intensive. Therefore, automating program repair (APR) is in demand [223]. However, APR for Android app crashes is challenged by insufficient fixing templates. We next detail these two challenges.

- **Bug Location.** To tame the Android app crashes, it is unavoidable to know where to fix them. Specifically, it is important to locate the line of code containing the bug that triggered the crash. Although Android apps are mainly developed in Java, traditional Java-based fault localization tools are challenged to be immediately usable on Android apps. The main reason is that these locators assume single entry point like Java programs. However, Android apps are even-driven and callback-based, which means that there exist multiple entry points. Moreover, the Android apps are not just code, they are a mixture of code and extra-code artifacts that both contribute to the app logic. However, existing locators created for Java targets only code. Therefore, it is challenging to locate crash-inducing bugs in Android apps.
- **Fix Templates.** Although automatic program repairing techniques have been borrowed to also fix Android apps [223]. They are challenged by the fact that there is a lack of sufficient fixing templates. Researchers have invented crawlers to retrieve fixes from open source app projects. However, there are still several concerns. First, there is a threat to external validity as only open source apps can be concerned when obtaining fixes. The dataset is not representative since a number of crashes may never be reported in issue tracking systems, and yet fixes have been applied to address them. Open source apps often deal with simple functionality scenarios, and generally, have a smaller code base compared to commercial ones. Thus there may be fewer occurrences of crashes. Second, the collection process of crash fixes is not scalable. Although researchers build crawlers to analyze GitHub repositories and select potential closed issues, they must manually verify in the code that the issue is real and that the provided fix is indeed related to the announced crash. Finally, the fix collection process invented for open source apps cannot be replicated in commercial development settings, which do not provide useful information on bug reports, or means to reproduce bugs, and information on how they were eventually fixed. The limited information available is often within release notes where developers may vaguely mention that a bug fix was performed.

To summarize, the lack of bug locators designed for Android apps and the lack of a scalable approach of generating Android crash fixing templates jointly challenge the task of fixing crash-inducing bugs.

1.2.3 Android Framework Documentation Challenges

The Android apps heavily use the Android framework APIs. However, a lot of crashes arise from incorrect usage of these framework APIs [92,99,140], since these APIs may potentially throw exceptions, which, when not handled, crash the apps. In order to assist the developers in preventing them from writing apps that may crash because of buggy usage of framework APIs, we need to have knowledge on which framework APIs may throw exceptions and when. However, this goal is challenged by the below facts.

- **Android official API reference.** Since API exceptions are the main reason why Android apps crash so often [56], it is imperative to provide to developers the information on the exceptions that APIs may throw. However, although the Android official API reference [72] describes in detail the functionality of the APIs, it rarely points out which unchecked exceptions may be thrown by these APIs. Therefore, the app developers are challenged to write error-free code in the absence of this knowledge. Also, since the Android framework is enormous and too complex to be manually analyzed, generating this exception knowledge urgently calls upon automation. However, such automation tools are never found in the literature.
- **API Parameter.** In order to prevent themselves from writing error-prone code, and to eliminate such hazards of API exceptions, developers need to know which API parameter(s) are linked to the exceptions. However, as mentioned above, this information is not provided in the API reference. Also, the developers are challenged to obtain this information on their own since understanding the Android framework code is extremely effort- and time-consuming. Therefore, automation is called upon.

In summary, providing preventive measures in taming Android app crashes is challenged by inaccurate and insufficient documentation of Android framework APIs.

1.3 Contributions

We now summarize the contributions of this dissertation as below:

- *Systematic Literature Review on automated app testing approaches*: We aim at providing a clear overview of the state-of-the-art works around the topic of Android app testing, in an attempt to highlight the main trends, pinpoint the main methodologies applied and enumerate the challenges faced by the Android testing approaches as well as the directions where the community effort is still needed. To this end, we conduct a Systematic Literature Review (SLR) during which we eventually identified 103 relevant research papers published in leading conferences and journals until 2016. Our thorough examination of the relevant literature has led to several findings and highlighted the challenges that Android testing researchers should strive to address in the future. After that, we further propose a few concrete research directions where testing approaches are needed to solve recurrent issues in app updates, continuous increases of app sizes, as well as the Android ecosystem fragmentation.
  This work has led to a research paper published on the IEEE Transaction on Reliability in 2019 (TRel).
- *Locating Android app crash-inducing bugs*: We perform an empirical study on 500 framework-specific crashes from an open benchmark. This study reveals that 37 percent of the crash types are related to bugs that are outside the crash stack traces. Moreover, Android programs are a mixture of code and extra-code artifacts such as the Manifest file. The fact that any artifact can lead to failures in the app execution creates the need to position the localization target beyond the code realm. We propose ANCHOR, a two-phase suspicious bug location suggestion tool. ANCHOR specializes in finding crash-inducing bugs outside the stack trace. ANCHOR is lightweight and source code independent since it only requires the crash message and the apk file to locate the fault. Experimental results, collected via cross-validation and in-the-wild dataset evaluation, show that ANCHOR is effective in locating Android framework-specific crashing faults.
  This work has led to a research paper submitted for peer review to the Springer Journal of Automated Software Engineering in 2021 (ASE J).
- *Mining Android app crash fix templates*: We propose a scalable approach, CraftDroid, to mine crash fixes by leveraging a set of 28 thousand carefully reconstructed app lineages from app markets, without the need for the app source code or issue reports. We develop a replicative testing approach that locates fixes among app versions which output different runtime logs with the exact same test inputs. Overall, we have mined 104 relevant crash fixes, further abstracted 17 fine-grained fix templates that are demonstrated to be effective for patching crashed apks. Finally, we release ReCBench, a benchmark consisting of 200 crashed apks and the crash replication scripts, which the community can explore for evaluating generated crash-inducing bug patches.
  This work has led to a research paper published on the 28th *International Symposium on Software Testing and Analysis in 2019 (ISSTA'19).*
- *Documenting Framework APIs' Unchecked Exceptions*:
  We propose Afuera, an automated tool that profiles Android framework APIs and provides information on when they can potentially trigger unchecked exceptions. Afuera relies on a static-analysis approach and a dedicated algorithm to examine the entire Android framework. With Afuera, we confirmed that 26 739 unique unchecked exception instances may be triggered by invoking 5 467 (24%) Android framework APIs. Identifying unchecked exceptions is an important first step, however, it is extremely complex and time-consuming to understand when these unchecked exceptions are triggered. Therefore, Afuera further analyzes the Android

framework to inform about which parameter(s) of an API method can potentially be the cause of the triggering of an unchecked exception. To that end, Afuera relies on fully automated instrumentation and taint analysis techniques. Afuera is run to analyze 50 randomly sampled APIs to demonstrate its effectiveness. Evaluation results suggest that Afuera has perfect true positive rate. However, Afuera is affected by false negatives due to the limitation of state-of-the-art taint analysis techniques.
This work has led to a research paper submitted for peer review to the 30th *International Symposium on Software Testing and Analysis in 2021 (ISSTA'21).*

1.4 Roadmap

Figure 1.1 illustrates the roadmap of this dissertation. Chapter 2 gives a brief introduction on the necessary background information. Then, we present two paths of studies in this dissertation. In Chapters 3, 4, 5, we present the path for the curative toolchain. In Chapter 6, we present the path for the preventive toolchain. In Chapters 3, we present a Systematic Literature Review (SLR) on the automated testing approaches on Android apps. In this SLR, we study what characteristics affect testing approaches in exposing app crashes efficiently. In Chapter 4, we present a crash-inducing bug locator designed specifically for Android apps. In Chapters 5, we present how to mine fixing templates in the absence of issue- and change-tracking systems for fixing Android app crashes. In Chapters 6, we present how to assist developers in preventing crash-inducing bugs by documenting Android framework APIs for unchecked exceptions. Finally, in Chapter 7, we conclude this dissertation and discuss some potential future works that are in line with this dissertation.

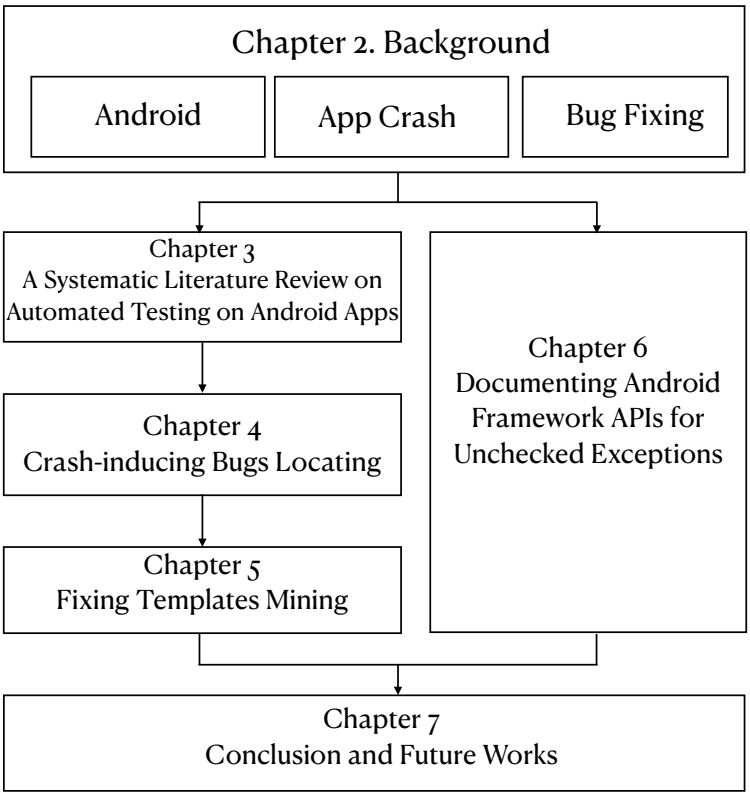


Figure 1.1: Roadmap of This Dissertation.

2 Background

In this chapter, we provide the necessary background needed to understand the targets, concerns and technical details of the 4 research studies we conducted in this dissertation. Specifically, we revisit the Android ecosystem, important concepts for app crash, the static analysis technique, and the datasets involved in this dissertation.

Contents

2.1 Android **8**
2.1.1 Architecture 8
2.1.2 API Level Evolution 9
2.1.3 Manifestation 10
2.2 App Crash **10**
2.2.1 Android Debug Bridge 10
2.2.2 Logcat 11
2.2.3 Stack Trace 11
2.3 Static Analysis **12**
2.3.1 Call Graph Construction 12
2.3.2 Taint Analyzer 13
2.4 Datasets **13**
2.4.1 F-Droid 13
2.4.2 AndroZoo 13
2.4.3 Lineage 13

2.1 Android

Android is an open source, Linux-based software stack designed for different types of devices, e.g., cell phones, TVs, car. Here in this section, we describe the architecture of Android platform [75], the evolution of the Android API version, and address the contents of Android app apk files.

2.1.1 Architecture

Figure 2.1 shows the architecture and main components of the Android platform. We now detail the major parts in the stack in the below list.

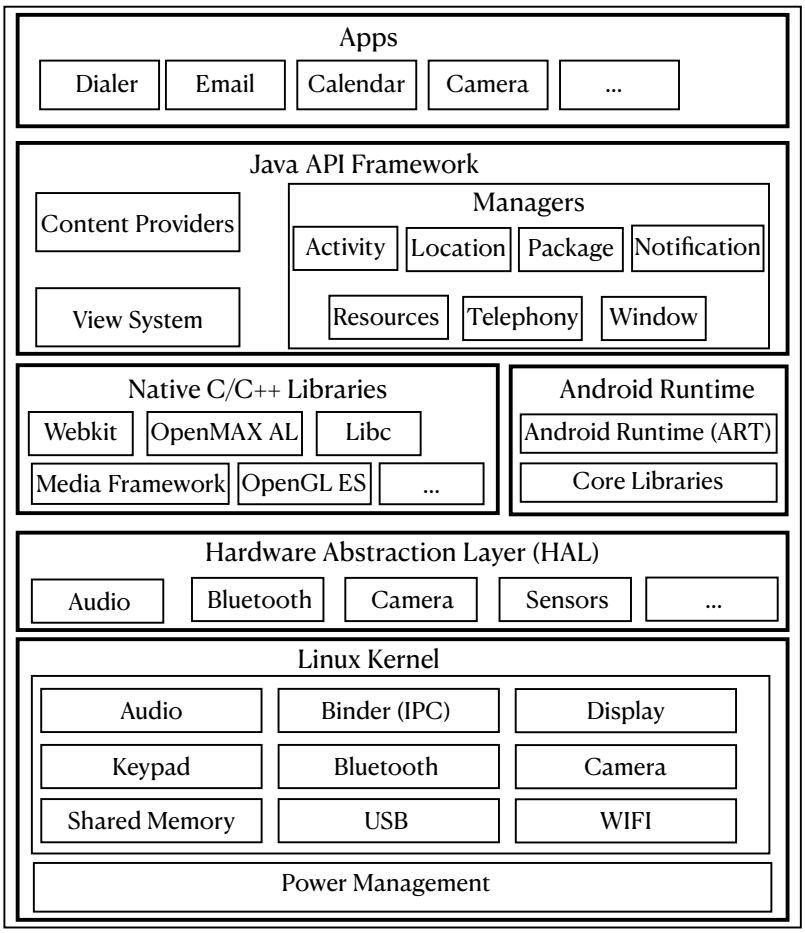


Figure 2.1: The Android Architecture.

- **The Linux kernel.** The entire Android platform is built on top of the Linux kernel. For example, the Android Runtime relies on the Linux kernel for basic functionalities like threading, low-level memory and power management. The advantages of using a Linux kernel are to allow Android to inherit the key security features and allow device manufacturers to also develop hardware drivers that can be migrated to a well-known kernel.
- **Hardware Abstraction Layer (HAL).** The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL consists of multiple library modules, each implementing an interface for a specific hardware component. With the advancement of new Android device hardware components, new modules are added to adapt to the changes. The apps running on the device

can access these hardware modules through the framework API, which in turn loads the library module for that component.

- **Android Runtime and Native C/C++ Libraries** For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART). Prior to Android version 5.0 (API level 21), Dalvik was the Android runtime. Apps that run on ART should also run on Dalvik. Also, Dalvik and ART share the same bytecode instruction set for the virtual machine. Many core Android system components and services, such as ART and HAL, are built from native code that requires native libraries written in C and C++.
- **Java API framework** The entire feature-set of the Android OS is available to the developers through APIs written in the Java language. Therefore, research works that were targeting the Java language can be important basis for carrying out analysis on the Android apps. These APIs are the building blocks for the developers to create Android apps by reusing the core, modular system components and services, as detailed in Figure 2.1. The developer apps have full access to the same framework APIs that the Android system apps use.
- **Apps** The apps layer contains both the system apps and the developer apps. The system apps are a set of core apps for email, SMS messaging, calendars, internet browsing, contacts and more. The users can also choose to install third-party apps. The third-party apps can replace the default system apps for general functions like calendar or keyboards if the user prefer. The third-party apps can also choose to reuse functionalities from the system apps or to add new features.

In this dissertation, our studies are primarily limited in analyzing the Java API framework and the apps. Although knowledge of other parts of the Android architecture are also needed. For example, knowledge of the ART bytecode helps to understand how static analysis tools (c.f Section 2.3) used in this dissertation function with compiled Android apps.

2.1.2 API Level Evolution

Android is a fast-evolving system [78]. Based on the Java framework API level, Android gives unique codenames and tags for different versions. Figure 2.2 depicts the Android version evolution. We select the first API version of each unique codename for demonstration. Note that versions with less than 1% market share are excluded from this figure. The percentage in Figure 2.2 is the cumulative distribution of the API level. For example, if an app runs on Lollipop 5.0 (API level 21), then it should be able to run on at least 94.1% Android devices. This is because Android SDK is forward compatible. Therefore, if an app runs on the current level of SDK, it is guaranteed to run on all higher levels of SDKs.

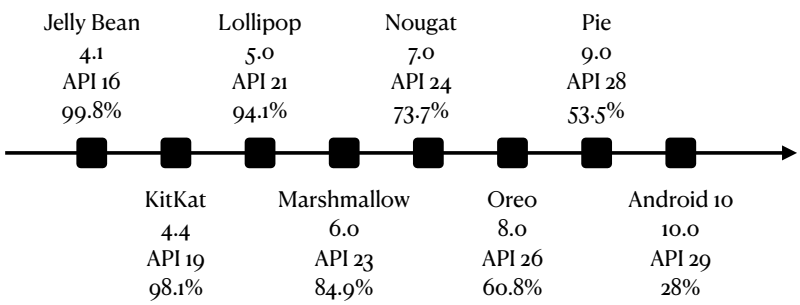


Figure 2.2: The Android Version Evolution (Data updated in March, 2021).

Understanding the Android API level evolution is important in understanding a specific type of Android app crashes: those caused by incompatibility between apps' targeted API level and hosting device's pre-installed API level [92, 99, 140].

2.1.3 Manifestation

The Android apps are distributed in the format of the Android Application Package (APK) files. It is a zip file containing code and other artifacts. The main contents and folders are listed in Table 2.1.

Table 2.1: Contents in an APK File.

Contained Files	Description
META-INF /	Meta-data relevant to the APK file contents
lib/	platform dependent compiled code
res/	resources not compiled into resources.arsc
assets/	application assets retrievable by *AssetManager*
AndroidManifest.xml	Gloable configuration file for the app
classes.dex	dex file compiled from Java code, understandable by Dalvik Virutal Machine and ART.
resources.arsc	precompiled resources, such as binary XML

Every apk must have an *AndroidManifest.xml* file at the root of the app project source set. It describes key information [74] about the app. The manifest must declare:

- **App package name** that usually match the code's namespace (although subpackages are also possible). Once the APK is compiled, the package attribute also represents the app's universally unique application ID. We use this attribute widely in this dissertation for grouping different apk versions of the same app, separating app-specific classes from library classes for static analysis, and more.
- **The components of the app** which include all instances of the 4 basic components [77] of Android apps: Activities, Services, Broadcast Receivers, and Content Providers. Each component must define basic properties such as the class names, device configurations it can handle and intent filters that describe how the component can be started.
- **Permissions** that the apps need to access protected parts of the system or other apps. It also declares any permissions that other apps must have if they want to access content from the current app. Notably, apps may also crash for security-related exceptions originated from lack of granted permissions [1–3, 82].
- **Features** of hardware and software that the app requires. This information affects which devices can install the app. If this information is not declared properly, and when apps are installed on devices that they do not support, the apps may also face crashing.

Understanding the files and folders in the app APK file is crucial for both automating the testing process as well as analyzing them with static tools.

2.2 App Crash

In this section, we describe two major command-line tools that are fundamentally important for analyzing the Android app crashes. We also describe the core information retrieved with these tools that come with all app crashes.

2.2.1 Android Debug Bridge

Android Debug Bridge (ADB) [73] is a versatile command-line tool that lets the developers and the researchers to communicate with the device (real device or emulators). The ADB command facilities a variety of device actions, e.g., installing and uninstalling apps, sending test inputs, and retrieving runtime logs. It provides access to a unix shell that can be used to run a variety of commands on the devices. It is a program-server component that includes three components: a client, a daemon (adbd) and a server. The client runs on the development environment and send commands. The daemon runs the commands on the device as a background process. The server runs as a background

process on the development environment and manages the communication between the client and the daemon.

ADB is the fundamental part of almost every automated testing tools (c.f Chapter 3). These tools send testing events that mimic user or sensor inputs via ADB. Some tools also retrieve runtime information like layout XML files via ADB commands, and adjust the testing input generation strategy based this information. Also, ADB can be used to access Logcat runtime information which logs app crashes and other useful information, as detailed in below subsections.

2.2.2 Logcat

Logcat [81] is a command-line tool of Android devices that dumps a log of system messages. Specifically, the stack traces (c.f Section 2.2.3) are in this log when the apps crash from an exception. The developers can also write messages to this log by using the *Log* class in their apps. When accessing the logcat tool from the development environment, the ADB tool is needed to connect the hosted device. Logcat is useful from different perspectives throughout this dissertation. First, we filter out Logcat outputs to know if the target app has crashed. Second, we retrieve relevant information about the crash, especially the stack traces and perform the analysis and fixing procedures based on this information. Third, researchers use the *Log* class widely to instrument the apps under study and collect customized runtime information. Such runtime information is useful in adjusting the testing strategy and for bug localization.

Listing 2.1 is a Logcat dump of Android app *Transistor*[1]. This app helps users to subscribe to radio channels through internet. The listing shows the log message related to *Transistor*'s crashing. Lines 1-3 contain the basic information of *Transistor*'s log time, log description, the process name and process ID. Line 4 describes the exception type that caused the crash, as well as the exception message. In Lines 5-17, the stack trace of the crash is included, which will be described in detail in Section 2.2.3.

```
1  01-21 00:37:46.789 31054-31054/org.y20k.transistor E/AndroidRuntime:
2  FATAL EXCEPTION: main
3  Process: org.y20k.transistor, PID: 31054
4  java.lang.IllegalStateException: Fragment MainActivityFragment{e7db358} not attached to Activity
5  at android.support.v4.app.Fragment.startActivityForResult(Fragment.java:925)
6  at org.y20k.transistor.MainActivityFragment.selectFromImagePicker(MainActivityFragment.java:482)
7  at org.y20k.transistor.MainActivityFragment.access$500(MainActivityFragment.java:58)
8  at org.y20k.transistor.MainActivityFragment$6.onReceive(MainActivityFragment.java:415)
9  at android.support.v4.content.LocalBroadcastManager.executePendingBroadcasts(LocalBroadcastManager.
       java:297)
10 at android.support.v4.content.LocalBroadcastManager.access$000(LocalBroadcastManager.java:46)
11 at android.support.v4.content.LocalBroadcastManager$1.handleMessage(LocalBroadcastManager.java:116
       )
12 at android.os.Handler.dispatchMessage(Handler.java:102)
13 at android.os.Looper.loop(Looper.java:148)
14 at android.app.ActivityThread.main(ActivityThread.java:5417)
15 at java.lang.reflect.Method.invoke(Native Method)
16 at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:726)
17 at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:616)
```

Listing 2.1: Logcat Dump of app Transistor.

2.2.3 Stack Trace

Like all Java[2] based software, when Android apps crash, they can dump execution traces which include the exception being thrown, a crash message, and most importantly, a stack trace of a

[1] https://github.com/y20k/transistor/issues/21

[2] Kotlin has also been widely used in recent years as an alternative for Android app development, it is designed to fully interoperate with Java.

callee-caller chain. This chain starts from the *Signaler*, i.e., the method that initially constructed and threw the exception object.

Listing 2.2.3 contains an example of stack trace for the crash of the app *Transistor*. On Line 4, the exception *IllegalStateException* is thrown. On the same line, the log system reports message "Fragment MainActivityFragment{e7db358} not attached to Activity". In Line 5, the *Signaler* of the stack trace is listed: it is this framework method that instantiates the exception type, composes the log message and throws it to its caller to handle. Line 5 is also the *API*, which is the only framework method in this stack trace that is visible to the developer. Since the crash happens directly due to invocation to it, we call it the *Crash API*. In practice, it is not uncommon that the *Signaler* and the *Crash API* are different methods which may have direct or indirect caller-callee relation. Line 6 is the developer method that invoked this API. Line 8 is the developer implementation of the callback, inherited from the superclass of the Android framework. The crash stack trace is often the first thing that developers want to examine when a crash is reported [115]. Even when it is not given, developers would reproduce [56, 270] and retrieve them.

In this dissertation, we use the ADB to perform automated testing. We use the Logcat to retrieve the stack trace and other app execution related information for analysis.

2.3 Static Analysis

Static analysis technique aims at examining the app without executing it. It is the key technique that we leverage throughout this dissertation. It helps with three main aspects for taming Android app crashes. First, static analysis is widely used to extract knowledge from the apps about the components and to instrument the apps by inserting logging points. These two operations are fundamental for generating the automated testing strategy for Android apps, as detailed in Chapter 3. Second, static analysis techniques are crucial for the program repairing for app crashes. For example, in Chapter 4 we need to use static analysis for locating crash-inducing bugs. In Chapter 5, we use static analysis techniques to retrieve true fixes from app lineages. Third, static analysis techniques can be used to analyze the Android framework SDK, as studied in Chapter 6.

Since Android apps mainly implement their logic in Java, we can make use of the static analysis tools designed for Java over the past decades. Throughout this dissertation, we mainly leverage the Soot [22] static analysis tool chain to perform the static analysis. Soot was initially proposed to analyze and optimize the Java programs. It transfers Java bytecode into the Intermediate Representation (IR) forms and perform the operation on the IR code. Then, with the proposal of Dexpler [30], Soot can also transfer Dalvik bytecode into the same IR sets. Therefore, the existing tools designed for Java can be in principle reused for analyzing Android apps. In this section, we mainly list two static analyzers that are fundamentally important for this dissertation.

2.3.1 Call Graph Construction

The call graph construction for Java programs and for Android apps are very similar but with explicit differences. Java programs have a single entry point, i.e., the *main* method. However, Android apps contain multiple components, each with one or more entry points. Passing control to which entry point during execution is largely affected by the framework logic itself. Therefore, the static analysis for Android apps will not be precise if it does not take the framework logic in consideration. Consequently, to obtain a precise and largely complete call graph of the app, the Soot framework creates a dummy *main()* method and invoke the entry points from it. To create the dummy *main()* method, Soot analyzes various types of files, including source code, manifest, layout xml files for the components and the *resourcs.arsc* file (c.f Table 2.1). Also, since the Android developers can register callbacks in the code as well as declaring them in the respective layout XML file, it requires great

care to precisely model the callbacks for the app components. When the dummy *main()* method is properly constructed, Soot uses its existing call graph construction framework SPARK [129] or other implemented call graph construction algorithms designed for Java to finish the task.

2.3.2 Taint Analyzer

FlowDroid [23] is an essential part of Soot that can perform highly precise data-flow tracking by taint analysis. FlowDroid was initially developed to find privacy issues in Android apps. It can verify if there is privacy leakage from one method invocation to another. In Chapter 6, we demonstrate how to analyze the Android framework Java SDK with a small add-on to FlowDroid.

2.4 Datasets

In this section, we present 3 datasets related to Android apps that are fundamentally important to the research community of Android as well as to this dissertation.

2.4.1 F-Droid

F-Droid [53] is a repository and an installable catalog of Free and Open Source Software (FOSS) for the Android platform. The apps hosted on F-Droid are either product of online open source Android repositories or their modified versions published on other markets. F-Droid keeps track of the application's 3 most recent releases, while other commercial markets often only provide the newest one. Since F-Droid also provides links to the online repositories from which the apps are compiled, researchers can also download the source code of these apps to evaluate their research works [44, 119, 179, 216, 264] related to testing and static analysis for Android apps. F-Droid is also adopted by researchers to form new benchmarks. For example, Fan et al. [55] collected a dataset of closed issue reports related to crashes of apps hosted on F-Droid by analyzing the issue-tracking systems of such online repositories. In Chapters 4 and 5, we use such dataset to evaluate our own studies.

2.4.2 AndroZoo

While F-Droid forms a collection of open source Android apps, sophisticated studies also need to evaluate a much larger set of Android apps: the commercial and closed source apps. AndroZoo [11, 143, 144] is a growing collection of Android Applications collected from several sources, including the official Google Play app market. It currently contains more than 14 million different APKs, each of which has been (or will soon be) analyzed by around 60 different AntiVirus products to know which applications are detected as Malware. Researchers can use this dataset freely to train or evaluate their tools [66, 69, 238, 239]. The Chapters 4 and 5 rely heavily on this dataset to select Android apps for analysis.

2.4.3 Lineage

The concept of app lineage (i.e., a series of apk releases of a given app) was first introduced by Gao et al. [65]. In this dissertation, we use the same approach to construct app lineages from AndroZoo [11] apps. Overall, the app lineages are constructed via the following process: (1) identify unique apps, where APKs sharing the same *package name* are considered to be the same app, and (2) link and order the different app versions of the same app as shown in Figure 2.3. As a result, an app lineage

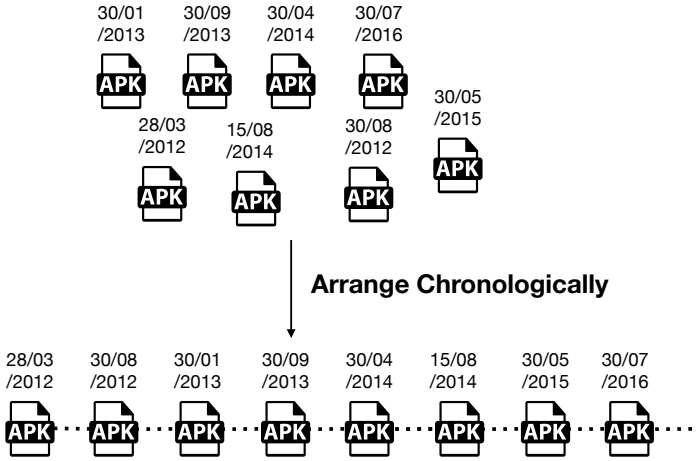


Figure 2.3: The Formation of App Lineage.

contains a set of Android apps that share the same package name while are totally ordered based on their release time. Note that a lineage can be sparse given that AndroZoo is not exhaustive in the collection of app versions. In Chapter 5, we show, by operations on the app lineages, how to obtain a dataset of reproducible app crashes.

3 Automated Testing of Android Apps: A Systematic Literature Review

In this chapter, we aim at providing a clear overview of the state-of-the-art works around the topic of Android app testing, in an attempt to highlight the main trends, pinpoint the main methodologies applied and enumerate the challenges faced by the Android testing approaches as well as the directions where the community effort is still needed. To this end, we conduct a Systematic Literature Review (SLR) during which we eventually identified 103 relevant research papers published in leading conferences and journals until 2016. This study further helps us in selecting automated testing tools that can expose Android app crashes efficiently. To be more specific, the selected tools need to be light-weight (i.e., does not require instrumentation), have good code coverage, be source code independent, and function both on emulators and real devices.

This chapter is based on the work published in the following research paper:

- Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 2018

Contents

3.1 Overview **16**
3.2 Methodology of This SLR **18**
3.2.1 Initial Research Questions 18
3.2.2 Search Strategy 19
3.2.3 Exclusion Criteria 20
3.2.4 Review Protocol 21
3.3 Primary Publications Selection **21**
3.4 Taxonomy of Android Testing Research **23**
3.5 Literature Review **24**
3.5.1 What concerns do the approaches focus on? 24
3.5.2 Which Test Levels are Addressed? 27
3.5.3 How are the Test Approaches Built? 28
3.5.4 To What Extent are the Approaches Validated? 32
3.6 Discussion **38**
3.6.1 Trend Analysis 38
3.6.2 Evaluation of Authors 39
3.6.3 Research Output Usability 40
3.6.4 Open Issues and Future Challenges 41
3.6.5 New Research Directions 42
3.7 Threats to Validity **43**
3.8 Related Work **43**
3.9 Summary **44**

3.1 Overview

Android smart devices have become pervasive after gaining tremendous popularity in recent years. As of July 2017, Google Play, the official app store, is distributing over 3 million Android applications (i.e., apps), covering over 30 categories ranging from entertainment and personalisation apps to education and financial apps. Such popularity among developer communities can be attributed to the accessible development environment based on familiar Java programming language as well as the availability of libraries implementing diverse functionalities [136]. The app distribution ecosystem around the official store and other alternative stores such as Anzhi and AppChina is further attractive for users to find apps and organisations to market their apps [143].

Unfortunately, the distribution ecosystem of Android is porous to poorly-tested apps [119, 130, 234]. Yet, as reported by Kochhar [119], error-prone apps can significantly impact user experience and lead to a downgrade of their ratings, eventually harming the reputation of app developers and their organizations [234]. It is thus becoming more and more important to ensure that Android apps are sufficiently tested before they are released on the market. However, instead of manual testing, which is often laborious, time-consuming and error-prone, the ever-growing complexity and the enormous number of Android apps call for scalable, robust and trustworthy automated testing solutions.

Android app testing aims at testing the functionality, usability and compatibility of apps running on Android devices [137, 141]. Fig. 3.1 illustrates a typical working process. At Step (1), target app is installed on an Android device. Then in Step (2), the app is analysed to generate test cases. We remind the readers that this step (in dashed line) is optional and some testing techniques such as automated random testing do not need to obtain pre-knowledge for generating test cases. Subsequently, in Step (3), these test cases are sent to the Android device to exercise the app. In Step (4), execution behaviour is observed and collected from all sorts of perspectives. Finally, in Step (5), the app is uninstalled and relevant data is wiped. We would like to remind the readers that installation of the target app is sometimes not a necessity, e.g., frameworks like Robolectric allow tests directly run in JVM. In fact, Fig. 3.1 can be borrowed to describe the workflow of testing almost any software besides Android apps. Android app testing, on the contrary, falls in a unique context and often fails to use general testing techniques [43, 50, 98, 169, 181, 263]. There are several differences with traditional (e.g., Java) application testing that motivate research on Android app testing. We enumerate and consider for our review a few common challenges:

First, although apps are developed in Java, traditional Java-based testing tools are not immediately usable on Android apps since most control-flow interactions in Android are governed by specific event-based mechanisms such as the Inter-Component Communication (ICC [132]). To address this first challenge, several new testing tools have been specifically designed for taking Android specificities into account. For example, RERAN [70] was proposed for testing Android apps through a timing- and touch-sensitive record-and-replay mechanism, in an attempt to capture, represent and replay complicated non-discrete gestures such as *circular bird swipe with increasing slingshot tension in Angry Birds*.

Second, Android fragmentation, in terms of the diversity of available OS versions and target devices (e.g., screen size varieties), is becoming acuter as now testing strategies have to take into account different execution contexts [140, 241].

Third, the Android ecosystem attracts a massive number of apps requiring scalable approaches to testing. Furthermore, these apps do not generally come with open source code, which may constrain the testing scenarios.

Finally, it is challenging to generate a perfect coverage of test cases, in order to find faults in Android apps. Traditional test case generation approaches based on *symbolic execution* and tools such as *Symbolic Pathfinder (SPF)* are challenged by the fact that Android apps are available in Dalvik bytecode that differs from Java bytecode. In other words, traditional Java-based symbolic execution

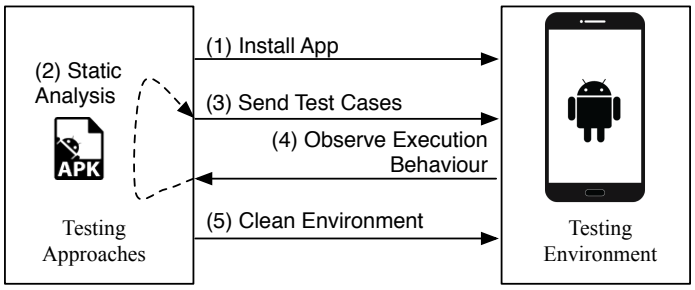


Figure 3.1: Process of testing Android apps.

approaches cannot be directly applied to tackle Android apps. Furthermore, the event-driven feature, as well as framework libraries, pose further obstacles for systematic generation of test cases [182].

Given the variety of challenges in testing Android apps, it is important for this field, which has already produced a significant amount of approaches, to reflect on what has already been solved, and on what remains to tackle. To the best of our knowledge, there is no related literature review or survey summarizing the topic of Android testing. Thus, we attempt to meet this need through a comprehensive study. Concretely, we undertake a systematic literature review (SLR), carefully following the guidelines proposed by Kitchenham et al. [117] and the lessons learned from applying SLR within the software engineering domain by Brereton et al. [34]. To achieve our goal, we have searched and identified a set of relevant publications from four well-known repositories including the ACM Digital Library and from major testing-related venues such as ISSTA, ICSE. Then, we have performed a detailed overview on the current state of research in testing Android apps, focusing on the types and phases of the testing approaches applied as well as on a trend analysis in research directions. Eventually, we summarize the limitations of the state-of-the-art apps and highlight potential new research directions.

The main contributions of this chapter are:

- We build a comprehensive repository tracking the research community effort to address the challenges in testing Android apps. In order to enable an easy navigation of the state-of-the-art, thus enabling and encouraging researchers to push the current frontiers in Android app testing, we make all collected and built information publicly available at

  `http://lilicoding.github.io/TA2Repo/`

- We analyse in detail the key aspects in testing Android apps and provide a taxonomy for clearly summarising and categorising all related research works.
- Finally, we investigate the current state of the art, enumerate the salient limitations and pinpoint a few directions for furthering the research in Android testing.

The rest of this chapter is organized as follows: Section 3.2 depicts the methodology of this systematic literature review, including a general overview and detailed reviewing processes of our approach. In Section 3.3, we present the results of our selected primary publications, along with a preliminary trend and statistic analysis on those collected publications. Later, we introduce our data extraction strategy and their corresponding findings in the following two sections: Section 3.4 and 3.5. After that, we discuss the trends we observed and challenges the community should attempt to address in Section 3.6 and enumerate the threats to validity of this SLR in Section 3.7. A comparison of this work with literature studies is given in Section 3.8 and finally we conclude this SLR in Section 3.9.

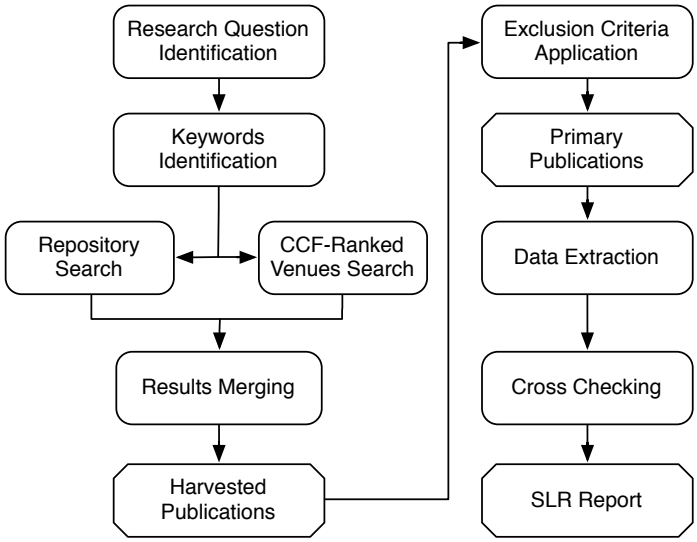


Figure 3.2: Process of the SLR.

3.2 Methodology of This SLR

We now introduce the methodology applied in this SLR. We remind the readers that an SLR follows a well-defined strategy to systematically identify, examine, synthesize, evaluate and compare all available literature works in a specific topic, resulting in a reliable and replicable report [117, 139, 190]. Fig. 3.2 illustrates the process of our SLR. At the beginning, we define relevant research questions (cf. Section 3.2.1) to frame our investigations. The following steps are unfolded to search and consolidate the relevant literature, before extracting data for answering the research questions, and finalizing the report.

Concretely, to harvest all relevant publications, we identify a set of search keywords and apply them in two separate processes: 1) online repository search and 2) major[1] venues search. All results are eventually merged for further reviewing (cf. Section 3.2.2). Next, we apply some exclusion criteria on the merged list of publications, to exclude irrelevant papers (e.g., papers not written in English) or less relevant papers (e.g., short papers), in order to focus on a small, but highly relevant, set of primary publications (cf. Section 3.2.3). Finally, we have developed various metrics and reviewed the selected primary publications against these metrics through full paper examination. After the examination, we cross-check the extracted results to ensure their correctness and eventually we report on the findings to the research community (cf. Section 3.2.4).

3.2.1 Initial Research Questions

Given the common challenges enumerated in the Introduction section, which have motivated several research lines in Android apps, we investigate several research questions to highlight how and which challenges have been focused on in the literature. In particular, with regards to the fact that Android has programming specificities (e.g., event-based mechanisms, GUI), we categorize test concerns targeted by the research community. With regards to the challenge of ensuring scalability, we study the tests levels which are addressed in research works. With regards to the challenge of generating test cases, we investigate in details the fundamental testing techniques leveraged. Finally, with regards to the fragmentation of the Android ecosystem, we explore the extent of validation schemes for research approaches. Overall, we note that testing Android apps is a broad activity that can target a variety of functional and non-functional requirements and verification issues, leverage different techniques

[1] We rely on the China Computer Federation (CCF) ranking of computer science venues.

and focus on different granularity levels and phases. Our investigation thus starts with the following related research questions:

- **RQ1: What are the test concerns?** With this research question, we survey the various objectives sought by Android app testing researchers. In general, we investigate the testing objectives at a high level to determine what requirements (e.g., security, performance, defects, energy) the literature addresses. We look more in-depth into the specificities of Android programming, to enumerate the priorities that are tackled by the community, including which concerns (e.g., GUI and ICC mechanism) are factored in the design of testing strategies.
- **RQ2: Which test levels are addressed?** With the second research question, we investigate the levels (i.e., when the tests are relevant in the app development process) that research works target. The community could indeed benefit from knowing to what extent regression testing is (or is not) developed for apps which are now commonly known to evolve rapidly.
- **RQ3: How are the testing approaches built?** In the third research question, we process detailed information on the design and implementation of test approaches. In particular, we investigate the fundamental techniques (e.g., concolic testing or mutation testing) leveraged, as well as the amount of input information (i.e., to what extent the tester should know about the app prior to testing) that approaches require to perform.
- **RQ4: To what extent are the testing approaches validated?** Finally, the fourth research question investigates the metrics, datasets and procedures in the literature for measuring the effectiveness of state-of-the-art approaches. Answers to this question may shed light on the gaps in the research agenda of Android testing.

3.2.2 Search Strategy

We now detail the search strategy that we applied to harvest literature works related to Android app testing.

Identification of search keywords. Our review focuses on two key aspects: Testing and Android. Since a diversity of terms may be used by authors to refer, broadly or precisely, to any of these aspects, we rely on the extended set of keywords identified in Table 3.1. Our final search string is then constructed as a conjunction of these two categories of keywords ($search_string = cat1$ & $cat2$), where each category is represented as a disjunction of its keywords ($cat = kw1 \mid kw2 \mid kw3$).

Table 3.1: Search Keywords

Category	Keywords
Android	android, mobile, portable device, smartphone, smart phone, smart device
Test	test, testing, measure, measurement, measuring, check, checking, detect, detecting, detection

Online repository search. We use the search string on online literature databases to find and collect relevant papers. We have considered four widely used repository for our work: ACM Digital Library[2], IEEE Xplore Digital Library[3], SpringerLink[4], and ScienceDirect[5]. The "advanced" search functionality of the four selected online repositories are known to be inaccurate, which usually result in a huge set of irrelevant publications, noising the final paper set [139]. Indeed, those irrelevant publications do not really match our keywords criteria. For example, they may not contain any of the keywords shown in the *Test* category. Thus, we develop scripts (combined with Python and Shell) to perform off-line matching verification on the papers yielded by those search engines, where the

[2]http://dl.acm.org/
[3]http://ieeexplore.ieee.org/Xlpore/home.jsp
[4]http://link.springer.com
[5]http://www.sciencedirect.com

scripts follow exactly the same criteria that we have used for online repository search. For example, regarding the keywords enumerated in the *Test* category, if none of them is presented in a publication, the scripts will mark that publication as irrelevant and subsequently exclude it from the candidate list.

Major venues search. Since we only consider a few repositories for search, the coverage can be limited given that a few conferences such as NDSS[6] and SEKE[7] do not host their proceedings in the aforementioned repositories. Thus, to mitigate the threat to validity of not including all relevant papers, we further explicitly search in proceedings of all major venues in computer science. We have chosen the comprehensive CCF-ranking of venues[8] and leveraged the DBLP[9] repository to collect the Document Object Identifiers (DOI) of the publications in order to crawl abstracts and all publication metadata. Since this search process considers major journal and conference venues, the resulting set of literature papers should be a representative collection of the state-of-the-art.

3.2.3 Exclusion Criteria

After execution of our search based on the provided keywords, a preliminary manual scanning showed that the results are rather coarse-grained since it included a number of irrelevant or less relevant publications which, nonetheless, matched[10] the keywords. It is thus necessary to perform a fine-grained inclusion/exclusion in order to focus on a consistent and reliable set of primary publications and reduce the eventual effort in further in-depth examination. For this SLR, we have applied the following exclusion criteria:

1. Papers that are not written in English are filtered out since English is the common language spoken in the worldwide scientific peer-reviewing community.
2. Short papers are excluded, mainly because such papers are often work-in-progress or idea papers: on the one hand, short papers are generally not mature, and, on the other hand, many of them will eventually appear later in a full paper format. In the latter case, mature works are likely to already be included in our final set. In this work, we take a given publication as a short paper when it has fewer than 4 pages (included) in IEEE/ACM-like double-column format [11] or fewer than 8 pages (included) in LNCS-like single column format as short papers are likely to be 4 pages in double column format and 8 pages in single column format.
3. Papers that are irrelevant to testing Android apps are excluded. Our search keywords indeed included broad terms such as *mobile* and *smartphone* as we aimed at finding all papers related to Android even when the term "Android" was not specifically included in the title and abstract. By doing so, we have excluded papers that only deal with mobile apps for other platforms such as iOS and Windows.
4. Duplicated papers are removed. It is quite common for authors to publish an extended version of their conference paper to a journal venue. However, these papers share most of the ideas and approach steps. To consider both of them would result in a biased weighting of the metrics in the review. To mitigate this, we identify duplicate papers by first comparing paper titles, abstracts and authors and then further manually check when a given pair of records share a major part of their contents. We filter out the least recent publication when duplication is confirmed.
5. Papers that conduct comparative evaluations, including surveys on different approaches of testing Android apps, are excluded. Such papers indeed do not introduce new technical contributions for testing Android apps.

[6] The Network and Distributed System Security Symposium
[7] International Conference on Software Engineering & Knowledge Engineering
[8] http://www.ccf.org.cn/sites/ccf/paiming.jsp, we only take into account *software engineering* and *security* categories, as from what have observed, the majority of papers related to *testing Android apps.*
[9] http://dblp.uni-trier.de
[10] The keywords were found for example to be mentioned in the related sections of the identified papers.
[11] Note that we have actually kept a short paper entitled "GuiDiff: a regression testing tool for graphical user interface" because it is very relevant to our study and it does not have an extended version released in the following years.

6. Papers in which the testing approach targets the operating system, networks, or hardware, rather than mobile apps are excluded.
7. Papers that assess[12] existing testing methods are also filtered out. The publications that they discuss are supposed to be already included in our search results.
8. Papers demonstrating how to set up environments and platforms to retrieve runtime data from Android apps are excluded. These papers are also important for Android Apps testing, but they are not focusing on new testing methodology.
9. Finally, some of our keywords (e.g., "detection" of issues, "testing" of apps) have led to the retrieval of irrelevant literature works that must be excluded. We have mainly identified two types of such papers: the first includes papers that perform *detection* of malicious apps using machine learning (and not testing); the second includes papers that describe the building of complex platforms, adopting existing mature *testing* methodologies.

We refer to all collected papers that remain after the application of exclusion criteria as **primary publications**. These publications are the basis for extracting review data.

3.2.4 Review Protocol

Concretely, the review is conducted in two phases: 1) First, we perform an abstract review and quick full paper scan to filter out irrelevant papers based on the exclusion criteria defined above. At the end of this phase, the set of primary publications is known. 2) Subsequently, we perform a full review of each primary publication and extract relevant information that is necessary for answering all of our research questions.

In practice, we have split our primary publications to all the co-authors to conduct the data extraction step. We have further cross-checked all the extracted results: when some results are in disagreement, informal discussions are conducted until a consensus is reached.

3.3 Primary Publications Selection

Table 3.2: Summary of the selection of primary publications.

Step	Count
Repository and Major Venues Search	9259
After reviewing titles/abstracts (scripts)	472
After reviewing titles/abstracts	255
After skimming/scanning full paper	171
After final discussion	103

Table 3.2 summarizes statistics of collected papers during the search phase. Overall, our repository search and major venue search have yielded in total 9,259 papers.

Following the exclusion criteria in Section 3.2, the papers satisfying the matching requirements immediately drop from 9259 to 472. We then manually go through the title and abstract of each paper to further dismiss those that match the exclusion criteria. After this step, the set of papers is reduced to 255 publications. Subsequently, we go through the full content of papers in the set, leading to the exclusion of 84 more papers. Finally, after discussion among the authors for the rest of the set, we reach a consensus on considering 103 publications as relevant primary publications. Table 3.10 enumerates the details of those 103 publications.

[12]For example, [271] and [259] propose tools and algorithms for measuring the code coverage of testing methods.

Figure 3.3: Word Cloud based on the Venue Names of Selected Primary Publications.

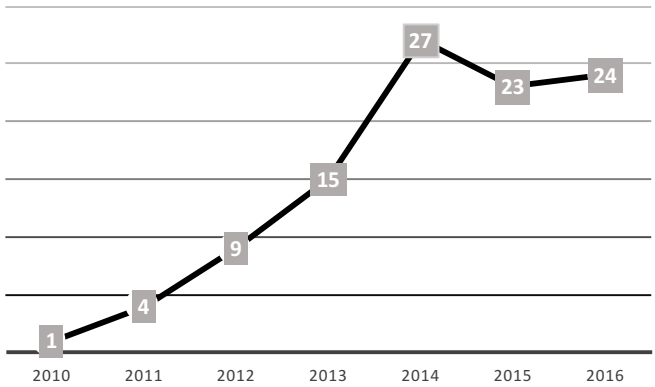


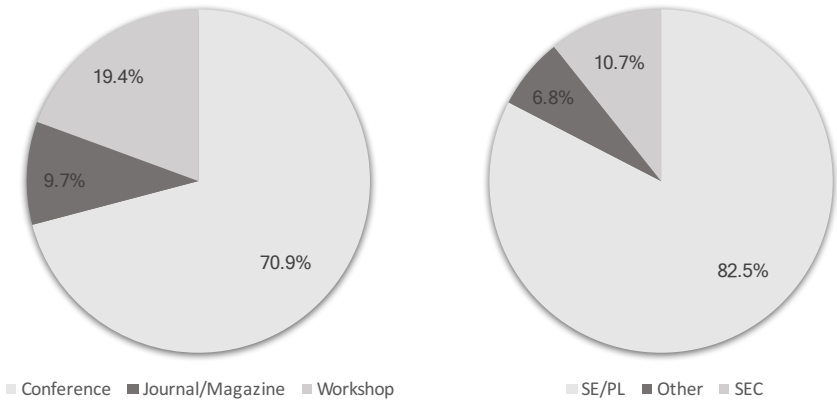
Figure 3.4: The number of publications in each year.

It is noteworthy that around 4% of the final primary publications are exclusively found by major venues search, meaning that they cannot be found based on well-known online repositories such as IEEE and ACM. This result, along with our previous experiences [139], suggests that repository search is necessary but not sufficient for harvesting review publications. Other steps (e.g., top venues search based on Google Scholar impact factor [139] or CCF ranking) should be taken in complement to ensure reliable coverage of state-of-the-art papers.

Fig. 3.3 presents a word cloud based on the venue names of selected primary publications. The more papers selected from a venue, the bigger its name showing in the word cloud. Not surprisingly, the recurrently targeted venues are mainly testing-related conferences such as ISSTA, ICST, ISSRE, etc.

Fig. 3.4 illustrates the trend of the number of publications in each year we have considered. From this figure, we can observe that the number of papers tackling the problem of testing Android apps has increased gradually to reach a peak in 2014. Afterwards, the pace of developing new testing techniques has stabilized.

We further look into the selected primary publications through their published venue types and domains. Fig. 3.5a and Fig. 3.5b illustrate the statistic results, respectively. Over 90% of examined papers are published in conferences and workshops (which are usually co-located with top conferences) while only 10% papers are published in journals. These findings are in line with the current situation where intense competition in Android research forces researchers to make available their works as fast as possible. We further find that over 80% of examined papers are published in software engineering



a Venue Types. **b** Venue Domains.

Figure 3.5: Distribution of examined publications through published venue types and domains.

and programming language venues, showing that testing Android apps is mainly a concern in the software engineering community. Nevertheless, as shown by several papers published in proceedings of security venues, testing is also a valuable approach to address security issues in Android apps.

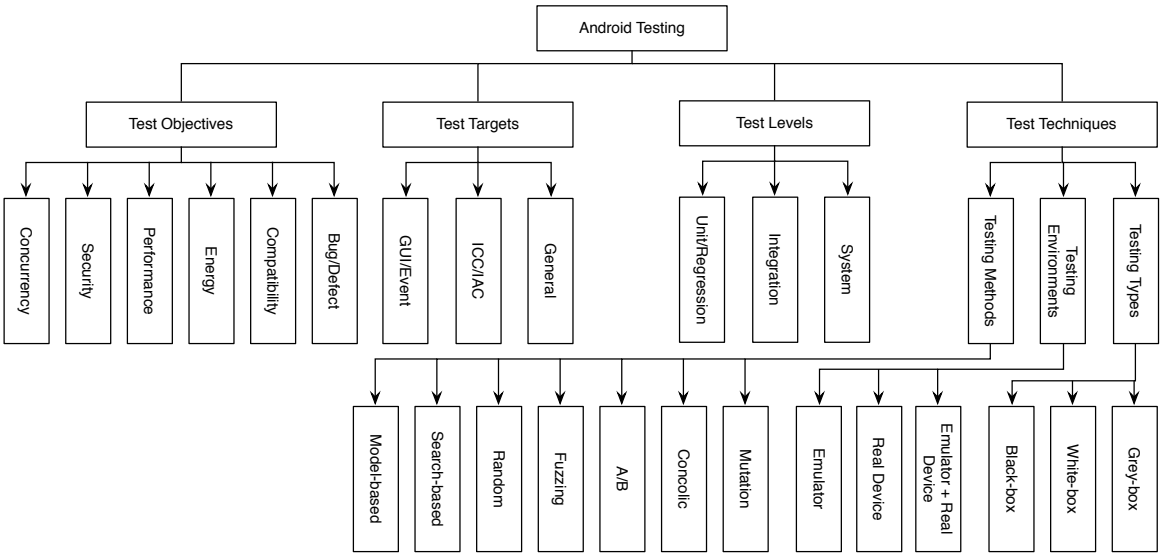


Figure 3.6: Taxonomy of Android App Testing.

3.4 Taxonomy of Android Testing Research

To extract relevant information from the literature, our SLR must focus on specific characteristics eventually described in each publication. To facilitate this process in a field that explores a large variety of approaches, we propose to build a taxonomy of Android testing. Such a taxonomy eventually helps to gain insights into the state-of-the-art by answering the research questions proposed in Section 3.2.1.

By searching for answers to the aforementioned research questions in each publication, we are able to make a systematic assessment of the literature with a schema for classifying and comparing different approaches. Fig. 3.6 presents a high-level view of the taxonomy diagram spreading in four dimensions

(i.e., **Test Objectives**, **Test Targets**, **Test Levels** and **Test Techniques**) associated with the first three research questions[13].

Test Objectives. This dimension summarizes the targeted objectives of our examined testing-related publications. We have enumerated overall 6 recurring testing objectives such as Bug/Defect detection.

Test Targets. This dimension summarizes the representative targets where testing approaches focus on. In particular, for testing Android apps, the GUI/Event and ICC/IAC are recurrently targeted. For simplicity, we regroup all the other targets such as normal code analysis into *General*.

Test Levels. This dimension checks the different levels (also known as phases) at which the test activities are performed. Indeed, there is a common knowledge that software testing is very important and has to be applied to many levels such as *unit testing*, *integration testing*, etc. Android apps, as a specific type of software, also need to go through a thorough testing progress before being released to public markets. In this dimension, we sum up the targeted testing phases/levels of examined approaches, to understand what has been focused so far by the state-of-the-art.

Test Techniques. Finally, the fourth dimension focuses on the fundamental methodologies (e.g., Fuzzy or Mutation) that are followed to perform the tests, as well as the testing environments (e.g., on emulated hardware) and testing types (e.g., black-box testing).

3.5 Literature Review

We now report on the findings of this SLR in light of the research questions that we have raised in Section 3.2.1.

3.5.1 What concerns do the approaches focus on?

Our review investigates both the objectives that testing approaches seek to achieve and the app elements that are targeted by the test cases. *Test objectives* focus on problems that can be located anywhere in the code, while *test targets* focus on specific app elements that normally involve only certain types of code (e.g., functionality).

3.5.1.1 Test objectives

Android testing research has tackled various objectives, including the assessment of apps against non-functional properties such as app efficiency in terms of *energy* consumption, and functional requirements such as the presence of bugs. We discuss in this section some recurrent test objectives from the literature.

Concurrency. Android apps expose a concurrency model that combines multi-threading and asynchronous event-based dispatch, which may lead to subtle concurrency errors because of unforeseen thread interleaving coupled with non-deterministic reordering of asynchronous tasks. These error-prone features are however useful and increasingly becoming common in the development of efficient and feature-rich apps. To mitigate concurrency issues, several works have been proposed, notably for detecting races such as data races, event-based races, etc. in Android apps. As an example, Maiya et al. [166] have built DroidRacer, which identifies data races (i.e., the *read* and *write* operations happen in parallel) by computing the happens-before relation on execution traces that are generated

[13] **Test Objectives** and **Test Targets** for RQ1 (test concerns), **Test Levels** for RQ2 (test levels) and **Test Techniques** for RQ3 (test approaches). RQ4 explores the validity of testing approaches that is not summarised in the taxonomy.

Table 3.3: Test objectives in the literature.

Tool	Concurrency	Security	Performance	Energy	Compatibility	Bug/Defect	Tool	Concurrency	Security	Performance	Energy	Compatibility	Bug/Defect
Dagger [252]		✓					Malisa et al. [168]		✓				
CRASHSCOPE [183]						✓	MAMBA [113]		✓				
Pretect [111]			✓				SSDA [97]						✓
TrimDroid [181]						✓	ERVA [98]	✓					
SAPIENZ [169]						✓	RacerDroid [224]	✓					✓
DiagDroid [112]			✓				MOTIF [71]						✓
DRUN [220]	✓						GAT [246]						✓
Zhang et al. [265]						✓	Jabbarvand et al. [103]				✓		✓
Qian et al. [196]			✓			✓	Ermuth et al. [51]		✓				
Zhang et al. [267]			✓				Zhang et al. [268]					✓	
dLens [233]				✓			Packevičius et al. [195]						✓
Knorr et al. [118]		✓					IntentDroid [91]		✓				
Farto et al. [57]			✓				Bielik et al. [33]	✓					
MobiGUITAR [18]						✓	Aktouf et al. [10]						✓
AppAudit [247]						✓	Hassanshahi et al. [90]		✓				
iMPAcT [186]						✓	Deng et al. [47]						✓
Espada et al. [52]				✓			Zhang et al. [266]			✓			
QUANTUM [261]						✓	CRAXDroid [260]		✓	✓			✓
IntentFuzzer [253]		✓				✓	Vikomir et al. [231]					✓	
Shahriar et al. [207]		✓	✓			✓	APSET [204]		✓				
DROIDRACER [166]	✓						AppACTS [100]					✓	
CAFA [95]	✓						Guo et al. [87]		✓				
Griebe et al. [84]						✓	PBGT [46]					✓	
Banerjee et al. [29]				✓			A5 [230]		✓				
Suarez et al. [219]		✓					Linares et al. [152]				✓		
Sasnauskas et al. [205]						✓	AMDetector [269]		✓				
RERAN [70]						✓	Yang et al. [254]			✓			✓
DroidTest [201]		✓					Appstrument [189]			✓			
Avancini et al. [25]		✓					LEAKDROID [250]			✓			
Mahmood et al. [164]		✓	✓				Franke et al. [60]	✓					
Dhanapal et al. [48]			✓				SmartDroid [272]						✓
JarJarBinks [167]						✓	Hu et al. [96]						✓
Count								7	18	13	5	4	27

systematically through running test scenarios against Android apps. Bielik et al. [33] later have proposed a novel algorithm for scaling the inference of happens-before relations. Hu et al. [98] present a work for verifying and reproducing event-based races, where they have found that both imprecise Android component modelling and implicit happens-before relation could result in false positive for detecting potential races.

Security. As shown by Li et al. [139], the Android research community is extensively working on providing tools and approaches for solving various security problems for Android apps. Some of these works involve app testing, e.g., to observe defective behaviour [260] and malicious behaviour [164], track data leaks [201]. For example, Yan et al. [250] have built a novel and comprehensive approach for the detection of resource leaks using test criteria based on neutral cycles: sequences of GUI events should have a "neutral" effect and should not increase the usage of resources. Hay et al. [91] dynamically detect inter-application communication vulnerabilities in Android apps.

Performance. Android apps are sensitive to performance issues. When a program thread becomes expensive, the system may stop app execution after warning on the user interface that the "Application [is] Not Responding". The literature includes several contributions on highlighting issues related to the performance of Android apps such as poor responsiveness [111] and exception handling [266]. Yang et al. [254], for example, have proposed a systematic testing approach to uncover and quantify common causes of poor responsiveness of Android apps. Concretely, they explicitly extend the delay for typical problematic operations, using the test amplification approach, to demonstrate the effects of expensive actions that can be observed by users.

Energy. One of the biggest differences between traditional PC and portable devices is the fact that portable devices may run on battery power which can get depleted during app usage. A number of research works have investigated energy consumption hotspots arising from software design defects, unwanted *service* execution (e.g., advertisement), or have leveraged energy fingerprints to detect mobile malware. As an example, Wan et al. [233] present a technique for detecting display energy hotspots to guide the developers to improve the energy efficiency of their apps. Since each activity performed on a battery powered device drains a certain amount of energy from it, if the normal energy consumption is known for a device, the additionally used energy should be flagged as abnormal.

Compatibility. Android apps are often suffering from compatibility issues, where a given app can run successfully on a device, characterized by a range of OS versions while failing on others [240]. This is mainly due to the fragmentation in the Android ecosystem brought by its open source nature. Every vendor, theoretically, can have its own customized system (e.g., for supporting specific low-level hardware) and the screen size of its released devices can vary as well. To address compatibility problems, there is a need to devise scalable and efficient approaches for performing compatibility testing before releasing an app into markets. Indeed, as pointed out by Vilkomir et al. [231], it is expensive and time-consuming to consider testing all device variations. The authors thus proposed to address the issue with a combinatorial approach, which attempts to select an optimal set of mobile devices for practical testing. Zhang et al. [268] leverage a statistical approach to optimize the compatibility testing strategy where the test sequence is generated by K-means statistic algorithm.

Bug/Defect[14]. Like most software, Android apps are often buggy, usually leading to runtime crashes. Due to the high competition of apps in the Android ecosystem, defect identification is critical since they can be detrimental to user rating and adoption [114]. Indeed, researchers in this field leverage various testing techniques such as fuzzing testing, mutation testing, and search-based testing to dynamically explore Android apps to pinpoint defective behaviour [260], GUI bugs [96], Intent defects [205], crashing faults [169], etc.

Table 3.3 characterizes the publications selected for our SLR in terms of the objectives discussed above. Through our in-depth examination, the most considered testing objective is bug/defect, accounting for 23.3% of the selected publications.

3.5.1.2 Test targets

Test approaches in software development generally target core functionality code. Since Android apps are written in Java, the literature on Android app testing focused on Android specificities, mainly on how to address the GUI testing with a complex event mechanism as well as inter-component and inter-application communications.

GUI/Event. Android implements an event-driven graphical user interface system, making Android apps testing challenging, since they intensively interact with user inputs, introducing uncertainty and non-determinism. It is generally complicated to model the UI/system events because it not only needs the knowledge of the set of GUI widgets and their supporting actions (e.g., click for buttons) but also requires the knowledge of system events (e.g., receiving a phone call) which however are usually unknown in advance. Consequently, it is generally difficult to assemble a valid set of input event sequences for a given Android app with respect to *coverage*, *precision*, and *compactness* test criteria [256]. The Android testing community has proposed many approaches to address this challenge. For example, Android-GUITAR, an extension of the GUITAR tool [173] was proposed to model the structure and execution behaviour of Android GUI through a formalism called GUI

[14] Terminologically, the aforementioned objectives could also be categorised as bug/defect problems (e.g., concurrency issues). To make the summarisation more meaningful in this work, we only flag publications as bug/defect as long as their main focuses are bug/defect problems, e.g., when they address the gap between app's misbehaviour and developer's original design.

forests and event-flow graphs. Denodroid [163] applies a dynamic approach to generate inputs by instrumenting the Android framework to record the reaction of events.

ICC/IAC. The Inter-Component Communication (ICC) and Inter-Application communication (IAC[15]) enable a loose coupling among components [131,191], thus reducing the complexity to develop Android apps with a generic means to reuse existing functionality (e.g., obtain the contact list). Unfortunately, ICC/IAC also come with a number of security issues, among which the potential for implementing component hijacking, broadcast injection, etc. [133]. Researchers have then investigated various testing approaches to highlight such issues in Android apps. IntentDroid [91], for instance, performs comprehensive IAC security testing for inferring Android IAC integrity vulnerabilities. It utilizes lightweight platform-level instrumentation, which is implemented through debug breakpoints, to recover IAC-relevant app-level behaviour. IntentFuzzer [253], on the other hand, leverages fuzz testing techniques to detect capability leaks (e.g., permission escalation attacks) in Android apps.

General For all other publications which did not address the above two popular targets, the category *General* applies. Publications with targets like normal code analysis are grouped into this category.

Table 3.4 characterizes the test targets discussed above. The most frequently addressed testing target is GUI/Event, accounting for 45.6% of the selected publications. Meanwhile, there are only 12 publications targeted ICC/IAC. 44 publications are regrouped under the *General* category.

Insights from RQ1 - on Targets and Objectives

– "Bug/defect" has been the most trending concern among Android research community. "Compatibility" testing which is necessary for detecting issues that plague the Android fragmented ecosystem remains under-studied. Similarly, we note that because mobile devices are quickly getting powerful, developers build increasingly complex apps with services exploring hardware multi-core capabilities. Therefore, the community should invest more efforts in approaches for concurrency testing.
– Our review has also confirmed that GUI is of paramount importance in modern software development for guaranteeing a good user experience. In Android apps, the GUI actions and reactions are intertwined with the app logic, increasing the challenges of analysing app codes for defects. For example, modelling GUI behaviour while taking into account potential runtime interruption by system events (e.g., incoming phone call) is necessary, yet not trivial. These challenges have created opportunities in Android research: as our literature review shows, most test approaches target GUI or the Event mechanism. The community now needs to focus on transforming the approaches into scalable tools that will perform deeper security analyses and accurate defect identification in order to improve the overall quality of apps distributed in markets.

3.5.2 Which Test Levels are Addressed?

Development of Android apps involves classical steps of traditional software development. Therefore, there are opportunities in various phases to perform tests with specific emphasis and purpose. The Software testing community commonly acknowledges four levels of software testing [62,149]. Our literature review has identified that Android researchers have proposed approaches which considered *Unit/Regression testing*, *Integration testing*, and *System testing*. *Acceptance testing*, which involves end-users evaluating whether the app complies with their needs and requirements, still faces a lack of research effort in the literature.

Unit Testing is usually applied at the beginning of the development of Android apps, which are usually written by developers and can be taken as a type of white-box testing. Unit testing intends to ensure that every functionality, which could be represented as a function or a component, works

[15]IAC is actually ICC where the communicating components are from different apps.

properly (i.e., in accordance with the test cases). The main goal of unit testing is to verify that the implementation works as intended. Regression testing consists in re-executing previously executed test cases to ensure that subsequent updates of the app code have not impacted the original program behaviour, allowing issues (if presented) to be resolved as quickly as possible. Usually, regression testing is based on unit testing. It re-executes all the unit test cases every time when a piece of code is changed. As an example, Hu et al. [96] have applied unit testing to automatically explore GUI bugs, where JUnit, a unit testing framework, is leveraged to automate the generation of unit testing cases.

Integration Testing. Integration testing combines all units within an app (iteratively) to test them as a group. The purpose of this phase is to infer interface defects among units or functions. It determines how efficient the units are interactive. For example, Yang et al. [253] have proposed a tool called IntentFuzzer to test the capability problems involved in inter-component communication.

System Testing. System testing is the first step that the whole app is tested as a whole. The goal of this phase is to assess whether the outlined requirements and quality standards have been fulfilled. Usually, system testing is done in a black-box style, which is usually conducted by independent testers who have no knowledge of the apps to be tested. As an example, Mao et al. [169] have proposed a testing tool named Sapienz that combines several approaches including fuzzing testing, search-based testing to systematically explore faults in Android apps.

Table 3.5 summarises the aforementioned test phases, where the most recurrently applied testing phase is system testing (accounting for nearly 80% of the selected publications), followed by unit testing and integration testing, respectively.

Insights from RQ2 - on Test Levels

– The large majority of approaches reviewed in this SLR are about testing the whole app against given test criteria. This correlates with the test methodologies detailed below. Unit and regression testing, which would help developers assess individual functionalities in a white-box testing scenario, are limited to a few approaches.

3.5.3 How are the Test Approaches Built?

Our review further investigates the approaches in-depth to characterize the methodologies they leverage, the type of tests that are implemented as well as the tool support they have exploited. In this work, we refer to *test technique* as a broad concept to describe all the technical aspects related to testing, while we constrain the term *test methodology* to specifically describe the concrete methodology that a test approach applies.

3.5.3.1 Test methodologies

Table 3.6 enumerates all the testing methodologies we observed in our examination.

Model-based Testing is a testing methodology that goes one step further than traditional methodologies by automatically generating test cases based on a model, which describes the functionality of the system under test. Although such methodology incurs a substantial, usually manual, effort to design and build the model, the eventual test approach is often extensive, since test cases can be automatically generated and executed. Our review has revealed that model-based testing is the most common methodology used in Android testing literature: 63% of publications involve some model-based testing steps. Takala et al. [221] present a comprehensive documentation on their experiences in applying a model-based GUI testing to Android apps. They typically discuss how

model-based testing and test automation are implemented, how apps are modelled, as well as how tests are designed and executed.

Search-based Testing is using the metaheuristic search techniques to generate software tests [237], with the aim to detect as many bugs as possible, especially the most critical ones, in the system under test. In [165], the authors developed an evolutionary testing framework for Android apps. Evolutionary testing is a form of search-based testing, where an individual corresponds to a test case, and a population comprised of many individuals is evolved according to certain heuristics to maximize the code coverage. Their technique thus tackles the common shortcoming of using evolutionary techniques for system testing. In order to generate the test suites in an effective and efficient way, Amalfitano et al. [13] proposed a novel search-based testing technique based on the combination of genetic and hill climbing techniques.

Random Testing is a software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is a *pass* or a *fail* [88]. In the absence of specifications, program exceptions are used to detect test case *fails*. Random testing is also acquired by almost all other test suite generation methodologies and serves as a fundamental technique. Random testing has been used in several literature works [8, 146, 160, 163, 274].

Fuzzing Testing is a testing technique that applies invalid, unexpected, or random data as inputs to a testing object. It is commonly used to test for security problems in software or computer systems. The main focus then shifts to monitoring the program for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. A number of research papers (e.g., [96, 271]) have explored this type of testing via automated or semi-automated *fuzzing*. Fuzzing testing is slightly different from random testing, as it mainly embraces, usually on purpose, unexpected, invalid inputs and focuses on monitoring crashes/exceptions of the tested apps while random testing does not need to conform to any of such software specifications.

A/B Testing provides a means for comparing two variants of a testing object, and hence determining which of the two variants is more effective. A/B testing is recurrently used for statistical hypothesis tests. In [9], Adinata et al. have applied A/B testing to test mobile apps, where they have solved three challenges of applying A/B testing, including element composition, variant delivery and internet connection. Holzmann et al. [94] conduct A/B testing through a multivariate testing tool.

Concolic Testing is a hybrid software verification technique that performs symbolic execution, a classical technique which treats program variables as symbolic variables, along with a concrete execution path (testing on particular inputs). Anand et al. [19] propose a concolic testing technique, CONTEST, to alleviate the path explosion problem. They develop a concolic-testing algorithm to generate sequences of events. Checking the subsumption condition between event sequences allows the algorithm to trim redundant event sequences, thereby, alleviating path explosion.

Mutation Testing is used to evaluate the quality of existing software tests. It is performed by selecting a set of mutation operators and then applying them to the source program, one operator at a time, for each relevant program location. The result of applying one mutation operator to the program is called a mutant. If the test suite is able to detect the change (i.e., one of the tests fails), then the mutant is said to be killed. In order to realize an end-to-end system testing of Android apps in a systematic manner, Mahmood et al. [165] propose EvoDroid, an evolutionary approach of system testing of apps, in which two types of mutation (namely, input genes and event genes) are leveraged to identify a set of test cases that maximize code coverage. Mutation testing-based approaches are however not common in the Android literature.

Overall, our review has shown that the literature often combines several methodologies to improve test effectiveness. In [146], the authors combined model-based testing with random testing to complete the testing. Finally, EvoDroid [165] is a framework that explores model-based, search-based and mutation testing techniques.

3.5.3.2 Test types

In general, there are three types of testing, namely the *White-box testing*, *Black-box testing*, and *Grey-box testing*. Table 3.7 summarizes these testing types by emphasizing on the ideal tester (the software developer or a third-party), on whether knowledge on implementation details is fully/partially/not required.

White-box testing is a scenario in which the software is examined based on the knowledge of its implementation details. It is usually applied by the software developers in early development stages when performing *unit testing*. Another common usage scenario is to perform thorough tests once all software components are assembled (known as *regression testing*). In this SLR, when an approach requires app source (or byte) code knowledge, whether obtained directly or via reverse engineering, we consider it a white-box approach.

Black-box testing, on the other hand, is a scenario where internal design/implementation of the tested object is not required. Black-box testing is often conducted by third-party testers, who have no relationships with the developers of tested objects. If an Android app testing process only requires the installation of the targeted app, we reasonably put it under this category.

Grey-box testing is a trade-off between white-box testing and black-box. It does not require the testers to have full knowledge on the source code where white-box testing needs. Instead, it only needs the testers to know some limited specifications like how the system components interact. For the investigations of our SLR, if a testing approach requires to extract some knowledge (e.g., from the Android manifest configuration) to guide its tests, we consider it a grey-box testing approach.

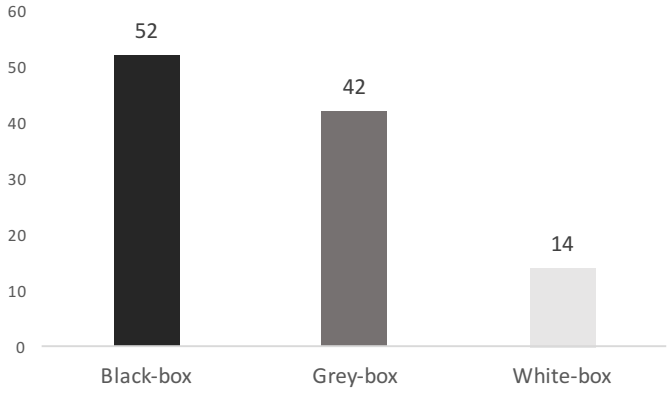


Figure 3.7: Breakdown of examined publications regarding their applied testing types.

Fig. 3.7 illustrates the distribution of test types applied by examined testing approaches. White-box testing is the least used type, far behind black-box and grey-box testing. This is expected because Android apps are usually compiled and distributed in APK format, so testers in most scenarios have no access to source code. We also wish to address that one literature can make use of more than one testing type, this is why the sum of the three types in Fig. 3.7 is larger than 103.

3.5.3.3 Test environments

Unlike static analysis of Android apps [139], testing requires to actually run apps on an execution environment such as a *real device* or an *emulator*.

Real Device has a number of advantages: they can be used to test apps w.r.t compatibility aspects [100, 231, 268], energy consumption [29, 152, 233], and the poor responsiveness issue [111, 254]. Unfortunately, using real devices is not efficient, since it cannot scale in terms of execution time and resources (several devices may be required).

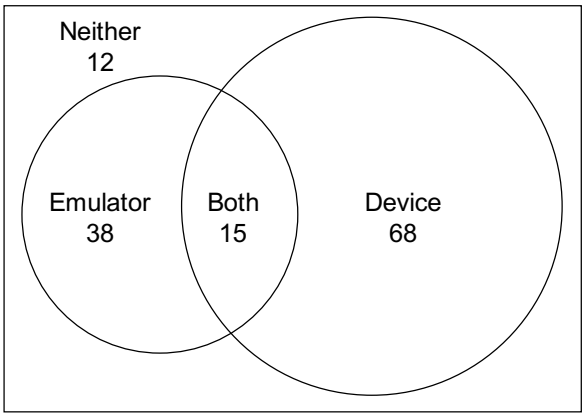


Figure 3.8: Venn Diagram of Testing Environment.

Emulator, on the contrary, can be scalable. When deployed on the cloud, using the emulator can grant a tester great computing resources and carry out parallel tests at a very large scale [164]. Unfortunately, emulators are ineffective for security-relevant tests, since some malware have the functionality to detect whether they are running on an emulator. If so, they may decide to refrain from exposing their malicious intention [229]. Emulators also introduce huge overhead when mimicking real-life sensor inputs, e.g., requiring altering the apps under testing at source code level [85].

Emulator + Real Device, can be leveraged together to test Android apps. For example, one can first use an emulator to launch large-scale app testing for pre-selecting a subset of relevant apps and then resort to real devices for more accurate testing.

As can be seen from Figure 3.8, real devices are largely used by 68 publications in our final list. Only 38 publications used emulators, despite the fact that they are cheap. 15 publications chose both environments to avoid disadvantages of either. Deducting these 15 publications, we can calculate that 23 publications focused solely on emulators, where 53 publications selected real devices as the only environment.

3.5.3.4 Tool support

While performing the SLR, we have observed that several publicly available tools were recurrently leveraged to implement or complement the state-of-the-art approaches. Table 3.8 enumerates such tools with example references to works where they are explored.

AndroidRipper is a tool for automatic GUI testing of Android apps. It is driven by a user-interface ripper that automatically and systematically travels the app's GUI aiming at exercising a given app in a structured way. In order to generate test cases in an effective and efficient way, Amalfitano et al. [13] extend this work with search-based testing techniques, where genetic and hill climbing algorithms are considered.

EMMA is an open-source toolkit for measuring and reporting Java code coverage. Since Android apps are written in Java, researchers often use EMMA to compute the code coverage of their Android app testing approaches, including EvoDroid [165] and SIG-Droid [180].

Monkey is a test framework released and maintained by Google, the official maintainer of Android. It generates and sends pseudo-random streams of user/system events into the running system. This functionality is exploited in the literature to automatically identify defects of ill-designed apps. As an example, Hu et al. [96] leveraged Monkey to identify GUI bugs of Android apps. The randomly generated test cases (events) are fed into a customized Android system that produces log/trace files during the test. Those log/trace files can then be leveraged to perform post analysis and thereby to discover event-related bugs.

RERAN is a record and replay tool for testing Android apps. Unlike traditional record-and-reply tools, which are inadequate for Android apps because of their expressiveness on smartphone features, RERAN supports sophisticated GUI gestures and complex sensor events. Moreover, RERAN achieves accurate timing requirements among various input events. A^3E [26] for example uses RERAN to record its targeted and depth-first exploration for systematic testing of Android apps. Those recorded explorations can later be replayed so that to benefit debuggers in quickly localizing the exact event stream that has led to the crash.

Robotium is an open-source test framework, which has full support for native and hybrid apps. It also eases the way to write powerful and robust automatic black-box UI tests of android apps. SIG-Droid [180] for example leverages Robotium to execute its generated test cases (with the help of symbolic execution). We have found during our review that Robotium were most frequently leveraged by state-of-the-art testing approaches.

Robolectric is a unit testing framework, which simulates the Android execution environment (either on a real device or on an emulator) in a pure Java environment. The main advantage of doing that is to improve the testing efficiency because tests running inside a JVM are much faster than that of running on an Android device (or even emulator), where it usually takes minutes to build, deploy and launch an app. Sadeh et al. [203] have effectively used Robolectric framework to conduct unit testing for their calculator application. They have found that it is rather easy to write test cases with this framework, which requires only a few extra steps and abstractions. Because testers do not need to maintain a set of fake objects and interfaces, it is even preferable for complex apps.

Sikuli uses visual technology to automate GUI testing through screenshot images. It is particularly useful when there is no easy way to obtain the app source code or the internal structure of graphic interfaces. Lin et al. [150, 151] leveraged Sikuli in their work to enable record-and-replay testing of Android apps, where the user interactions are saved beforehand in Sikuli test formats (as screenshot images).

Insights from RQ3 - on Used Techniques

– Given the complexity of interactions among components in Android apps as well as with the operating system, it is not surprising that most approaches in the literature resort to "model-based" techniques which build models for capturing the overall structure and behaviour of apps to facilitate testing activities (e.g., input generation, execution scenarios selection, etc.).
– The unavailability of source code for market apps make white-box techniques less attractive than grey-box and black-box testing for assessing apps in the wild. Nevertheless, our SLR shows that the research community has not sufficiently explored testing approaches that would directly benefit app developers during the development phase.
– Tool support for building testing approaches is abundant. The use of the Robotium open source test framework by numerous approaches once again demonstrates the importance of making tools available to stimulate research.

3.5.4 To What Extent are the Approaches Validated?

Several aspects must be considered when assessing the effectiveness of a testing approach. We consider in this SLR the measurements performed on *code coverage* as well as on *accuracy*. We also investigate the use of a *ground truth* to validate performance scores, the *size of the experimental dataset*.

Coverage is a key aspect for estimating how well the program is tested. Larger coverage generally correlates with higher possibilities of exposing potential bugs and vulnerabilities, as well as uncovering malicious behaviour. There are numerous coverage metrics leveraged by state-of-the-art works. For example, for evaluating **Code Coverage**, metrics such as *LoC* (Lines of Code) [13, 165, 169], *Block* [274], Method [26, 146], Branch [41] have been proposed in our community. In order to profile

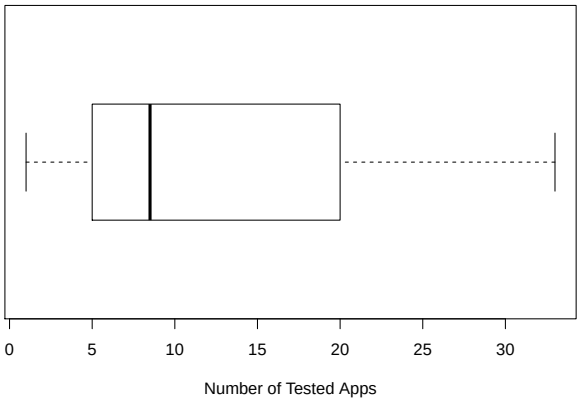


Figure 3.9: The distribution of the number of tested apps (outliers are removed).

the **Accuracy** of testing approaches, other coverage metrics are also proposed in the literature such as bugs [233] and vulnerabilities [91] (e.g., *how many known vulnerabilities can the evaluated testing approach cover?*). Table 3.9 enumerates the coverage metrics used in the literature, where *LoC* appears to be the most concerned metric.

Ground Truth refers to a reference dataset where each element is labelled. In this SLR, we consider two types of ground truths. The first is related to malware detection approaches: the ground truth then contains apps labelled as benign or malicious. As an example, the Drebin [21] dataset has recurrently been leveraged as ground truth to evaluate testing approaches [213]. The second is related to vulnerability and bug detection: the ground truth represents code that is flagged to be vulnerable or buggy based on the observation of bug reports summited by end users or bug fix histories committed by developers [96, 266].

Dataset Size The *Dataset Size* is the number of apps tested in the experimental phase. We can see from Fig. 3.9 that most works (ignoring outliers) carried out experiments on no more than 100 apps, with a median number of 8 apps. Comparing to the distribution of the number of evaluated apps summarized in an SLR of static analysis of Android apps [139], where the median and maximum numbers are respectively 374 and 318,515, far bigger than the number of apps considered by testing approaches. This result is somehow expected as testing approaches (or dynamic analysis approaches) are generally not scalable.

Insights from RQ4 - on Approach Validation

Although literature works always provide evaluation section to provide evidence (often through comparison) that their approaches are effective, their reproducibility is still challenged by the fact that there is a lack of established ground truth and benchmarks. Yet, reproducibility is essential to ensure that the field is indeed progressing based on a baseline performance, instead of relying on subjective observation by authors and on datasets with variable characteristics.

Table 3.4: Test targets in the literature.

Tool	GUI/Event	ICC/IAC	General	Tool	GUI/Event	ICC/IAC	General
Zeng et al. [263]	✓			Dagger [252]			✓
Malisa et al. [168]	✓			CRASHSCOPE [183]	✓		
MAMBA [113]			✓	Pretect [111]	✓		
DroidMate [104]	✓			SSDA [97]			✓
TrimDroid [181]	✓			ERVA [98]			✓
Clapp et al. [43]	✓			SAPIENZ [169]			✓
RacerDroid [224]			✓	Baek et al. [27]	✓		
DiagDroid [112]	✓			MobiPlay [198]	✓		
MOTIF [71]			✓	DRUN [220]			✓
DroidDEV [20]	✓			GAT [246]	✓		
Zhang et al. [265]	✓			Jabbarvand et al. [103]			✓
Qian et al. [196]			✓	Ermuth et al. [51]			✓
Cadage [274]	✓			Zhang et al. [267]			✓
Zhang et al. [268]			✓	dLens [233]			✓
Sonny et al. [211]			✓	Packevičius et al. [195]	✓		
SIG-Droid [180]			✓	Knorr et al. [118]			✓
TAST [107]			✓	IntentDroid [91]		✓	
Griebe et al. [85]	✓			Farto et al. [57]			✓
Bielik et al. [33]			✓	MobiGUITAR [18]	✓		
AGRippin [13]	✓			Aktouf et al. [10]		✓	
THOR [8]	✓			AppAudit [247]			✓
Morgado et al. [185]	✓			Hassanshahi et al. [90]		✓	
iMPAcT [186]	✓			Deng et al. [47]			✓
Espada et al. [52]			✓	Zhang et al. [266]			✓
QUANTUM [261]	✓			CRAXDroid [260]			✓
IntentFuzzer [253]		✓		Vikomir et al. [231]			✓
Shahriar et al. [207]			✓	APSET [204]		✓	
DROIDRACER [166]			✓	EvoDroid [165]			✓
SPAG-C [151]	✓			Caiipa [148]			✓
UGA [146]		✓		AppACTS [100]			✓
CAFA [95]			✓	Holzmann et al. [94]	✓		
Guo et al. [87]		✓		Griebe et al. [84]			✓
PBGT [46]	✓			Chen et al. [40]			✓
Banerjee et al. [29]			✓	Amalfitano et al. [12]	✓		
Adinata et al. [9]			✓	A5 [230]		✓	
Suarez et al. [219]	✓			Linares et al. [152]	✓		
Sasnauskas et al. [205]		✓		AMDetector [269]			✓
RERAN [70]	✓			Yang et al. [254]	✓		
ORBIT [256]	✓			DroidTest [201]			✓
Appstrument [189]			✓	Dynodroid [163]	✓		
SPAG [150]	✓			SwiftHand [41]	✓		
A^3E [26]	✓			Avancini et al. [25]		✓	
Amalfitano et al. [15]	✓			SALES [45]			✓
LEAKDROID [250]	✓			GUIdiff [32]	✓		
Collider [106]	✓			Mirzaei et al. [182]			✓
JPF-Android [175]		✓		Mahmood et al. [164]			✓
MASHTE [116]	✓			Franke et al. [60]			✓
Dhanapal et al. [48]			✓	ACTEve [19]	✓		
SmartDroid [272]	✓			JarJarBinks [167]		✓	
TEMA [221]	✓			Sadeh et al. [203]	✓		
Hu et al. [96]	✓			A2T2 [14]	✓		
ART [160]	✓						
		Count			47	12	44

Table 3.5: Recurrent testing phases.

Tool	Unit/Regression	Integration	System	Tool	Unit/Regression	Integration	System
Zeng et al. [263]			✓	Dagger [252]			✓
Malisa et al. [168]			✓	CRASHSCOPE [183]			✓
MAMBA [113]			✓	Pretect [111]			✓
DroidMate [104]			✓	SSDA [97]			✓
TrimDroid [181]			✓	ERVA [98]			✓
Clapp et al. [43]			✓	SAPIENZ [169]			✓
RacerDroid [224]			✓	Baek et al. [27]			✓
DiagDroid [112]			✓	MobiPlay [198]			✓
MOTIF [71]			✓	DRUN [220]			✓
DroidDEV [20]			✓	GAT [246]			✓
Zhang et al. [265]			✓	Jabbarvand et al. [103]			✓
Qian et al. [196]			✓	Ermuth et al. [51]	✓		
Cadage [274]			✓	Zhang et al. [267]	✓		
Zhang et al. [268]			✓	dLens [233]			✓
Sonny et al. [211]			✓	Packevičius et al. [195]			✓
SIG-Droid [180]			✓	Knorr et al. [118]			✓
TAST [107]			✓	IntentDroid [91]		✓	
Griebe et al. [85]	✓			Farto et al. [57]			✓
Bielik et al. [33]			✓	MobiGUITAR [18]	✓		
AGRippin [13]			✓	Aktouf et al. [10]			✓
THOR [8]			✓	AppAudit [247]	✓		
Morgado et al. [185]			✓	Hassanshahi et al. [90]			✓
iMPAcT [186]			✓	Deng et al. [47]			✓
Espada et al. [52]			✓	Zhang et al. [266]		✓	
QUANTUM [261]			✓	CRAXDroid [260]	✓		✓
IntentFuzzer [253]		✓		Vikomir et al. [231]			✓
Shahriar et al. [207]	✓			APSET [204]			✓
DROIDRACER [166]			✓	EvoDroid [165]			✓
SPAG-C [151]			✓	Caiipa [148]			✓
UGA [146]			✓	AppACTS [100]			✓
CAFA [95]			✓	Holzmann et al. [94]			✓
Guo et al. [87]			✓	Griebe et al. [84]	✓	✓	
PBGT [46]			✓	Chen et al. [40]			✓
Banerjee et al. [29]			✓	Amalfitano et al. [12]	✓		
Adinata et al. [9]			✓	A5 [230]			✓
Suarez et al. [219]			✓	Linares et al. [152]	✓		
Sasnauskas et al. [205]			✓	AMDetector [269]			✓
RERAN [70]			✓	Yang et al. [254]			✓
ORBIT [256]		✓		DroidTest [201]	✓		
Appstrument [189]	✓			Dynodroid [163]			✓
SPAG [150]			✓	SwiftHand [41]			✓
A^3E [26]			✓	Avancini et al. [25]			✓
Amalfitano et al. [15]			✓	SALES [45]			✓
LEAKDROID [250]			✓	GUIdiff [32]	✓		
Collider [106]			✓	Mirzaei et al. [182]			✓
JPF-Android [175]			✓	Mahmood et al. [164]			✓
MASHTE [116]	✓	✓		Franke et al. [60]	✓		
Dhanapal et al. [48]			✓	ACTEve [19]			✓
SmartDroid [272]			✓	JarJarBinks [167]			✓
TEMA [221]	✓			Sadeh et al. [203]	✓	✓	
Hu et al. [96]	✓			A2T2 [14]	✓		
ART [160]			✓				
Count					19	7	81

Table 3.6: Test method employed in the literature.

Tool	Model-based	Search-based	Random	Fuzzing	A/B	Concolic	Mutation	Tool	Model-based	Search-based	Random	Fuzzing	A/B	Concolic	Mutation
Zeng et al. [263]			✓					Dagger [252]	✓						
Malisa et al. [168]	✓							CRASHSCOPE [183]	✓						
MAMBA [113]	✓							DroidMate [104]	✓						
SSDA [97]	✓			✓				TrimDroid [181]	✓						
ERVA [98]	✓							Clapp et al. [43]	✓						
SAPIENZ [169]		✓		✓				RacerDroid [224]	✓						
Baek et al. [27]	✓							DiagDroid [112]			✓				
MOTIF [71]	✓							DRUN [220]	✓						
DroidDEV [20]	✓							GAT [246]	✓						
Zhang et al. [265]	✓							Jabbarvand et al. [103]	✓						
Qian et al. [196]	✓							Ermuth et al. [51]	✓						
Cadage [274]	✓							Zhang et al. [267]	✓						
Zhang et al. [268]	✓							dLens [233]	✓						
Sonny et al. [211]	✓							Packevičius et al. [195]	✓						
SIG-Droid [180]	✓							TAST [107]	✓						
IntentDroid [91]				✓				Farto et al. [57]	✓						
Bielik et al. [33]	✓							MobiGUITAR [18]	✓						
AGRippin [13]		✓					✓	Aktouf et al. [10]	✓						
THOR [8]			✓					AppAudit [247]	✓						
Morgado et al. [185]	✓							Hassanshahi et al. [90]	✓			✓			
iMPAcT [186]	✓							Deng et al. [47]							✓
Espada et al. [52]	✓							Zhang et al. [266]			✓				
QUANTUM [261]	✓							CRAXDroid [260]				✓			
IntentFuzzer [253]				✓				Shahriar et al. [207]	✓			✓			
APSET [204]	✓							DROIDRACER [166]	✓						
EvoDroid [165]	✓	✓					✓	SPAG-C [151]	✓						
Caiipa [148]				✓				UGA [146]	✓		✓				
AppACTS [100]			✓					Holzmann et al. [94]					✓		
Guo et al. [87]	✓							Griebe et al. [84]	✓						
PBGT [46]	✓							Amalfitano et al. [12]	✓						
Adinata et al. [9]					✓			A5 [230]			✓				
Suarez et al. [219]	✓							Linares et al. [152]	✓						
Sasnauskas et al. [205]	✓			✓				AMDetector [269]	✓						
RERAN [70]	✓							Yang et al. [254]	✓						
ORBIT [256]	✓							Dynodroid [163]			✓				
SwiftHand [41]	✓							A^3E [26]	✓						
Avancini et al. [25]	✓							SALES [45]	✓						
LEAKDROID [250]	✓							GUIdiff [32]			✓				
Collider [106]	✓					✓		JPF-Android [175]	✓						
Mahmood et al. [164]	✓			✓				ACTEve [19]						✓	
SmartDroid [272]	✓							JarJarBinks [167]				✓			
TEMA [221]	✓							Hu et al. [96]			✓				
A2T2 [14]	✓							ART [160]			✓				
Count									65	3	11	11	2	2	3

Table 3.7: Common test types.

Testing Type	Ideal Tester	Implementation Knowledge
White-box	Developer	Known
Black-box	Independent Tester	Unknown
Grey-box	Independent Tester	Partially Known

Table 3.8: Summary of basic tools that are frequently leveraged by other testing approaches.

Tool	Brief Discription	Example Usages
AndroidRipper	An automated GUI-based testing tool	Yang et al. [254], Amalfitano et al. [12,15], AGRippin [13], MobiGUITAR [18]
EMMA	A free Java code coverage measuring tool	Mirzaei et al. [182], Mahmood et al. [164], SIGDroid [180], BBOXTESTER [271], EvoDroid [165]
Monkey	An automated testing tool that generates and executes randomly generated test cases	Hu et al. [96], BBOXTESTER [271], TAST [107],
RERAN	A timing- and touch-sensitive record and replay tool for Android apps	UGA [146], dLens [233], A^3E [26]
Robotium	An open-source test framework for writing automatic black box testing cases for Android apps	A2T2 [14],Chen et al. [40], UGA [146], THOR [8], Yang et al. [254], ORBIT [256], Mahmood et al. [164], AGRippin [13], Guo et al. [87], SIGDroid [180]
Robolectric	A unit test framework that enables tests run inside JVM instead of DVM	Sadeh et al. [203], Mirzaei et al. [182]
Sikuli	A visual technology to automate and test GUIs using screenshot images	SPAG [150], SPAG-C [151]

Table 3.9: Assessment Metrics (e.g., for Coverage, Accuracy).

Metrics (# of)	Example Publications
LoC	EvoDroid [165], AGRippin [13], THOR [8] Zeng et al. [263], SAPIENZ [169]
Block	Cadage [274]
Branch	SwiftHand [41]
Method	UGA [146], A^3E [26]
Exception	Zhang et al. [266]
Action	ORBIT [256]
Activity	A^3E [26], Avancini et al. [25], Malisa et al. [168] MAMBA [113], Clapp et al. [43]
Service	Zhang et al. [267]
Bug	dLens [233], TEMA [221], Hu et al. [96], MobiGUITAR [18]
Defect	APSET [204]
Fault	QUANTUM [261], Vikomir et al. [231], Sonny et al. [211]
Crash	Shahriar et al. [207], Caiipa [148], .CRASHSCOPE [183]
Vulnerability	Sadeh et al. [203], IntentDroid [91]
Leakage	CRAXDroid [260], Yang et al. [253]

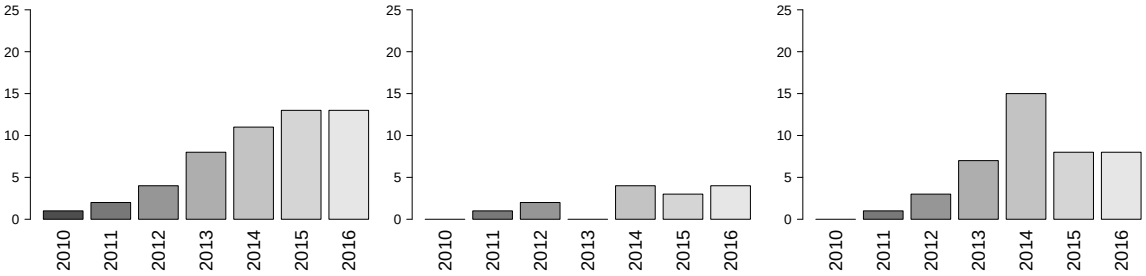
3.6 Discussion

Research on Android app testing has been prolific in the past years. Our discussion will focus on the trends that we observed while performing this SLR, as well as on the challenges that the community should still attempt to address.

3.6.1 Trend Analysis

The development of the different branches in the taxonomy is disparate.

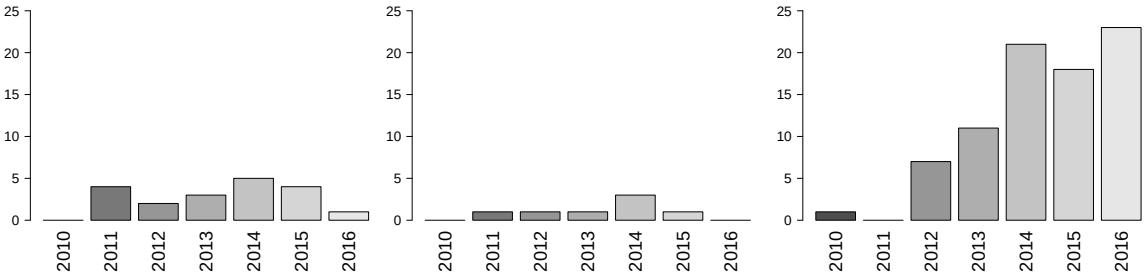
Fig. 3.10 illustrates the trend in testing types over the years. Together, black-box and grey-box testing are involved in 90% of the research works. Their evolution is thus reflected by the overall evolution of research publications (cf. Fig. 3.4). White-box testing remains low in all years.



a Black-box. **b** White-box. **c** Grey-box.

Figure 3.10: Trend of Testing Types.

Fig. 3.11 presents the evolution over time of works addressing different test levels. Unit/regression and integration testing phases include a low, but stable, number of works every year. Overall, system testing has been heavily used in the literature and has even doubled between 2012 and 2014. System testing of Android apps is favored since app execution is done on a specific virtual machine environment with numerous runtime dependencies: it is not straightforward to isolate a single block for unit/regression testing or to test the integration of two components without interference from other components. Nevertheless, with the increasing use of code instrumentation [132], there are new opportunities to eventually slice android apps for performing more grey-box and white-box testing.



a Unit/Regression. **b** Integration. **c** System.

Figure 3.11: Trend of Testing Levels.

Trend analysis of testing methods in Fig. 3.12 confirms that model-based testing is dominating in the literature of Android app testing, and its evolution is reflected in the overall evolution of testing approaches. Most approaches indeed start by constructing a GUI model or a call graph (CG) to generate efficient test cases. In the last couple of years, mutation testing has been appearing in the literature, similarly to search-based techniques.

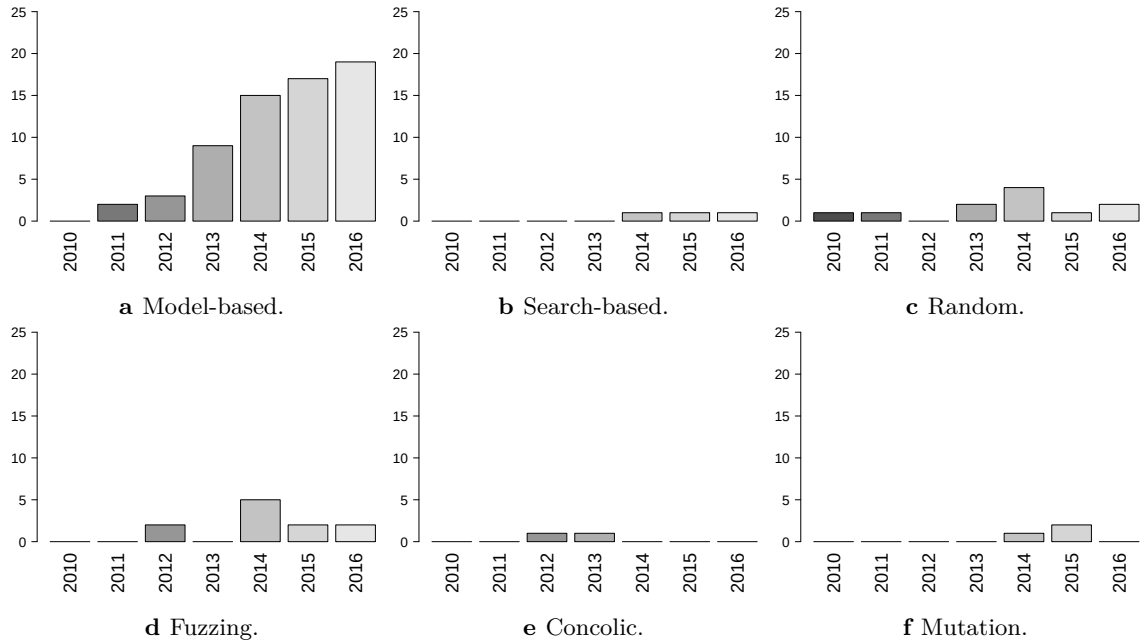


Figure 3.12: Trend of Testing Methods.

With regard to testing targets, Fig. 3.13(a-b) shows that the graphical user interfaces, as well as the event mechanism, are continuously at the core of research approaches. Since Android *Activities* (i.e., the UIs) are the main entry points for executing test cases, the community will likely continue to develop black-box and grey-box test strategies that increase interactions with GUI to improve code coverage. Inter-component and inter-application communications, on the other hand, have been popular targets around 2014.

With regard to testing objectives, Fig. 3.13(c-h) shows that *security* concerns have attracted a significant amount of research, although the output has been decreasing in the last couple of years. Bug/defect identification, however, has somewhat stabilized.

3.6.2 Evaluation of Authors

Android testing is a new field of research which has attracted several contributions over the years due to the multiple opportunities that it offers for researchers to apply theoretical advances in the domain of software testing. We emphasize the attractiveness of the field by showing in Fig. 3.14 the evolution of single authors contributing to research approaches. We count in each year, the *Total Authors* who participated in at least one of our selected publications, the *New Authors* that had had no selected publication until that year, and the number of *Stayed Authors* who had publications selected both that year and the years to come. Overall, the figures raise several interesting findings:

- Every year, the community of Android testing research authors is almost entirely renewed.
- Only a limited number of researchers publish again in the theme after one publication.

These facts may suggest that the research in Android app testing is often governed by opportunities. Furthermore, challenges (e.g., building a sound GUI event model) quickly arise, making authors lose interest in pursuing in this research direction. Although we believe that the fact that the topic is within reach of a variety of authors from other backgrounds is good for bringing new ideas and cross-fertilizing, the maturity of the field will require commitment from more authors staying in the field.

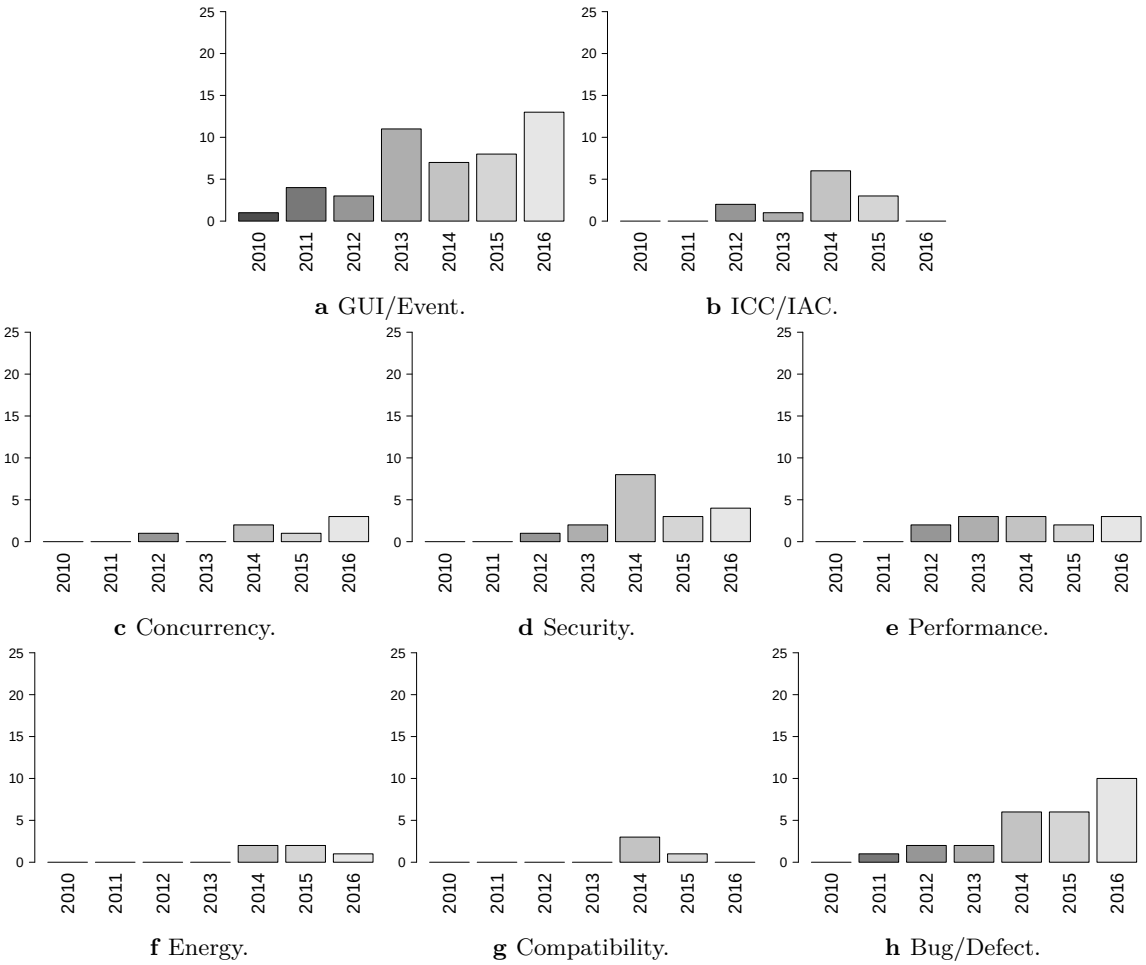


Figure 3.13: Trend of Testing Targets and Objectives.

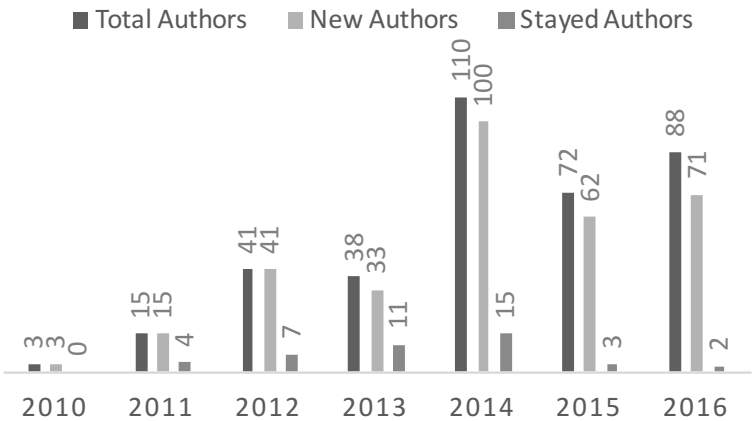


Figure 3.14: Trend in community authors. “New Authors” and “Stayed Authors" indicate the number of authors that enter the field (no relevant publications before) and have stayed in the field (they will keep publishing in the following years).

3.6.3 Research Output Usability

In the course of our investigations for performing the review, we have found that the research community on Android app testing seldom contributes with reusable tools (e.g., implementation of

approaches for GUI testing), not even mention to contribute with open source testing tools. Yet, the availability of such tools is necessary not only to limit the efforts in subsequent works but also to encourage true progress beyond the state-of-the-art.

Despite most testing approaches are not made publicly available, it is nevertheless gratifying to observe that some of them have been leveraged in industry. For example, research tool TEMA has now been integrated into the RATA project[16], where researchers (from Tampere University of Technology) and practitioners (from Intel Finland, OptoFidelity, and VTT) work together to provide robot-assisted test automation for mobile apps. Another research tool named SAPIENZ has led to a start-up called MaJiCKe and recently been acquired by Facebook London, being the core component of Facebook's testing solutions for mobile apps.

3.6.4 Open Issues and Future Challenges

Although the publications we chose all have their own solid contributions, some authors posed open issues and future challenges to call in more research attention to the domain. We managed to collect the concerns and summarized as follows:

- **Satisfying Fastidious Pre-conditions.** One recurrently discussed issue is to generate test cases that can appropriately satisfy pre-conditions such as login to an app. When the oracles generate events to traverse the activities of Android apps, some particular activities are extremely hard to be touched. A publicly known condition is to tap the same button for 7 consecutive times in order to trigger developer mode [180, 263]. Another example would be to break through the login page which requires a particular combination of user account and passwords. Both preconditions are clearly not easy to be satisfied during the process of testing Android apps.
- **Modelling Complex Events (e.g., Gestures or Non-user Events).** In addition to simple events such as clicking, Android OS also involves quite a lot of complex events such as user gestures (swipe, long press, zoom in/out, spin, etc.) and system events (network connectivity, events coming from light, pressure and temperature sensors, GPS, fingerprint recognizer, etc.). All the events would introduce non-deterministic behaviours if they are not properly modelled. Unfortunately, at the moment, most of our reviewed papers only tackle simple events like clicking, letting other events remain untouched [46, 85].
- **Bridging Incompatible Instruction Sets.** To improve the performance of Android apps, Google provides a toolset, i.e., the Android Native Developer Kit (NDK), allowing app developers to implement time-intensive tasks via C/C++. Those tasks implemented with C/C++ are closely dependent on the CPU instruction sets (e.g., Intel or ARM) and hence can only be launched in right instruction sets, e.g., tasks implemented based on the ARM architecture can only be executed on ARM-based devices). However, as most mobile devices nowadays are assembled with ARM chips while most PCs running Android emulators are assembled with Intel chips, running ARM-based emulators on Intel-based PCs are extremely slow, this gap has caused problems for emulator-based testing approaches [198].
- **Evaluating Testing Approaches Fairly.** Frequently, researchers complain about the fact that our community has not provided a reliable coverage estimator to approximate the coverage (e.g., code coverage) of testing approaches and to fairly compare them [111, 195, 263, 268]. Although some outstanding progress has been made for developing estimation tools [271], our SLR still indicates that there does not exist any universally acquired tool that supports fair comparison among testing approaches. We, therefore, urge our fellow researchers to appropriately resolve this open issue and subsequently contribute to our community a reliable artefact benefiting many aspects of future research studies.
- **Addressing Usability Defect.** The majority of the research studies focuses on functional defects of Android apps. The usability defect does not attract as much attention as the users are

[16] http://wiki.tut.fi/RATA/WebHome

concerned [47]. Usability defect like poor responsiveness [254] is a major drawback of Android apps and receives massive complaints from users. Bad view organization on the screen arising from incompatibility and repetitive imprecise recognition of user gestures also imply bad user experience.

3.6.5 New Research Directions

In light of the SLR summary of the state-of-the-art and considering the new challenges reported in the literature, there are opportunities for exploring new testing applications to improve the quality of Android apps or/and increase confidence in using them safely. We now enumerate three example directions:

3.6.5.1 Validation of app updates

Android app developers regularly update their apps for various reasons, including keeping them attractive to the user base[17]. Unfortunately, recent studies [225] have shown that updates of Android apps often come with more security vulnerabilities and functional defects. In this context, the community could investigate and adapt regression techniques for identifying defect-prone or unsafe updates. To accelerate the identification of such issues in updates, one can consider exploring approaches with behavioural equivalence, e.g., using "record and replay" test-case generation techniques.

3.6.5.2 Accounting for the ecosystem fragmentation

As previously highlighted, the fragmentation of the Android ecosystem (with a high variety in operating system versions where a given app will be running, as well as a diversity of hardware specifications) is a serious challenge for performing tests that can expose all issues that a user might encounter on his specific device runtime environment. There is still room to investigate test optimization and prioritization for Android to cover a majority of devices and operating system versions. For example, on top of modelling apps, researchers could consider modelling the framework (and its variabilities) and account for it during test execution.

3.6.5.3 Code prioritization vs test prioritization

Finally, we note that Android apps are becoming larger and larger in terms of size, including obsolete code for functionalities that are no longer needed, or to account for the diversity of devices (and their OS versions). For example, in large companies, because of developer rotation, "dead" code/functionality may remain hidden in plain sight of app code without development teams risking to remove them. As a result, the effort thrown in maintaining those apps increases continuously, where consequently the testing efforts required to verify the functional correctness of those apps also boost. Therefore, to alleviate this problem, we argue that testing such apps clearly necessitates optimizing the selection of code that must be tested in priority. Test cases prioritization must then be performed in conjunction with a code optimization process to focus on actively used code w.r.t. user interactions to the app.

[17] https://savvyapps.com/blog/how-often-should-you-update-your-app

3.7 Threats to Validity

We have identified the following threats to validity in our study:

On potential misses of literature – We have not considered for our review books and Master or PhD dissertations related to the Android testing. This threat is mitigated by the fact that the content of such publications is eventually presented in peer-reviewed venues which we have considered. We have also considered only publications written in English. Nevertheless, while searching with the compiled English keywords, we have also found a few papers written in other languages, such as German and Chinese. The number of such non-English papers remain however significantly small, compared with the collected English literature, suggesting that our SLR is likely complete. Last but not the least, although we have refined our searching keywords several times, it is still possible that some synonyms are missed in this work. To mitigate this, we believe that natural language processing (NLP) could be leveraged to disclose such synonyms. We, therefore, consider it as our future work towards engineering sound keywords for supporting SLR.

On data extraction errors – Given that papers are often imprecise with information related the aspects that we have investigated, the extracted data may not have been equally reliable for all approaches, and data aggregation can still include several errors as warned by Turner et al. [226] for such studies. We have nevertheless strived to mitigate this issue by applying a cross-checking mechanism on the extracted results, following the suggestion of Brereton et al. [34]. To further alleviate this, we plan to validate our extracted results through their original authors.

On the representativeness of data sources and metrics – We have implemented the "major venues search" based on the venue ranking provided by the CCF. This ranking is not only potentially biased towards a specific community of researchers but may also change from one year to another. A replication of this study based on other rankings may lead to different primary publications set, although the overall findings will likely remain the same since most major venues continue to be so across years and across ranking systems.

The aspects and metrics investigated in this approach may also not be exhaustive or representative of everything that characterizes testing. Nevertheless, these metrics have been collected from testing literature to build the taxonomy and are essential for comparing approaches.

3.8 Related Work

Mobile operating systems, in particular, the open-source Android platform, have been fertile ground for research in software engineering and security. Several surveys and reviews have been performed on approaches for securing [222, 248], or statically analysing Android apps [139]. A systematic literature review is indeed important to analyse the contributions of a community to resolve the challenges of a specific topic. In the case of Android testing, such a review is missing.

Several works in the literature have however attempted to provide an overview of the field via surveys or general systematic mappings on mobile application testing techniques. For example, the systematic mapping of Sein et al. [262] addresses all together Android, iOS, Symbian, Silverlight and Windows. The authors have provided a higher-level categorization of techniques into five groups: 1) usability testing; 2) test automation; 3) context-awareness; 4) security and 5) general category. Méndez-Porras et al. [174] have provided another mapping, focusing on a more narrowed field, namely automated testing of mobile apps. They discuss two major challenges for automating the testing process of mobile apps, which are an appropriate set of test cases and an appropriate set of devices to perform the testing. Our work, with this SLR, goes in-depth to cover different technical aspects of the literature on specifically Android app testing (as well as test objectives, targets and publication venues).

Other related works have discussed directly the challenges of testing Android apps in general. For example, Amalfitano et al. [17] analyse specifically the challenges and open issues of testing Android apps, where they have summarized suitable and effective principles, guidelines, models, techniques and technologies related to testing Android apps. They enumerate existing tools and frameworks for automated testing of Android apps. They typically summarize the issues of software testing regarding non-functional requirements, including performance, stress, security, compatibility, usability, accessibility, etc.

Gao et al. [63] present a study on mobile testing-as-a-service (MTaaS), where they discuss the basic concepts of performing MTaaS. Besides, the motivations, distinct features, requirements, test environments and existing approaches are also discussed. Moreover, they have also discussed the current issues, needs and challenges of applying MTaaS in practice.

More recently, Starov et al. [214] performed a state-of-the-art survey to look into a set of cloud services for mobile testing. Based on their investigation, they divide the cloud services of mobile testing into three sub-categories: 1) Device clouds (mobile cloud platforms); 2) Services to support application lifecycle management and 3) Tools to provide processing according to some testing techniques. They also argue that it is essential to migrate the testing process to the clouds, which would make teamwork become possible. Besides, it can also reduce the testing time and development costs.

Muccini et al. [187] conducted a short study on the challenges and future research directions for testing mobile apps. Based on their study, they find that (1) Mobile apps are so different from traditional ones and thus they require different and specialized techniques in order to test them and (2) There seems to have many challenges. As an example, the performance, security, reliability and energy are strongly affected by the variability of the testing environment.

Janicki et al. [105] surveyed the obstacles and opportunities in deploying model-based GUI testing of mobile apps. Unlike conventional automatic test execution, model-based testing goes one step further by considering the automation of test generation phases as well. Based on their studies, they claim that the most valuable kind of research need (as future work) is to perform a comparative experiment on using conventional test and model-based automation, as well as exploratory and script-based manual testing to evaluate concurrently on the same system and thus to measure the success of those approaches.

Finally, the literature includes several surveys [170, 202, 222, 251] on Android, which cover some aspects of Android testing. As an example, Tam et al. [222] have studied the evolution of Android malware and Android analysis techniques, where various Android-based testing approaches such as A^3E have been discussed.

3.9 Summary

We report in this chapter on a systematic literature review performed on the topic of Android app testing. Our review has explored 103 papers that were published in major conferences, workshops and journals in software engineering, programming language, and security domain. We have then proposed a taxonomy of the related research exploring several dimensions including the objectives (i.e., what functional or non-functional concerns are addressed by the approaches) that were pursued, and the techniques (i.e., what type of testing methods – mutation, concolic, etc.) that were leveraged. We have further explored the assessments presented in the literature, highlighting the lack of established benchmarks to clearly monitor the progress made in the field. Finally, beyond quantitative summaries, we have provided a discussion on future challenges and proposed new research directions of Android testing research for further ensuring the quality of apps with regards to compatibility issues, vulnerability-inducing updates, etc.

Table 3.10: The Full List of Examined Publications.

Year	Venue	Title
2016	APSEC	Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising
2016	ISSRE	Experience Report: Detecting Poor-Responsive UI in Android Applications
2016	ASE	Generating test cases to expose concurrency bugs in android applications
2016	AST	Fuzzy and cross-app replay for smartphone apps
2016	ICST	Automatically Discovering, Reporting and Reproducing Android Application Crashes
2016	JCST	Prioritizing Test Cases for Memory Leaks in Android Applications
2016	SecureComm	Using Provenance Patterns to Vet Sensitive Behaviors in Android Apps
2016	ICSE	Reducing combinatorics in GUI testing of android applications
2016	FSE	Minimizing GUI event traces
2016	ESORICS	Mobile Application Impersonation Detection Using Dynamic User Interface Extraction
2016	AST	Automated test generation for detection of leaks in Android applications
2016	ISSTA	Energy-aware test-suite minimization for android apps
2016	ISSREW	Replaying Harmful Data Races in Android Apps
2016	FSE	DiagDroid: Android performance diagnosis via anatomizing asynchronous executions
2016	ISSTA	Automatically verifying and reproducing event-based races in Android apps
2016	ICSE	Mobiplay: A remote execution based record-and-replay tool for mobile applications
2016	ISSTA	Sapienz: multi-objective automated testing for Android applications
2016	FSE	Automated test input generation for Android: are we really there yet in an industrial case?
2016	APSEC	Testing Android Apps via Guided Gesture Event Generation
2016	AST	Graph-aided directed testing of Android applications for checking runtime privacy behaviours
2016	ASE	Automated model-based android gui testing using multi-level gui comparison criteria
2016	ICSE	Mining Sandboxes
2016	MOBILESoft	Reproducing context-sensitive crashes of mobile apps using crowd-sourced monitoring
2016	ISSTA	Monkey see, monkey do: effective generation of GUI tests with inferred macro events
2015	ISSTA	Systematic execution of Android test suites in adverse conditions
2015	ICST	Detecting Display Energy Hotspots in Android Apps
2015	OOPSLA	Scalable race detection for Android applications
2015	SEKE	Generating various contexts from permissions for testing Android applications
2015	ToR	To What Extent is Stress Testing of Android TV Applications Automated in Industrial Environments?
2015	SoMet	Towards Automated UI-Tests for Sensor-Based Mobile Applications
2015	ESORICS	Web-to-Application Injection Attacks on Android: Characterization and Detection

2015	ICIST	The Testing Method Based on Image Analysis for Automated Detection of UI Defects Intended for Mobile Applications
2015	MCS	Runtime Verification of Expected Energy Consumption in Smartphones
2015	ASEW	Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns
2015	ICSTW	Towards mutation analysis of Android apps
2015	SOSE	Testing Location-Based Function Services for Mobile Applications
2015	IS	MobiGUITAR: Automated Model-Based Testing of Mobile Apps
2015	DeMobile	AGRippin: a novel search based testing technique for Android applications
2015	ICSTW	Security testing for Android mHealth apps
2015	SOSE	Compatibility Testing Service for Mobile Applications
2015	ISSRE	SIG-Droid: Automated System Input Generation for Android Applications
2015	COMPSAC	A Context-Aware Approach for Dynamic GUI Testing of Android Applications
2015	S&P	Effective Real-Time Android Application Auditing
2015	ISSTA	Dynamic detection of inter-application communication vulnerabilities in Android
2015	ASE	The iMPAcT Tool: Testing UI Patterns on Mobile Applications
2015	SEKE	Test Model and Coverage Analysis for Location-based Mobile Services
2015	ENTCS	Evaluating the Model-Based Testing Approach in the Context of Mobile Applications
2014	SPSM	A5: Automated Analysis of Adversarial Android Applications
2014	WISE	Improving code coverage in android apps testing by exploiting patterns and automatic test case generation
2014	MobiCom	Caiipa: Automated Large-scale Mobil App Testing through Contextual Fuzzing
2014	SAC	A model-based approach to test automation for context-aware mobile applications
2014	PLDI	Race detection for event-driven mobile applications
2014	AsiaCCS	IntentFuzzer: detecting capability leaks of android applications
2014	AST	An automated testing approach for inter-application security in Android
2014	ICSTW	Using Combinatorial Approaches for Testing Mobile Applications
2014	MoMM	Multivariate Testing of Native Mobile Applications
2014	STTT	APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities
2014	FSE	Detecting energy bugs and hotspots in mobile apps
2014	ICSE	Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain
2014	SEKE	Towards Automatic Consistency Checking between Web Application and its Mobile Application
2014	MobileCloud	AppACTS: Mobile App Automated Compatibility Testing Service
2014	ESORICS	Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models
2014	MSR	Mining energy-greedy API usage patterns in Android apps: an empirical study
2014	QUATIC	Pattern Based GUI Testing for Mobile Applications
2014	FSE	EvoDroid: segmented evolutionary testing of Android apps
2014	ICST	Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps
2014	TrustCom	Attack Tree Based Android Malware Detection with Hybrid Analysis

2014	SERE-C	CRAXDroid: Automatic Android System Testing by Selective Symbolic Execution
2014	HASE	Testing of Memory Leak in Android Applications
2014	WODA/PERTEA	Intent fuzzer: crafting intents of death
2014	PLDI	Race Detection for Android Applications
2014	APSEC	User Guided Automation for Testing Mobile Apps
2014	TSE	On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices
2014	ICODSE	A/B test tools of native mobile application
2013	FASE	A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications
2013	IJIS	DroidTest: Testing Android Applications for Leakage of Private Information
2013	ISSRE	Systematic testing for resource leaks in Android applications
2013	MOBIQUITOUS	Appstrument - A Unified App Instrumentation and Automated Playback Framework for Testing Mobile Applications
2013	IS	Improving the Accuracy of Automated GUI Testing for Embedded Systems
2013	OOPSLA	Targeted and depth-first exploration for systematic testing of android apps
2013	MOBS	Testing for poor responsiveness in android applications
2013	ESEC/FSE	Dynodroid: An Input Generation System for Android Apps
2013	ICST	GUIdiff - A Regression Testing Tool for Graphical User Interfaces
2013	AST	Security testing of the communication among Android applications
2013	ICSTW	Considering Context Events in Event-Based Testing of Mobile Applications
2013	SCN	Context data distribution with quality guarantees for Android-based mobile systems
2013	ICSE	Reran: Timing-and touch-sensitive record and replay for android
2013	OOPSLA	Guided GUI testing of android apps with minimal restart and approximate learning
2013	ISSTA	Automated testing with targeted event sequence generation
2012	SEN	Verifying android applications using Java PathFinder
2012	FSE	Automated concolic testing of smartphone apps
2012	ASEA/DRBC	Hybrid Mobile Testing Model
2012	AST	A whitebox approach for automated security testing of Android applications on the cloud
2012	SEN	Testing android apps through symbolic execution
2012	ICST	Testing Conformance of Life Cycle Dependent Properties of Mobile Applications
2012	SPSM	SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications
2012	SRII	An Innovative System for Remote and Automated Testing of Mobile Phone Applications
2012	DSN	An empirical study of the robustness of Inter-component Communication in Android
2011	ICSTW	A GUI Crawling-Based Technique for Android Mobile Application Testing
2011	AST	Automating GUI testing for Android applications
2011	ICST	Experiences of System-Level Model-Based GUI Testing of an Android Application
2011	ICSECS	Towards Unit Testing of User Interface Code for Android Mobile Applications
2010	ICCET	Adaptive random testing of mobile application

4 Anchor: Locating Android Framework-specific Crashing Faults

We select Android app testing tools based on criteria established in Chapter 3 and build a dataset named ReCBench. ReCBench consists of framework-specific crashes of up-to-date Android app releases. Our empirical study on 500 crash cases of ReCBench has revealed that 37 percent of the crash types are related to bugs that are outside the stack traces. In practice, locating such bugs is challenged by the callback-based event-driven mechanism of Android. Furthermore, such bugs may even hide in extra-code artifacts of Android apk file, causing it more difficult to locate them. In this chapter, we propose Anchor*, a two-phase suspicious bug location suggestion tool.* Anchor *can find crash-inducing bugs not only inside, but even outside the stack trace.* Anchor *is lightweight and source code independent since it only requires the crash message and the apk file to locate the fault. Experimental results, collected via cross-validation on ReCBench and in-the-wild dataset evaluation, show that* Anchor *is effective in locating Android framework-specific crashing faults.*

This chapter is based on the work published in the following research paper:

- Pingfan Kong, Li Li, Jun Gao, Timothée Riom, Yanjie Zhao, Tegawendé F Bissyandé, and Jacques Klein. Anchor: Locating android framework-specific crashing faults. *arXiv preprint arXiv:2008.01676*, 2020

Contents

4.1 Overview **51**
4.2 Background **52**
4.2.1 Android App Crash Stack Trace 52
4.2.2 Callback-based and Event-driven Mechanism 53
4.2.3 Android APK File Format 54
4.3 Motivating Example **54**
4.4 Empirical Study on Fault Locations **56**
4.4.1 Dataset Construction 56
4.4.2 Ground Truth & Results 57
4.4.3 Category A: in Stack Trace 58
4.4.4 Category B: out of Stack Trace, in the Code 58
4.4.5 Category C: out of Stack Trace, out of Code 59
4.5 Ranking Suspicious Locations **60**
4.5.1 Phase 1: Categorization 60
4.5.2 Phase 2: Localization 60
4.6 Study Setup **63**
4.6.1 Research questions 63
4.6.2 Metrics 63
4.6.3 Cross-validation 64
4.6.4 Feature Selection 64
4.7 Study Results **64**
4.7.1 RQ1: Effectiveness of Categorization 64
4.7.2 RQ2: Effectiveness of Localization 65
4.7.3 RQ3: Overall Performance of ANCHOR 65
4.7.4 RQ4: Performance in the Wild 65
4.8 Disscussion **66**
4.8.1 Comparing ANCHOR with other Locators 66
4.8.2 Developer Effort for Locating Bugs 67
4.9 Threats to Validity **67**
4.9.1 Internal Threats 67
4.9.2 External Threats 67
4.10 Related Work **67**
4.11 Summary **68**

4.1 Overview

App crashes are a recurrent phenomenon in the Android ecosystem [240]. They generally cause damages to the app reputation and beyond that to the provider's brand [83]. Apps with too many crashes can even be simply uninstalled by annoyed users. They could also receive bad reviews which limit their adoption by new users. Too many apps crashes could also be detrimental to specific app markets that do not provide mechanisms to filter out low-quality apps concerning proneness to crash. The challenges of addressing Android app crashes have attracted attention in the research community.

Fan et al. [56] have recently presented insights on their large-scale study on framework-specific exceptions raised by open source apps. Tan et al. [223] further presented an approach to repair Android crashing apps. A common trait of all these crash-related studies is that the underlying approaches heavily rely on the generated stack traces to identify the fault locations. Although the state of the art is effective for many bugs, they are generally tailored to the generic cases where the stack traces provide relevant information for locating the bug. Unfortunately, there is a fair share of faults whose root causes may remain invisible outside the stack trace. Wu et al. [244] have already reported this issue in their tentative to locate crashing faults for general-purpose software. In the realm of Android, the phenomenon where the stack trace may be irrelevant for fault localization is exacerbated by two specificities of Android:

The Android system is supported by a callback-based and event-driven mechanism: Each component in Android has its lifecycle and is managed by a set of callbacks. Every callback serves as a standalone entry point and root to a separate call graph. Yet, existing crash-inducing bug localization techniques for Java such as CrashLocator [244] assume a single entry point to compute certain metrics for the suspiciousness score of different methods. Additionally, since the Android system is event-driven, the invocation sequence to functions and callbacks is affected by non-deterministic user inputs or system events, making the stack trace unreliable for quick analyses.

The Android app package includes both code and resources that together form the program: Android apps are more than just code. They are combinations of Java/Kotlin code, XML files, and resources (such as images and databases). Apps provide extensions to the Android Operating System (OS), which directly analyses XML files from the app to map callback functions, which the OS must trigger to exploit functionalities in the apps. Therefore, an error by developers within an XML document can eventually lead to a runtime crash. Similarly, it is important to note that crashes can occur due to other concerns such as the arrangements of app resources, use of deprecated APIs, omissions in permission requests, etc. Typical such errors, which occur outside of code pointed to by stack traces, will cause either developers or Automatic Program Repair (APR) tools (e.g., [223]) to pointlessly devote time in attempting to fix the code.

This chapter. Our work aims at informing the research community on the acute challenges of debugging framework-specific crashes. To that end, we propose to perform an empirical study that investigates the share of crashes that cannot be located by current localization approaches. Following this study, we present a new approach to locate faults, aiming at covering different categories of root cause locations. Overall, we make the following contributions:

- We present the results of an empirical study performed on a set of 500 app crashes retrieved from the ReCBench dataset (c.f Chapter 5). A key finding in this study is that we were able to identify that 37% crash root causes are associated with crash cases where the stack trace is not directly relevant for fault localization.
- We propose Anchor, a tool-supported approach for locating crashing faults. Anchor unfolds in two phases and eventually yields a ranked list of location candidates. The first phase applies a classification algorithm to categorize each new crash into a specific category. Depending on this category, a dedicated localization algorithm is developed in the second phase. Anchor

currently implements 3 localization algorithms that eventually generate a ranked list of buggy methods (when the bug is in the code) or resource types (when it is outside of code).
- We performed 5-fold cross-validation on the 500 crash cases to assess the effectiveness of ANCHOR in placing the crashing fault location in the top of its ranked list of suggestions. ANCHOR exhibited an overall MRR (Mean Reciprocal Rank) metric value of 0.85. An analysis of the open dataset of crashed open-source Android apps further shows that our method scales to new app crashes.

The rest of this chapter is organized as follows. Section 4.2 introduces background details on Android app crashes and callback-based event-driven mechanisms. Section 4.3 revisits the motivating example by the previous work [223] and demonstrates why research in crash localization has standing challenges. Section 4.4 discusses the findings of our empirical study and explores the insights that can be leveraged for a new approach. Section 4.5 presents ANCHOR. We describe experimental setup in Section 4.6 and approach evaluation in Section 4.7. We bring further discussion in Section 4.8. Threats to validity are acknowledged in Section 4.9 and related work is presented in Section 4.10. Finally, Section 4.11 concludes the chapter.

4.2 Background

In this section, we introduce the important concepts related to this chapter.

4.2.1 Android App Crash Stack Trace

Like all Java[1] based software, when Android apps crash, they can dump execution traces which include the exception being thrown, a crash message, and most importantly, a stack trace of a callee-caller chain starting from the *Signaler*, i.e., the method that initially constructed and threw the exception object. Figure 4.1 is an example of stack trace for the crash of the app *Sailer's Log Book*. This app helps sailors to keep their logbook accurate and up-to-date. On the first line, the exception *IllegalArgumentException* is thrown. On the second line, the log system reports message "recursive entry to executePendingTransactions". Starting from the third line, the *Signaler* of the stack trace is listed: it is this framework method that instantiates the exception type, composes the log message and throws it to its caller to handle. On Lines 4-5 that are also marked in grey, there are other two methods that continue to pass on the exception. Line 5 holds the *API*, which is the only framework method in this stack trace that is visible to the developer. Since the crash happens directly due to invocation to it, we call it the *Crash API*. Line 6 is the developer method that invoked this API. Line 7 is the developer implementation of the callback, inherited from the superclass of the Android framework. Android framework decides, based on certain conditions and system/user events, when to invoke this method, and what parameter values to pass in. Lines 8-9 are part of the Android framework core that is, again, not accessible to developers.

The crash stack trace is often the first thing that developers want to examine when a crash is reported [115]. Even when it is not given, developers would reproduce and retrieve them. Intuitively, the crash arises from mistakes in the developer methods, e.g., Lines 6-7 in Figure 4.1. Particularly, the *Crash method* that directly invoked the *Crash API*. Our empirical study in Section 4.4 shows that this intuition is correct, that 63% of the total crash types are in the stack trace. However, in the rest of this section, we will introduce the specialty of Android that may lead to the rest 37%.

[1]Kotlin has also been widely used in recent years as an alternative for Android app development, it is designed to fully interoperate with Java.

Exception	IllegalArgumentException		1
Message	Recursive entry to executePendingTransactions.		2
Framework	android.app.FragmentManagerImpl.executePendingActions	Signaler	3
	android.app.FragmentManagerImpl.executePendingTransactions		4
	android.app.FragmentManagerImpl.popBackStackImmediate	Crash API	5
Developer	com.poterion.logbook.activities.MainActivity.show	Crash method	6
	com.poterion.logbook.fragments.SettingsFragment. onSharedPreferenceChanged	Callback	7
	:		
Framework	com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run	Core	8
	com.android.internal.os.ZygoteInit.main		9

Figure 4.1: Crash Stack Trace of app Sailer's Log Book.

4.2.2 Callback-based and Event-driven Mechanism

Unlike traditional Java programs, Android apps have multiple entry points. Each entry point is a callback method (e.g., Line 7 in Figure 4.1), which is declared in one of the Android framework component classes, inherited by the developer defined subclass, and maybe overridden by the developer. The Android framework core, based on the user inputs and system environments, decides when to invoke the callbacks and what parameter values to pass in. Each callback is standalone, and in general Android does not encourage developers to invoke those callbacks from their self-defined methods, unless these methods are callbacks overriding and invoking their super. As a result, existing static call graph based fault localization techniques [244] for Java programs can not be simply reused, since they assume single entry points and need to compute weighing scores based on the graph. There are, however, works [192, 255] that have invented methods to track the control flows or data flows and tried to build the callback connections. These proposed approaches are either computationally expensive or confined in limited number of component classes, and does not scale to all scenarios. Other approaches like [138] or [23] create a dummy main to invoke all callbacks in order to reuse Java based analysis tools, but this method discarded the relation among callbacks, which is crucial to estimate the possibility of a method containing the real bug.

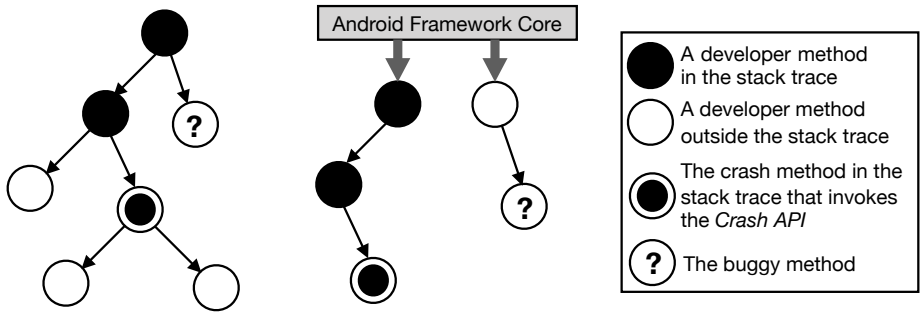


Figure 4.2: Call Graph Comparison between General Java Program (left) and Android App (right), inspired from [244]

Figure 4.2 examplifies the difference of call graphs between general Java program (left) and Android app (right). The filled circles represent the developer methods in the stack trace, while the non-filled circles represent developer methods outside the stack trace. The partially filled circles represent the *Crash method* that invokes the *Crash API*. Generally, the buggy method is the *Crash method*. However, as shown in our empirical study, it appears that the buggy method (the circle filled with question mark in Figure 4.2) is not connected to the *Crash method*. A traditional Java program static call graph based approach such as CrashLocator [244] will be able to locate this buggy method only

if the buggy method is "close enough" to the generated call graph (roughly speaking they generate an extended call graph leveraging the stack trace). However, on the right, in the case of Android apps, the buggy method could be in a separate call graph because of callback methods that are invoked by the Android framework. Such cases will be missed by approaches such as CrashLocator [244] that only detects buggy methods captured by its extended call graph, but does not consider callback methods.

4.2.3 Android APK File Format

Android apps are distributed in a package file format with extension ".apk". It is a compressed folder containing code, resources, assets, certificates, and manifest file. All of these files are crucial to the expected good functioning of the apps. Therefore, some crashes may be induced when there are problems with these files.

4.2.3.1 Android Manifest File

Every app project needs to have an AndroidManifest.xml file at the root of the project source set [161]. Along with the package name and components of the app, this file also declares the permissions that the apps needs, as well as the hardware and software features that the app requires.

4.2.3.2 Android Component Layout Description File

Android component layout description files are also crucial to the execution of the app. E.g., Listing 4.1 is the layout file of the main Activity of an Android app *Transistor*. In this file, a child fragment is defined and described. The attribute *android:id* defines the layout file name to be inflated for the fragment, the attribute *android:name* gives the full name of the user defined Fragment class. When the main Activity is being created, the Android framework scans this layout file, and invokes a series of relevant callbacks on this Fragment to draw it along with the main Activity.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <fragment xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     android:id="@+id/fragment_main"
5     android:name="org.y20k.transistor.MainActivityFragment"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     tools:layout="@layout/fragment_main" />
```

Listing 4.1: Main Activity Layout File of app Transistor.

4.3 Motivating Example

We further illustrate the challenges of locating faults outside Android app stack traces by revisiting an example that was used to motivate a previous work on Android app crash automatic repairing by Tan et al. [223]. *Transistor*[2] is a popular online radio streaming app. We showed its partial resources in Section 4.2.3.2. However, it was reported that following the input sequence in Figure 4.3, the app will crash.

[2]https://github.com/y20k/transistor/issues/21

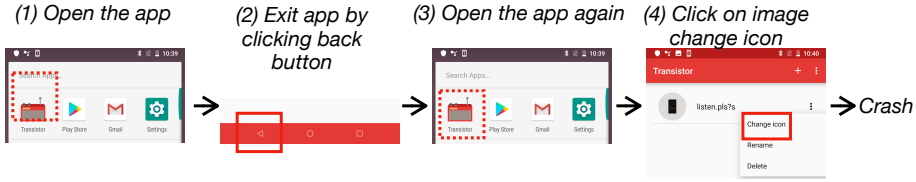


Figure 4.3: Crash of Transistor.

```
1 java.lang.IllegalStateException:
2 MainActivityFragment{e7db358} not attached to Activity
3 at ...MainActivityFragment.startActivityForResult(Fragment.java:925) (Crash API)
4 at ...agment.selectFromImagePicker(MainActivityFragment.java:482) (Crash method)
5 at ...k.transistor.MainActivityFragment.access$500(MainActivityFragment.java:58)
6 at ...transistor.MainActivityFragment$6.onReceive(MainActivityFragment.java:415)
```

Listing 4.2: Crash Message of *Transistor*.

The crash message filtered out from logcat is shown in Listing 5.1. It appears that invoking the *startActivityForResult API* on the `MainActivityFragment` (line 3) throws an unhandled *IllegalStateException* (line 1), and the Android system reports that the fragment is not attached to the hosting activity (line 2). By inspecting the source code of Android framework of the *Crashed API* (line 3), we see that the *startActivityForResult* method of the fragment instance attempts to invoke its context's (i.e., its host `Activity`'s) *API* with the same name *startActivityForResult*. This invocation is guarded by an if-clause, which checks whether the fragment is still attached to the host `Activity`. If not, however, the *IllegalStateException* will be thrown.

```
1 new BroadcastReceiver(){
2    onReceive(...){   ...
3 +     if(getActivity()!=null)
4       startActivityForResult(pickImageIntent,REQUEST_LOAD_IMAGE);}}
```

Listing 4.3: Fix from Tan et al.

Biased by the assumption that the fault should only be in the developer methods in the stack trace (lines 4-6), Tan et al. [223] proposed to amend the *Crash method* (line 4). Listing 4.3 shows their fix. Their fix applies a checker on invocation to *startActivityForResult*, which will not be executed if value of *getActivity* is null (i.e., when the fragment is no longer attached to its hosting `Activity`). As a result, the app avoids crashing. This fix indeed prevents the exception. However, it is not explainable: applying the checker not only prevents the crash, but it should also prevent opening the *SelectImageActivity* as designed for. Due to this paradox, we have a good reason to suspect that the true bug location is still hidden.

Transistor's developer, who is also dedicated in debugging in the stack trace, proposed a fix on her/his own in Listing 4.4. Realizing that the `Fragment` lost its reference to the host `Activity`. The developer declared a variable `mActivity` to hold the reference. Then in the *Crash method* (line 4 in Listing 5.1), she/he switched the invocation of the *startActivityForResult API* from `Fragment` to `mActivity`.

```
1 +  mActivity = getActivity(); ...
2 new BroadcastReceiver(){
3    onReceive(...){   ...
4 -     startActivityForResult(pickImageIntent,REQUEST_LOAD_IMAGE);
5 +     mActivity.startActivityForResult(pickImageIntent,REQUEST_LOAD_IMAGE);}}
```

Listing 4.4: Fix from Developer.

This fix also bypassed the crash, but it causes regression. After the final step in Figure 4.3, if the user clicks on the back button two more times, the app should have first returned to the `MainActivity`,

then back to the home screen. Instead, it opens another *SelectImageActivity*. In the issue tracking, the developer admits that she/he had no idea of how to fix it. While after several months, the bug "fixed" itself, which she/he described as "scary". Even Tan et al. failed to explain the cause of this regression.

Based on the understanding of Android' callback-based mechanism introduced in Section 4.2.2, we suspect that the bug may not exist in the stack trace. We confirmed our fix shown in Listing 4.5. This fix is reported to the developer and we received positive feedback in the issue tracking, as can be verified in *Transistor*'s respository given above.

```
MainActivityFragment extends Fragment{
   onDestroy(){
+     super.onDestroy();
+     LocalBroadcastManager.getInstance(mApplication).unregisterReceiver(
     imageChangeRequestReceiver,imageChangeRequesIntentFilter);}}
```

Listing 4.5: Fix Inspired by this Chapter.

We broaden the search for the bug outside the stack trace. Noticing the crash originated from the *onReceive* callback (cf. line 6 in Listing 5.1), we examine the lifecycle of this `BroadcastReceiver` object. We found that it is registered in the *onCreate* callback of `MainActivityFragment`, but never unregistered in its counterpart callback *onDestroy*. As a result, after Step 2 (cf. Figure 4.3), the registered `BroadcastReceiver` and its host `MainActivityFragment` are leaked in the memory. In Step 4, the callbacks of the leaked objects are stealthily invoked by the Android framework and eventually caused the *IllegalStateException*. Knowing the true cause of the crash, it is not difficult to explain the paradox of Tan et al.'s fix and the regression caused by the developer's fix. However, given the page limit, we put detailed reasoning online at https://anchor-locator.github.io.

Hint: The fault locations in Android apps may: (1) Be outside the stack trace; (2) Be even outside the call graph extended from the stack trace; (3) Not even "exist" in the code, i.e., they are inherited methods without visible code snippets. Locating such faults may require tremendous efforts. Fixes based on incorrect localization may even cause regression.

4.4 Empirical Study on Fault Locations

In this section, we present the results of an empirical study that we performed on a set of 500 app crashes retrieved from the ReCBench dataset constructed by us. The details of this dataset are described in Chapter 5. This study aims at assessing to what extent the locations of crashing faults reside outside the stack trace.

4.4.1 Dataset Construction

We extract our dataset from ReCBench, an open dataset proposed by us in Chapter 5. ReCBench has been built by running hundreds of thousands of Android apps downloaded from various well-known Android markets [11, 143]. In addition to a collection of crashed Android apps focusing on framework-specific crashes[3], ReCBench offers the possibility to collect crash log messages and scripts to reproduce the crashes. Today, ReCBench contains more than 1000 crashed apps (still growing). For our empirical study, we focus on crashed apps for which:

[3]Android framework methods are not visible or understandable to general developers, hence greater challenge is acknowledged for locating framework-specific crashes compared to developer-written methods. [56]

- First, the stack trace of the crash contains at least one developer method. This is a requirement to be able to start an exploration process to find the crash root cause.
- Second, since we specifically target the crashes induced by Android *API*s, the *Signaler* must be Android-specific.

After applying these two rules, we randomly selected 500 crashed apps from the remaining apps. The dataset is publicly accessible at:
https://github.com/anchor-locator/anchor.

4.4.2 Ground Truth & Results

We manually inspect all the 500 crashed apps to understand the reason behind the crashes and to create our ground truth. We perform this manual inspection by leveraging the CodeInspect [61] tool, following same protocols discussed in [56]. Each of the crashed apps has been categorized into one of the following categories:

- Category A groups the crashed apps for which the buggy method (i.e., the root cause) is one of the developer methods present in the stack trace;
- Category B groups the crashed apps for which the buggy method is not present in the stack trace, but still in the code.
- Category C groups the crashed apps for which the crash arises from non-code reasons.

The above partition is one out of many alternatives, e.g., one can also separate bugs based on whether they are concurrent [33,145,166,224,235]. We later show in Section 4.5.2 how this partition helps with building our localization tool. Table 4.1 summarizes the outcome of the empirical study. It appears that for 89 (49+40) crashed apps (representing 18% of the total cases), the crashing fault location is not in any of the developer methods present in the stack trace. The respective numbers of Categories B and C are close, with 49 cases in Category B and 40 cases in Category C. To further investigate how many types of crashes occur for each category, we group crashes from a given category into *buckets*. Specifically, if two crash cases have identical framework crash sub-trace, they will be put into the same bucket. The last two columns in Table 4.1 present the number of buckets per category. Overall, there are 105 types of crashes (i.e., buckets) in the dataset. The percentage of types of crashes in Categories B and C are 16% and 21%, respectively. In total, there are 37% of buckets whose buggy reasons are not shown in the stack traces. Each unique framework crash sub-trace suggests a unique type of crash-inducing bug. Therefore, considering crash types encountered per the same number of cases (buckets#/case#) in each category, more debugging effort will be needed for Categories B and C than in Category A.

Table 4.1: Categories of Fault Locations in Android apps

Category	stack trace	code	case#	percent	bucket#	percent
A	in	yes	411	82%	66	63%
B	out	yes	49	**10%**	17	**16%**
C	out	no	40	**8%**	22	**21%**
Total	-	-	500	100%	105	100%

Hint: 18% of the crashes are due to bugs for which the location is outside the stack trace. A significant number of root causes (buckets), i.e., 37% (16%+21%), are associated with cases where the stack trace is not directly relevant for localization. In even 21% of the cases, the root causes are not located in the code.

We now detail each category in the rest of this Section.

4.4.3 Category A: in Stack Trace

Category A includes all crash cases whose bugs reside in one of the developer methods present in the stack trace. Most crashes in our dataset fall into this category. It is expected that by default, developers start their debugging process with the methods present in the stack trace [102,109,206,209]. The automatic crash-inducing bug repairing tool named Droix [223] implements a locater, by assuming that the *Crash method* is the bug location in all scenarios. However, we also notice that the true crashing fault may reside in other developer methods, in particular when moving downward in the stack trace. An example of such a case is when the caller methods pass an incorrect instance to the crashed developer methods. Generally, much less effort is needed in locating faults in this category. Since the number of suspected methods is limited and their names are already known. Therefore they are not the focus of this chapter.

4.4.4 Category B: out of Stack Trace, in the Code

It has drawn attention to researchers that Java program crashes can be caused by methods that are not listed in stack traces. Approaches like CrashLocator [244] broadens the search for such faulty methods in extended call graphs from stack traces. We demonstrate in the rest of this section why this broadened search is not enough for Android apps. There are in total 49 cases in this category, each crashed from wrongly handling a framework *API*. Based on the type of the framework *API* (call-in or callback), we further categorize them into two sub-categories: (1) Misused Call-In APIs and (2) Non-Overridden Callback APIs.

4.4.4.1 Type 1: Misused Call-In APIs (44 cases out of 49)

This first type groups crashing faults caused by the misuse of call-in APIs. This means that the bug leading to a crash is due to a buggy explicit invocation of an API from a developer method. Moreover, this invocation is often performed from another implemented callback, other than the callback in the stack trace. Since both callback methods are triggered by the framework, it is unlikely that an extended call graph can cover such methods (cf. Figure 4.2).

```
1 public class MainActivity extends Activity{
2    onCreate(...){
3    try{bindService(intent,serviceConnection,integer);/*Bug Location*/
4    }...}...
5    onDestroy(){unbindService(serviceConnection);/*Crash location*/}}
```

Listing 4.6: Bug Explanation to app Geography Learning.

Listing 4.6 gives a concrete example. This example is extracted from an app named *Geography Learning* which helps users to remember geography knowledge in a quiz game format. When the *MainActivity*[4] of this app is launched, the callback method `onCreate` is automatically triggered by the Android framework. Then, this `onCreate` method invokes the `bindService` API to bind to *Service*. *Service* is one of the four basic components of Android, and wrongly handling of *Service* is not uncommon [212] in Android app development. When the user exits the *MainActivity*, the Android Framework invokes the *onDestroy* callback method and tries to unbind the *Service* bound in the `onCreate` method. Thereafter, the app crashes with the exception type *IllegalArgumentException*. Analysing the message which says: *"Service not registered: com.yamlearning. geographylearning.e.a.e@29f6021"*, we understand that the *Service* has not been bound. In the method body of the overridden *onCreate* callback, we found that the invocation to API *bindService* was misused. Indeed, *bindService* is surrounded by a try-catch clause, and another statement preceding this API invocation threw an exception which redirects the execution flow to the catch block, skipping the invocation to *bindService*.

[4]The *Main Activity* of an app is the first screen shown to the user when launched.

Out of a total of 49 cases in Category B, 44 falls into this sub-category.

4.4.4.2 Type 2: Non-Overridden Callback APIs (5 cases out of 49)

This second type includes crashes caused by the non-overridden callback APIs. Callbacks, or call-afters, are APIs that are invoked when the Android framework decides so, based on certain system environment change and/or user actions. Callbacks are inherited, when developers define classes that are subclassing Android base component classes. Developers are often required to override certain, although not all, callback APIs. Forgetting to handle these callbacks may cause apps to crash immediately. Moreover, these crashes may often seem flaky, since its reproduction requires re-establishing the same system environments and/or repeating user action sequences. Existing Java crash locators fail to spot such bugs with two reasons: (1) These callback APIs are not in the extended call graphs of stack traces; (2) The method snippets in developer-defined codes do not exist, so are easily missed.

Listing 4.7 shows an example of this crash type. The app *Fengshui Master* is a Chinese fortune teller app. The app crashes when trying to get a reference to the writable database. However, when the app crashes, the exception *SQLiteDatabaseException* is triggered with a message claiming *"not able to downgrade database from version 19 to 17"*.

```
1 public class com.divination1518.f.s{
2    a(..){sqliteOpenHelper.getWritableDatabase();/*Crash location*/}}
3 public class com.divination1518.g.p extends SQLiteOpenHelper{ ...
4 + onDowngrade(..){...}/*Bug Location*/}
```

Listing 4.7: Bug Explanation to Android app Fenshui Master.

According to the Android documentation, the app developer needs to implement the callback method *onDowngrade* in the self-defined subclass of *SQLiteOpenHelper*. This callback method will be invoked when the database in storage has a higher version than that of the system distribution. Failing to override this callback API immediately crashes the app. Note that the motivating example (cf. Section 4.3) also falls into this sub-category. Given the stealthiness of this fault type, it is particularly difficult, even for a human developer, to spot the bug reason without being very familiar with the Android official documentation. Out of a total of 49 cases in Category B, 5 falls into this sub-category.

Note that we use api_h to denote the wrongly handled API (call-in API or callback API) for cases of Category B. This denotation is later needed for Section 4.5.2.2.

4.4.5 Category C: out of Stack Trace, out of Code

As introduced in Section 4.2.3, except code, an Android apk also contains resources, assets, certificate, and manifest. They are critical to the functioning of the app. As a result, mistakes in those files may also cause crashes. Table 4.2 gives a summary of the buggy locations outside of code. As illustrated, eleven cases of crashes originate from the *Manifest.xml* file. Most cases in this type are because the permissions are not properly declared in the manifest. Resources include specifically files with ".xml" extension (excluding the *Manifest.xml* file). An Android app uses these resource files to store the layout and pieces of information like string values. If the required resource is missing or wrong, then the app will crash. Assets are the large files, like fonts, images, bitmaps. Assets should be put in the correct directory. If the Android framework is not able to find them and it will crash the app.

Aside from the files inside the apk, some constraints put forward by the device's hardware and firmware, i.e., the framework may also cause the app to crash. For example, the Android billing service can only be tested on real devices, if, however, tested on emulators, the app crashes [162]. Also, since Android is quickly updated with new features and designs, old apps may crash on newly

Table 4.2: Crash Causes of Categorie C

Sub-Category	Manifest	Hardware	Asset	Resource	Firmware
Cases	11	5	4	2	18

distributed devices, due to reasons like deprecated APIs and new security protocols established by the framework. Developers should generally redesign the relevant functionalities, therefore no single buggy location can be decided.

4.5 Ranking Suspicious Locations

To help developers identify the true fault locations when debugging app crashes, including faults that reside outside the stack traces, we propose ANCHOR. ANCHOR is a fault location recommendation system based on a two-phase approach. In the first phase, ANCHOR categorizes a given crash in one of the three categories (A, B, or C) with a classification system. Then, in the second phase, according to the decided category, ANCHOR each adopts a unique algorithm to suggest a rank of locations that are suspected to contain the true faults. The rest of this section describes Phase 1 and Phase 2 in more detail.

4.5.1 Phase 1: Categorization

The first phase aims at assigning a new crash to one of the three categories (A, B, or C). We use the Naïve Bayes algorithm [200] for the categorization. Naïve Bayes is one of the most effective and competitive algorithms in text-based classification. It is widely used for spam detection [176, 258], weather forecasting [232], etc. It is especially suitable in the scenario when the training set does not contain a large number of records [101], e.g., our empirical dataset contains merely 500 manually constructed records.

To construct a vector for each crash record, we feature words extracted from the crash message. The value of each feature dimension is binary, indicating whether a word exists or not in the message. More specifically, we extract three parts from the crash message: (1) The exception type, which is a Java class (e.g., `IllegalArgumentException`); (2) The exception message, which briefly describes the reason of the crash, e.g., line 2 in Figure 4.1; (3) The top framework stack frames, each being a Java method, e.g., lines 3-5 in Figure 4.1. For (1) and (3), we use "." as the word separator, for (2), we use space as the separator. To avoid overfitting and to save computing resources, we do not need the entirety of the vocabulary to build the vector. In Section 4.6.4, we further discuss how many words are necessary.

With this categorization system, each new crash will firstly be categorized as a type of "A", "B" or "C" before being processed in Phase 2.

4.5.2 Phase 2: Localization

The goal of this phase is to provide a rank of potential bug locations (in descending order of suspiciousness), in the form of developer methods when the bug is in the code (i.e., Categories A and B) and of sub-categories when the bug is not in the code (i.e., Category C). Before presenting in the following sub-sections 3 standalone algorithms, one for each category, we explain how we compute a similarity score between two crashes. This similarity score is used in both Categories B and C localization algorithms.

Similarity between two Crashes: We quantify the similarity between two crashes C_1 and C_2 by computing the similarity between their crash messages as presented in Equation 4.1:

$$Sim_{C_1,C_2} = Edit_Sim(seq_{C_1}, seq_{C_2}) \tag{4.1}$$

seq is the sequence of framework stack frames in a crash message, e.g., lines 3-5 in Figure 4.1. Sim_{C_1,C_2} is then computed as the *Edit Similarity* [197] between seq_{C_1} and seq_{C_2}. The intuition here is that when two crashes share similar bug reasons, their *seq* tends to be similar, if not identical.

4.5.2.1 Category A: In Stack Trace

Since the crash is assigned to Category A, it indicates that the buggy method is one of the developer methods in the stack trace. We inherit the intuition from [223], that if the developer method is closer to the *Crash API* in the stack trace, there is a higher chance that it contains the true fault. Therefore, we can obtain the rank without changing the order of the developer methods in the stack trace. For example, in Figure 4.1, methods on line 6 and line 7 are respectively ranked first and second.

4.5.2.2 Category B: Out of Stack Trace, in the Code

When the crash is classified into Category B, it indicates that the buggy developer method is not in the stack trace, but still in the code. As discussed in Section 4.4.4, the buggy method should either be a developer method that misused a call-in API, or a callback API that has not been overridden. In the remainder of this section, we will note api_h this API (call-in API or callback API) that has been wrongly handled (cf. Section 4.4.4). To infer a ranked list of potentially buggy methods, we propose Algorithm 1. The overall idea is, starting from each developer method in the stack trace, in addition to examining the developer methods (1) in the extend call graph, we also examine those that either (2) control the Android components' lifecycles, or (3) are involved in the data flow of the crash. The computation of the suspiciousness score follows the same intuition as explained in Section 4.5.2.1.

First of all, Algorithm 1 requires three input data: (1) *crash*, the crash under study; (2) *ST*, which is the list of developer methods contained in the stack trace, e.g. lines 6-7 in Figure 4.1; (3) api_h, the wrongly handled API, which is approximated as the associated wrongly handled API of the most similar crash present in Category B of our empirical dataset. More formally, let be $Crash_B$ the set of all the crashes in Category B. We identify the most similar crash $crash_{sim}$ by following Equation 4.2. Since their crash reasons are the most similar, it is with the highest possibility that both have wrongly handled the same API.

$$Sim_{crash,crash_{sim}} = max(Sim_{crash,crash_b}), crash_b \in Crash_B \tag{4.2}$$

The algorithm starts with retrieving a set of developer methods S from the entire apk that has invoked the api_h (line 1). The outmost for-loop (lines 2-19) loops over each stack frame sf in the stack trace ST. Then based on the type of the api_h, there are two sub-routines: (a) when api_h type is "call-in" (lines 4-11); (b) when api_h type is "callback" (lines 13-17). Next we discuss both sub-routines in detail.

Sub-routine for type "call-in" is a for-loop (lines 4-11) that loops over each method s in S. We then loop over (lines 5-9) all *Active Methods* (AM) declared in the same class as sf, where *Active Methods* are methods having actual code snippets in the Java class files, not including the inherited ones. The function *links* (line 6) checks 3 sub-conditions: (1) if s is invoked by am, or (2) if s and am are declared in the same Java class or (3) if an instance of the declaring class of s has been passed to am as a parameter. Sub-condition (1) checks if s is in the extended call graph of am, same as

Data: *crash*: the crash to resolve
Data: ST: List of developer methods in stack trace of *crash*
Data: api_h: *Wrongly handled API*
Result: R: Rank of suspicious developer methods

```
1:  S ← Developer methods that invoke api_h;
2:  for sf ∈ ST do
3:    if api_h type "call-in" then
4:      for s ∈ S do
5:        for am ∈ AM do
6:          if s links am then
7:            s.score+ = 1/d
8:          end if
9:        end for
10:     end for
11:     R ← S.sort()
12:   else if api_h type "callback" then
13:     for nc ∈ NC do
14:       if nc inherits api_h then
15:         R.put(nc)
16:       end if
17:     end for
18:   end if
19: end for
```

Algorithm 1: Localization Algorithm for Category B

locators like [244]. Sub-condition (2) implies that s is a callback method that involves controling the component lifecycle as am does. Sub-condition (3) implies potential data flow between s and am. When the condition holds true in line 6, a score is added for s (line 7). Here d is the distance between sf and *Crashed API* in the stack trace. It reflects on the same intuition in Section 4.5.2.1.

Sub-routine for type "callback" is implemented with a for-loop (lines 13-17) that loops over all the inherited Non-overridden Callback (NC) of the class where sf is declared. If nc inherits from api_h (line 14), it implies that overriding it may fix the problem, therefore nc will be added to the rank R (line 15). With the same intuition in Section 4.5.2.1, this sub-routine is designed so that when sf is closer to *Crashed API* in the stack trace, nc is in the higher location in the rank.

Algorithm 1 addresses the concerns in the empirical study (cf. Section 4.4.4). It can further locate faulty methods that are not in the extended call graphs, or even methods without actual code snippets.

4.5.2.3 Category C: Out of Stack Trace, out of Code

Figure 4.4 describes the localization process for crashes that have been classified into Category C. To infer a ranked list of potentially buggy locations, this process computes a suspiciousness score for each location. Since the true fault locations in Category C are not in the code, the locations in this ranked list are sub-categories (e.g. manifest, asset, etc.).

With the new crash, we start by computing the similarity score $Sim_{crash,crash_c}, crash_c \in Crash_C$. Here $Crash_C$ is the set of all the crashes of Category C in the empirical dataset. In Figure 4.4, the similarity scores are denoted as Sim_{caseID} for short. We then take an average of Sim_{caseID} over the same sub-categories. Sub-categories with higher similarity scores take higher positions in the *Rank*.

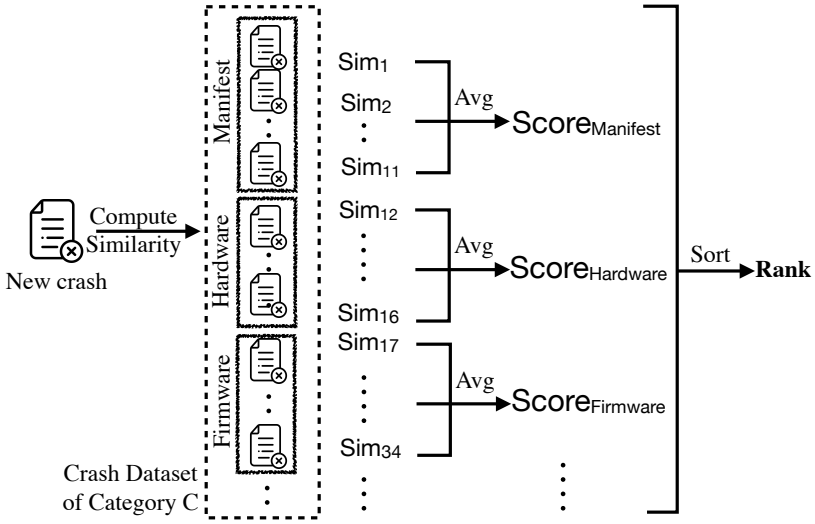


Figure 4.4: Localization Process for Category C.

4.6 Study Setup

This section clarifies the research questions, the metrics used to assess ANCHOR, and the parameter values involved.

4.6.1 Research questions

We empirically validate the performance of ANCHOR by investigating the following research questions:

- **RQ1:** To what extent is the categorization strategy effective?
- **RQ2:** To what extent are the localization algorithms reliable?
- **RQ3:** What is the overall performance of ANCHOR?
- **RQ4:** How does ANCHOR perform on crashes in the wild?

4.6.2 Metrics

Crash localization is a recommendation problem. To measure the performance of ANCHOR, we rely on rank-aware metrics, which are widely used in information retrieval communities and have been previously used to evaluate crash localization techniques [244].

Recall@k: The percentage of crash cases whose buggy functions appear in top k locations. A higher score indicates better performance of ANCHOR.

MRR (Mean Reciprocal Rank): The mean of the multiplicative inverse of the rank of the first correct location. As defined in Equation 4.3, $Rank_i$ is the rank for the ith crash case, in a set of crash cases E. A high value of MRR means developers on average need to examine fewer locations in the rank, and therefore, a better performance [208].

$$MRR = \frac{1}{|E|} \sum_{i=1}^{|E|} \frac{1}{Rank_i} \tag{4.3}$$

4.6.3 Cross-validation

We perform 5-fold cross-validation over the empirical dataset of 500 sample crashes. The dataset is randomly divided into 5 subsets of 100 sample crashes: 5 experiments are then carried where every time a specific subset of 100 is used as "test" data while the remaining subsets containing the rest 400 cases are merged to form "training" dataset. The computed performance metrics are then summed over the 5 folds.

4.6.4 Feature Selection

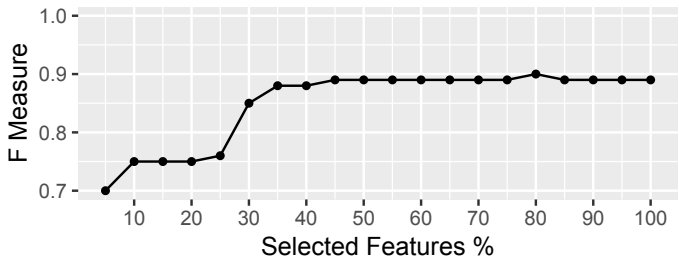


Figure 4.5: F Measure v.s. Selected Features.

In the empirical dataset, the vocabulary contains 1108 unique words. To avoid over-fitting, we select only a portion of them for Phase 1. We use the χ^2 test for each word [177]. A higher value of χ^2 indicates a stronger correlation between the word and the category. Figure 4.5 shows the relation between the F Measure of Phase 1 and the percentage of words chosen (ranked in descending order by χ^2 values). We can see that with the top 50% of the features, the overall performance already stabilizes. We then always use top 50% of the words in the vocabulary.

4.7 Study Results

4.7.1 RQ1: Effectiveness of Categorization

We use our ground truth of 500 crashes to assess the performance of ANCHOR in the first phase of the approach, namely the categorization. We provide in Table 4.3 the confusion matrix as well as the precision and recall of our experimental results. ANCHOR yields a very high precision for predicting crashes in Category A, reaching 0.96. The precision for crashes in Categories B and C are comparably lower, at 0.65 and 0.60, respectively. In terms of recall, the approach is effective for Category A (0.91), Category B (0.82), and Category C (0.75). Overall, ANCHOR is successful in categorizing 444 out of 500 crash samples (88.8%).

Table 4.3: Effectiveness of Categorization (Phase 1)

	Actual			
	A	B	C	Total
Predicted as Category A	374	6	8	388
Predicted as Category B	20	40	2	62
Predicted as Category C	17	3	30	50
Total	411	49	40	500

	Precision	Recall
Category A	0.96	0.91
Category B	0.65	0.82
Category C	0.60	0.75

Answer to RQ1: ANCHOR is overall effective in categorizing new crash samples. However, there is still room of improving the precision when predicting samples in Categories B and C.

4.7.2 RQ2: Effectiveness of Localization

To evaluate the localization phase of ANCHOR, we consider sample crashes for each category and assess the rank localization yielded by the specific algorithm developed for that category. Table 4.5 summarizes the Recall@k (with $k \in \{1, 5, 10\}$ and MRR.

To make sure the evaluation of Phase 2 is not affected by the outcome of Phase 1, we propose to assess the performance of localization with the assumption of perfect categorization.

Table 4.5: Localization Performance

Category	Recall@1	Recall@5	Recall@10	MRR
A	0.97(400/411)	0.99(406/411)	0.99(407/411)	0.98
B	0.39(19/49)	0.61(30/49)	0.63(31/49)	0.48
C	0.78(31/40)	1.00(40/40)	1.00(40/40)	0.85
Total	0.90(450/500)	0.95(476/500)	0.96(478/500)	0.92

For cases in Category A, the true fault location can almost always be found at the top of the rank. The high value of MRR at 0.98 confirms the intuition in Section 4.5.2.1 that it takes much less effort in finding fault location for Category A. For cases in Category B, the recall@1 starts at 0.39 and increased substantially for recall@5 at 0.61. One more case is successfully located with recall@10 at 0.63. The overall MRR is 0.48. Given the fact that the search space is vast (there can be tens of thousands of developer methods in the apk), Algorithm 1 demonstrates decent performance. For most cases in Category C, the true sub-category of the fault location can be found topmost, with the MRR at 0.85.

Answer to RQ2: The localization algorithms (Phase 2) of ANCHOR are reasonably effective for suggesting the correct fault location. ANCHOR shows descent performance even when challenged by the vast search space for crashes in Category B.

4.7.3 RQ3: Overall Performance of Anchor

Table 4.6 summarizes the overall performance of ANCHOR combining Phase 1 and 2. The MRR of all 3 categories slightly dropped, since some cases are miscategorized in Phase 1. Clearly, the overall performance is affected by Phase 1. However, since the two phases in ANCHOR are loosely coupled, we envisage improvements of overall performance in the future when better classifiers are proposed.

Table 4.6: Overall Performance of ANCHOR

Category	Recall@1	Recall@5	Recall@10	MRR
A	0.90(370/411)	0.91(373/411)	0.91(373/411)	0.90
B	0.37(18/49)	0.59(29/49)	0.61(30/49)	0.46
C	0.72(29/40)	0.75(30/40)	0.75(30/40)	0.73
Total	0.83(417/500)	0.86(432/500)	0.87(433/500)	0.85

Answer to RQ3: ANCHOR is an effective approach for locating crashing faults when they are in/outside stack traces, even outside code. Better performance is guaranteed when categorization (Phase 1) is further improved.

4.7.4 RQ4: Performance in the Wild

The heuristics based on which ANCHOR is built may be biased by the empirical dataset. To mitigate this threat, we assess the effectiveness of ANCHOR with a dataset selected in the wild. We want to

verify to what extent ANCHOR can be generalized. We leverage the independent dataset prepared by Fan et al. [56] who thoroughly (by crawling the entire GitHub) and systematically (by applying strict criteria) collected 194 crashed apks from open-source Android repositories. Before evaluation, we apply the constraint rules of Section 4.4.1, and focus on the 69 relevant crash cases that could be identified. Note that this dataset contains true fault locations already verified by the app developers. Since the cases in the dataset are from a wide time span (2011-2017), the partition is randomly decided on normal distribution over the year of app release.

Table 4.7: Categorization on an independent dataset.

	Actual			
	A	B	C	Total
Predicted as Category A	54	1	0	55
Predicted as Category B	3	6	0	9
Predicted as Category C	1	0	4	5
Total	58	7	4	69

	Precision	Recall
Category A	0.98	0.93
Category B	0.67	0.86
Category C	0.80	1.00

Table 4.7 shows the confusion matrix, as well as the precision and recall of Phase 1 (categorization) on this independent dataset. The precision for all categories is high, reaching 0.98 (54/55), 0.67 (6/9), and 0.80 (4/5) respectively. The recalls are also high, at 0.93 (54/58) for A, 0.86 (6/7) for B, and a perfect 1.00 (4/4) for C.

Table 4.9 provides measures for the overall performance. To compute the similarity scores which are required to locate the bug related to crashes from Categories B and C, we use the crash records from the empirical dataset. The recalls and MRR in Category A remain high. As for Category B, ANCHOR is able to yield recall@k values and MRR of 0.43 when suggesting fault locations. As for Category C, the total MRR is at 0.43, suggesting more stack traces in Category C might be the key for better performance.

Table 4.9: Recall@k and MRR on an independent dataset.

Category	Recall@1	Recall@5	Recall@10	MRR
A	0.72(42/58)	0.93(54/58)	0.93(54/58)	0.81
B	0.43(3/7)	0.43(3/7)	0.43(3/7)	0.43
C	0.25(1/4)	1.00(4/4)	1.00(4/4)	0.40
Total	0.67(46/69)	0.88(61/69)	0.88(61/69)	0.74

Answer to RQ4: The evaluation on an independent dataset shows that ANCHOR can be generalized. ANCHOR is a milestone in this respect that it considers various crashing location cases. However, a community effort is still required to construct a representative dataset of crashes to push forward the state of the art in crashing fault localization.

4.8 Disscussion

4.8.1 Comparing Anchor with other Locators

Along with their empirical analysis of Android app crashes, Fan et al. [56, 217] mentioned a prototype crashing fault locator: ExLocator. Unfortunately, since the tool has not been publicly released, we could not directly compare it against ANCHOR. We note, based on its description, however, that ExLocator has a limited usage scenario since it focuses on only 5 exception types. Fan et al. [55] also studied one specific type of exception locating. CrashLocator [244] can also locate faults outside the stack trace. However, CrashLocator needs to abstract patterns from a great number of

repeated crashes of the same project. Unfortunately, for both datasets presented in this chapter, this requirement is not satisfied. Moreover, CrashLocator requires source code and change tracking of the projects, unavailable for our empirical dataset. Therefore, we are not able to apply CrashLocator.

Although direct comparison in terms of effectiveness is not possible in this scenario, we can compare the applicability. ANCHOR is considered to have a wider application range compared to ExLocator, i.e., it can be applied to all exception types, and considered to be more lightweight and source code independent compared to CrashLocator, i.e., it requires only the crash message and the apk.

4.8.2 Developer Effort for Locating Bugs

In the motivating example, we demonstrated why locating buggy methods outside the stack trace can be challenging. We also want to measure the effort that developers put in locating such bugs. In Fan et al.'s dataset, each crash is documented with its duration, i.e., the time between the issue creation and its official closure by the developers. For bugs in the stack trace, it takes developers 26 days on average to close the issues. For bugs outside the stack trace, it drastically increases to 41 days. The ratio is 41/26=158%. Although it may not always be precise to measure effort in terms of issue duration, this would confirm our observation to some extent.

4.9 Threats to Validity

4.9.1 Internal Threats

In the empirical study presented in Section 4.4, we have manually built the ground truth of buggy locations that we made available to the community. Although we have tried our best to perform this manual inspection with the help of (1) the Android official documentation, (2) programmer information exchanging forums like StackOverflow or GitHub, (3) tools such as Soot or CodeInspect, there is no guarantee that all buggy locations we retrieved are the true causes for the crashes. This might affect the conclusions we draw from this dataset and the answers to RQ1-RQ3.

4.9.2 External Threats

We extracted our dataset from the ReCBench (c.f Chapter 5). Although the large dataset contains diverse apks collected from various popular app markets such as Google Play (ensuring a good diversity of apps), the collected crash cases are retrieved by testing apks with only two testing tools. Therefore, the yielded crashes could not be representative of the whole spectrum of crashes present in the Android ecosystem. Similarly, the dataset proposed by Fan et al. [56] is extracted from open source Android app GitHub repositories only. Moreover, they have applied certain rules for collecting the crashed cases, e.g., they extract only crash bugs that have been closed by repository maintainers. The potential limitations with both datasets may affect the effectiveness we have shown in RQ1-RQ4.

4.10 Related Work

A recent survey by Wong et al. [243] marks the activity of identifying the locations of faults in a program to be most tedious, time-consuming, and expensive, yet equally critical. Therefore, lots of techniques have been proposed attempting to ease the work of finding the fault locations. Although we did not find a dedicated tool for identifying locations in Android apps, there are some approaches proposed for general software programs. For example, Wu et al. proposed CrashLocator [244] to

score and rank suspicious locations that have caused program crashes. CrashLocator suggests that the buggy methods can be located in the static call graphs extended from the stack traces. However, it is not suitable to work on programs with multiple entry points and separate call graphs such as Android apps. Moreover, its scoring factors, which require source code and change histories, also limit its application scope to Android apps, for which most of them are released in a closed way (i.e., no change histories). Gu et al. [86] proposed another approach called CraTer that adopts information retrieval techniques to predict whether the real fault resides inside the stack traces. However, CraTer is not able to suggest the actual buggy location. BugLocator [273] applies a revisited Vector Space Model (rSVM) to retrieve relevant files for fixing a bug on a large number of bug reports. However, its granularity falls in file level, which still requires human verification for more fine-grained location identification. Wong et al. [242] build their work on top of BugLocator [273] and leveraged stack trace to improve the approach and indeed achieved better performance. Fan et al. [56] briefly describes a fault localization prototype ExLocator for Android apps. ExLcator only supports 5 exception types and has a limited usage scenario. APEChecker [55] manifests async programming errors and can statically spot faults for this specific crash inducing error. Furthermore, in the community of Automatic Program Repair (APR), statement-level fault localization is often among the first few steps. Researchers have improved it in various aspects [5–7, 110, 126, 128, 236].

Many research works have been proposed to address Android app crashes in recent years. For example, Fan et al. [56] performed a large scale analysis on framework-specific Android app crashes. They have invented the grouping techniques to group the Android app crash cases into buckets to study similar root causes based on each bucket. Researchers have also spent efforts attempting to automatically reproduce the reported crashes [139, 170]. Indeed, to achieve this purpose, Zhao et al. have proposed ReCDroid [270], which applies a combination of natural language processing (NLP) and dynamic GUI exploration to reproduce given crashes. Gómez et al. [71] proposed another approach for reproducing crashes by providing sensitive contexts. Moran et al. [183] further presented a prototype tool called CrashScope, aiming at generating an augmented crash report to automatically reproduce crashes on target devices. Researchers have gone one step deeper to propose automated tools to automatically fix such identified crashes. Indeed, Tan et al. [223] have proposed an automatic repairing framework named Droix for crashed Android apps. Droix adopts 8 manually constructed fixing patterns on crashed Android apps to generate app mutants and suggest one that fixes the crash. Following this work, we present to the community an automatic fix pattern generation approach named CraftDroid for fixing apps suffering from crashes (c.f Chapter 5).

The special Android callback-based mechanism and its effect have drawn the attention of many researchers with the ever-booming of Android devices. Yang et al. [255] targets the even-driven and multi-entry point issue of Android apps, and proposed a program representation that captures callback sequences by using context-sensitive static analysis of callback methods. Flowdroid [23] targets at exposing privacy leakages on Android phones. It establishes a precise model of the Android lifecycle, which allows the analysis to properly handle callbacks invoked by the Android framework. Relda2 [245] is a light-weight and precise static resource leak detection tool based on Function Call Graph (FCG) analysis, which handles the features of the callbacks defined in the Android framework. Together with other existing works like [138, 257], they all dealt with Android callback-based mechanism in various manners. Although these works are different from ours, their approach in handling lifecycle and callback methods could be borrowed to enhance our approach towards better dealing with Category B crashes.

4.11 Summary

In this chapter, we performed an empirical study. This study shows that 37% crash types are related to bugs that are outside the stack traces, which imposes challenges to the localization problem. We then proposed ANCHOR, a two-phase categorization and localization tool that is able to generate a

ranked list of bug locations for developers to examine. The effectiveness of ANCHOR is assessed with both this empirical dataset and an in-the-wild scenario on a third-party dataset. Our work brings inspiring insights into the crashing faults localization problem for Android apps and calls for more attention from both the developers and the research community.

5 Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems

Recently, the program repair community has started to investigate the possibility of fixing Android app crashes automatically. Current results, however, apply to limited example cases. The main issue is the need for more example data to drive the fix processes due to the high cost in time and effort needed to collect and identify fix examples. In this chapter, we propose a scalable approach, `CraftDroid`*, to mine crash fixes by leveraging a set of* 28 *thousand carefully reconstructed app lineages from app markets, without the need for the app source code or issue reports. We develop a replicative testing approach that locates fixes among app versions that output different runtime logs with the exact same test inputs. Overall, we have mined* 104 *relevant crash fixes, further abstracted* 17 *fine-grained fix templates that are demonstrated to be effective for patching crashed apks.*

This chapter is based on the work published in the following research paper:

- Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue- and change-tracking systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–89. ACM, 2019

Contents

5.1 Overview **72**
5.2 Motivating Example **73**
5.3 Study Design **74**
5.3.1 Phase I: Fix Mining 74
5.3.2 Phase II: Fix Grouping and Fix Template Abstraction 77
5.3.3 Patching Crashed Apks 78
5.4 Dataset and Statistics **79**
5.4.1 Crash Fixes from Lineages (from Phase I) 79
5.4.2 Fix Buckets & Fix Templates (from Phase II) 81
5.4.3 ReCBench for Evaluating Bug Patches 82
5.5 Study Results **83**
5.5.1 Explored Crashes vs. Reported Crashes 83
5.5.2 Benchmarks for Evaluating Patches 83
5.5.3 Evaluating Fix Templates on ReCBench 84
5.6 Threats to Validity **86**
5.7 Related Work **87**
5.8 Summary and Future Work **88**

5.1 Overview

Nowadays, computing on Android-powered devices is pervasive across all age and status segments in society. A concern that is commonly shared by the whole user base, however, is that Android apps are prone to crash [240]. This is due to many reasons, including compatibility issues with devices, poorly tested functionalities, etc. A number of studies [140, 240] on the reviews that users provide on Google Play even list this proneness to crash as one of the most recurrent complaints about apps, especially in comparison with apps on the main concurrent system, namely iOS.

A major advantage of Android in attracting developers is that its maintainers provide a Software Development Kit (SDK) which builds on familiar programming concepts, languages and tools. An extensive open source Application Programming Interface (API) is further provided to facilitate the exploration of device resources, and programming features are regularly extended for building ever-fancy mobile apps. In this context, and given time-to-market pressure, most apps are shipped to the market without being fully tested. A situation that is exacerbated by the fragmentation of the Android operating system: a given app may eventually have to run on over 20,000 unique device models [4] functioning at different API levels [240]. This, added to the fact that most Android app developers are not professional developers, leads to the situation where apps inevitably crash during operations.

Android app crashes yield stack traces pointing to a raised exception that developers must address. Framework crashes are the most difficult to debug, because the exception is thrown within the Android framework code, and developers can easily get lost in the enormous code maze. Although detailed documentation on API usage is provided by Google, and many testing platforms, as well as stack analysis tools, have been proposed by the research community, it is still far from being trivial to fix a crash. Nevertheless, with the huge interest in automated program repair in the software engineering community, there is an opportunity for building and applying some simple but effective online repair approaches targeted at Android recurring crashes.

Unfortunately, as Fan et al. [56] recently pointed out, there is a lack of comprehensive datasets of true crash fixes. This is a major obstacle which has prevented extensive research on crash analysis within the mobile community. Currently, authors turn to open source development code bases to track crash fixes based on the crashes reported in issue tracking systems. Fan et al. managed to identify 194 such issues for conducting a seminal study on Android crash fixes [56]. Such a process, along with the resulting dataset, presents several major limitations:

- There is a threat to external validity as only open source apps can be concerned. The dataset is not representative since a number of crashes may never be reported in issue tracking systems, and yet fixes have been applied to address them.
- The collection process of crash fixes is not scalable. Authors build crawlers to analyze GitHub repositories and select potential closed issues. Then, they must manually verify in the code that the issue is real and that the provided fix is indeed related to the announced crash.
- Open source apps often deal with simple functionality scenarios, and generally, have a smaller code base compared to commercial ones. Thus there may be fewer occurrences of crashes.
- Finally, this process cannot be replicated in commercial development settings, which do not provide useful information on bug reports, or means to reproduce bugs, and information on how they were eventually fixed. The limited information available is often within release notes where developers may vaguely mention that a bug fix was performed.

This chapter. Our work in this chapter deals with the automation of mining Android crash fixes in the wild. We propose to undertake an expensive testing campaign in a dataset of 450 thousand apks to replicate crashes in app lineages and retrieve fixes. Overall, the contributions of this chapter are as follows:

1. We design a workflow for collecting crash fix datasets in a fully automated and scalable manner in the absence of issue-tracking systems.
2. We implement the necessary toolset and experiment on a large-scale dataset of apps from the AndroZoo repository [11] and benefit the community with a benchmark comprised of app crashes and scripts for automatic test input replication.
3. We show that the fix templates mined are effective in patching crashed Android apps.

The remainder of this chapter is organized as follows. Section 5.2 motivates this work with a concrete example. Section 5.3 presents the overall methodology of this work. Section 5.4 depicts the dataset we adopt and the results our approach obtains. We then evaluate our approach in Section 5.5, followed by a discussion and related work in Section 5.6 and Section 5.7, respectively. Finally, Section 5.8 concludes this chapter.

5.2 Motivating Example

To better motivate this work, we now provide a concrete example demonstrating a crash and its fix that is, eventually, automatically obtained by our approach presented in this chapter.

When random testing the *WhatsApp* app (release version of 30-09-2013), our approach observes a runtime crash. Listing 5.1 illustrates the crash message that can be retrieved (e.g., via the *logcat* command) from the device running the app. The crash message starts with an exception type, here *java.lang.SecurityException* (cf., line 1), which is thrown by the security manager to indicate that there is a security violation. Then, a detailed message will be given to explain the possible reason behind the crash. In this example, the reason is that neither user 10074 nor the current process has been granted with the following Android permission: `READ_ PRECISE_PHONE_STATE` (cf., line 2). After this detailed explanation, a call chain (cf. lines 3-8) will be listed showing the crash point (which is normally the first method in the call chain, cf. line 3) and the methods that are successively called until reaching the crash point. Observant readers might have already noted that the methods in the call chain include framework APIs (e.g., line 6, *android.telephony.TelephonyManager.listen*) and app methods (e.g., line 7, *com.whatsapp.RegisterPhone.onCreate*) that are actually written by app developers.

```
1 1 java.lang.SecurityException: Neither user 10074 nor current process
2         has android.permission.READ_PRECISE_PHONE_STATE
3 2   at android.os.Parcel.readException
4 3   at android.os.Parcel.readException
5 4   at com.android.internal.telephony.
6         ITelephonyRegistry$Stub$Proxy.listenForSubscriber
7 5   at android.telephony.TelephonyManager.listen
8 6   at com.whatsapp.RegisterPhone.onCreate
9 7   at android.app.Activity.performCreate
```

Listing 5.1: Crash Message of *WhatsApp*.

After observing the aforementioned crash, one can apply the same testing strategy (i.e., the same test inputs) to a later version (released on 30-04-2014) of the *WhatsApp* app. Interestingly, the app does not crash anymore. Hence, we can suppose that the later version of the *WhatsApp* app has somehow fixed the crash issue illustrated in Listing 5.1. To further verify the existence of modifications, we leverage SimiDroid [134] and perform a method-level pairwise comparison of these two app versions. Because framework APIs cannot be changed by app developers, we focus on app methods appearing in the call chain. Listing 5.3 summarises the *diff* message resulted from the pairwise comparison.

Fortunately, in this example, there is only one app method appearing in the call chain, which is *com.whatsapp.RegisterPhone.onCreate*. In this method, a framework API is used. The invoked API method is *android.telephony.TelephonyManager.listen*. This method registers a listener object to

receive notification of changes in specified telephony states. In the crashed version, *int* -1 is passed as the second parameter while in the fixed version the value of the second parameter changed to 1535. By investigating the Android API documentation, we understand that the second parameter is the outcome of the bitwise `OR` operation of multiple `LISTEN_Flag`, representing the code for the state change that the app wants to listen to. As specifically mentioned in the documentation, some state changes are protected by permissions, i.e., developers need to declare the appropriate permission in order to access the state change information. The value, -1, is actually the outcome of the combination of all the possible state changes, where permission `READ_PRECISE_PHONE_STATE` is required. Unfortunately, this permission is not declared in the *AndroidManifest* configuration file, leading to crashes as expected. In the fixed version, the value of the second parameter is changed to 1535, after decoding it to binary, representing a combination of four flags, where none of them requires permission `READ_PRECISE_PHONE_STATE`.

```
1 void onCreate(android.os.Bundle $Bundle){
2     ...
3     -  $TelephonyManager.listen($PhoneStateListener, -1)
4     + $TelephonyManager.listen($PhoneStateListener, 1535)
5     ..
6 }
```

Listing 5.2: Illustration of Mined Fix from *WhatsApp*.

This motivating example demonstrates that it is possible to learn practical fixes to Android runtime crashes by learning from the code contributed by app developers (e.g., during the evolution of a given Android app). In this work, we call the two app versions ($Apk_{30.09.2013}, Apk_{30.04.2014}$) as a crashed-fixed pair. Our objective in this work is hence to mine crashed-fixed pairs from real-world Android apps so as to learn fix hints from them and benefit the research community with such dataset.

5.3 Study Design

Figure 5.1[1] presents an overview of the proposed `CraftDroid` approach. It is composed of two phases.

Phase 1: The *Fix Mining* phase consists of 3 steps. The objective of this phase is to mine fixes to Android crash-inducing bugs through a carefully designed approach leveraging both testing and static analysis.
Phase 2: The *Fix Grouping and Fix Template Abstraction* phase utilizes another 2 steps, to group the aforementioned fixes based on testing results feature identification and a manual abstraction method to abstract fix templates.

The crashed apks set as well as the test inputs replication scripts from the *Fix Mining* phase further form a benchmark called *ReCBench*, and will be utilized to showcase the effectiveness of our approach, as well as a contribution to the community.

5.3.1 Phase I: Fix Mining

The first phase unfolds in three steps: (1) App Lineage construction, (2) Crash Exploration and (3) Fix Verification. Each of these steps is implemented in a module whose details are provided below.

[1] Icons from www.flaticon.com

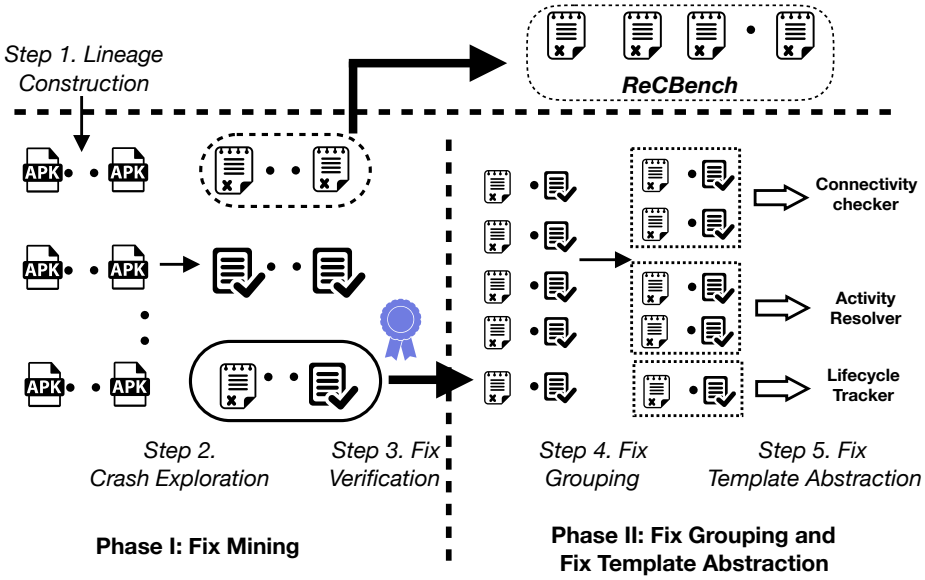


Figure 5.1: Overview of CraftDroid.

Step 1 - App Lineage Construction

The concept of app lineage (i.e., a series of apk releases of a given app) was first introduced by Gao et al. [65]. In this step, we use the same approach to construct app lineages from AndroZoo apps. Overall, the app lineages are constructed via the following process: (1) identify unique apps, where APKs sharing the same *package name* are considered to be the same app, and (2) link and order the different app versions of the same app. As a result, an app lineage contains a set of Android apps that share the same package name while are totally ordered based on their release time. Note that a lineage can be sparse given that AndroZoo is not exhaustive in the collection of app versions. Nevertheless, as we will show in the next phases, this does not impact the soundness of `CraftDroid`.

Step 2 - App Crash Exploration

This step aims to pinpoint real-world apps that crash at runtime. The most straightforward and reliable approach to achieve that objective is to launch a testing campaign on the apps. In our case, we generate diverse UI inputs, aiming to automatically and dynamically explore Android apps. Ideally, in order to cover as many crash cases as possible, we should employ different testing strategies. However, as dynamic analysis tools are known to be time-intensive, it is practically hard to adopt all possible random testing tools available in the Android community. Moreover, during our testing step, we need (1) to ensure that the testing tools generate the same sequence of testing inputs, (2) to ensure that the testing environment, i.e., the Android device emulator and the state of the operating system are identical for the apks of the same lineage (3) to ensure that our approach is scalable in order to quickly explore all apks for a huge amount of lineages within a limited time budget. We now detail these three requirements.

Replicative Testing Inputs. In order to achieve optimized crash exploration and to apply the same execution scenario, we utilized 3 testing policies from 2 testing tools. Monkey is a built-in random testing generation tool for Android emulators. The testing procedure will be reproducible if the selected seed is identical. Droidbot [147] is a light-weight test input generation tool for Android apps. It generates input events based on the analysis of *AndroidManifest.xml* and runtime objects. However, *Droidbot* currently does not have an option for reproducibility. For the purpose of this work, we contribute to DroidBot by implementing a reproducibility module. This module essentially records the sequence of events generated for the first app version of any lineage. If needed, the module can

apply the recorded events to new app versions so as to reproduce the events fired previously. We utilized two of *Droidbot*'s testing strategies, *dfs_greedy* and *bfs_greedy* for better coverage.

Restored Testing Environment. To ensure that each app version is tested in an identical environment, we used the same emulator instance for all app versions of any lineage. In order to mitigate the potential influence of remnant files and caches from both app installation time and runtime, we uninstall the previous app versions and carefully clear all temporal files prior to new app version testing steps. To make sure the test log collected is confined for each testing procedure, we also clear the buffer for the logging system in the emulators.

Scalable Testing Process. Since the testing process across different lineages is fully parallelized, we leverage the abilities of multi-core computing machines. `CraftDroid` detects the number of CPU cores on the computing device and starts as many Android emulator instances. Each lineage will be tested on a single instance and the whole process will scale to the number of available cores.

After the test procedure finishes on each apk (for each different test strategy/tool), we use the built-in logging tool *Logcat* from Android emulator to filter out the testing information. The log will contain basic information about the app, e.g., app package name, process id, etc. Especially, when the app crashes during the testing procedure, the crash log, including the exception type, crash message and stack trace will also be logged. Furthermore, we collected 200 crashed apks, each from a unique lineage, together with the scripts to reproduce them to form a benchmark called *ReCBench* (*Re*producible *C*rash *Bench*). *ReCBench* will be used in below sections to the evaluation of the effectiveness of our approaches, and also to benefit the community.

Step 3 - App Fix Verification

After the testing process finishes in *Step 2*, we then start to look for app crash-inducing bug fixes.

Analyzing Testing Logs. First, we analyze the test logs. To start, we give the definition of fixes present in lineages. Given an Android app a_1 that crashes following test inputs t, if a_1's subsequent version a_2 does not crash following the same test inputs t, we hypothesize that a_2 has been released with fixes to the crash appearing in a_1. In other words, a_2 contains practical code changes addressing the crash issue observed in a_1. Given an Android app a_1 with its crash information (i.e., stack trace), the objective of this step is to identify app a_2, which must thus satisfy two conditions:

- a_2 is released after a_1.
- a_2 no longer crashes following the same test input.

If such an app is identified, we mark it as the fixed version and hence put it with app a_1 into a pair, i.e., the potential "crashed-fixed" app pair (a_1, a_2). To avoid flaky cases, we further impose restrictions on our approach to only consider such cases where the subsequent app versions throughout the end of the current lineage do not crash anymore under the same tests.

Inspecting Code Changes. Then, with the potential crashed-fixed app pair identified, our approach performs a pairwise comparison between the two apps and records all the changes as potential fix changes. Unlike when working with change-tracking systems, app version updates represent a series of changes performed for various reasons, including adding new functionality to the app, fixing various functional and non-functional bugs, etc. Thus, not all recorded changes are relevant as crash fix modifications. Therefore, we limit ourselves to such changes that are made to the methods appearing in the stack trace of the runtime crash.

A stack trace, as illustrated in Listing 5.1, includes details on the method during whose execution the crash occurs, as well as the path (i.e., call chain) of execution flow that led to this point.

In this work, we consider all the methods that are from the Android framework as framework methods, and those that are not from the Android framework as app methods which should be considered to

look for modifications that fix crashes in apps. Depending on the signaler, i.e., the framework method which instantiates and throws the exception type, app crashes are divided into *framework-specific* crashes and *app-specific* crashes. In this chapter, we only study framework-specific crashes. We consider app-specific crashes to be out of the scope of our research interest, given that they mainly fall into below three categories, which can be readily diagnosed and whose fix patterns are straightforward to craft:

1. Most exceptions thrown are *NullPointerException*. This exception happens when the program tries to invoke a method on a null object reference, it often arises because of the developers forget to initialize variables or adding null checkers.
2. *ClassCastException* is thrown when developers try to cast a runtime instance to a nonmatching type.
3. Exceptions created and thrown by app methods. This means that the developer is aware of the condition when the exception is thrown and the fix will be very much app specific.

On the contrary, we find that framework-specific crashes and their fixes are more challenging and valuable for mainly 3 reasons:

1. The signalers that throw the exception instance are system methods, and are unseen to developers unless developers are very aware of AOSP (Android Open Source Project).
2. There might be long call layers between the signaler and the API that developer called, making the condition for triggering the exception more complicated.
3. The Android documentation is never fully clear about when and how an exception will be thrown upon using an API.

Since our apps are all tested on the same Android emulator instance, we consider that the framework method implementations are identical among stack traces, and thus we only need to compare the app methods. For each app method (e.g., *DevMethod1* in Figure 5.1), we perform a pairwise comparison at the code level between the two apps in the crashed-fixed pair and generate a *diff* snippet representing the changes made by the fixed app version. If any modification is present, we will manually check whether the modification is a true fix to the prior crash.

5.3.2 Phase II: Fix Grouping and Fix Template Abstraction

After collecting true fixes in Phase I from `CraftDroid`, we extract features from the stack trace of crashes, and group the fixes into buckets based on the features. Subsequently, we shall abstract fix templates from the groups. These fix templates are patterns of code change actions that can be applied to crashed apks in the cases of specific crash types.

Step 4 - Fix Grouping

For each crashed-fixed app pair (a_1, a_2), we analyze the crash stack trace from a_1 and collect 3 feature information: *Exception* is the exception class thrown that caused the crash; *Signaler* is the topmost system method that created and threw the exception; *CrashLoc* is the framework API that the app method called and which passed back the exception. Let us take Listing 5.1 as an example. The extracted features are *(SecurityException, readException, listen)*. Note that in this example, for better writing structure we used only method names, wherein actual work we use full-length identifiers. Fan et al. have also constructed similar tuple for grouping and finding the root causes for Android crashes. However, our tuple is different since we adopted *CrashLoc* instead of crash message. Because we consider the fixes are closely related to the context of the usage for the APIs. With the features extracted for each crashed-fixed pair, we then group fixes simply based on comparing their feature tuple and build buckets for these fixes. In other words, all fixes with the same extracted features

from the crash stack traces (i.e, the three elements $Exception, Signaler, CrashLoc$ are identical) are grouped into the same bucket.

Step 5 - Fix Template Abstraction

For each bucket, we follow the below protocols to manually look for fixes.

1. Based on the stack trace, we start from studying the code changes in the developer method which calls the crashed Android framework API, i.e., the crash location.
2. If fix is not found, we turn to the next developer method towards the bottom of the stack trace. We repeat this, until we find a fix or we reach the last developer method.
3. Once we find a fix, we turn to official Android development documentation, as well as online discussion forums to validate our finding.
4. If no fix is validated after studying all developer methods, we consider in this work that the code changes are irrelevant.

From the validated fixes, we summarize the most common fix template. These templates are fine-grained since they target to only solve crash-inducing bugs related to the misuses of specific framework APIs. Although automatically mining of templates is researched in the literature (cf. recent work of Koyuncu et al. [125]) they often lead to noises about patterns unrelated to repair changes due to the issues of tangled commits [49, 93, 154, 188]. We leave automation in this part as future work.

5.3.3 Patching Crashed Apks

Towards demonstrating the usefulness of `CraftDroid`, as well as the fix templates generated, we propose to apply the templates on crashed apks and evaluate the patched apks. As shown in Figure 5.2, we take crashed apks from ReCBench (cf. Section 5.3.1). As shown in Figure 5.1, this benchmark comes from *Phase I*. ReCBench is built by collecting the crashed apks and the associated testing inputs. For a given crashed apk a_c from ReCBench, to evaluate our fix templates, we follow a three-step process: 1) First, the crashed apk a_c is categorized into one of the buckets we collected by extracting features from the crash stack trace of a_c. 2) Then we use Soot [228] to decompile a_c into Jimple files[2] and retrieve the file that contains the app method which called the API. We manually apply the fix template associated with the corresponding bucket. 3) We re-compile the files into apk and run the test with the patched apk. This process was first introduced in [223]. If the patched apk does not crash with the same testing inputs, we consider our template valid.

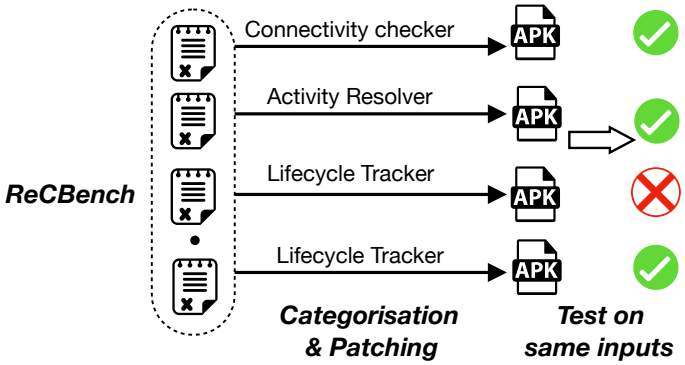


Figure 5.2: Patching and Evaluation.

[2]Jimple is the intermediate representation of Soot

5.4 Dataset and Statistics

5.4.1 Crash Fixes from Lineages (from Phase I)

Lineage Construction

We start by retrieving a maximum number of apps from the AndroZoo repository [11] using the API key provided by the maintainers of AndroZoo. We managed to collect 4 million apps to start our experiments. Although AndroZoo currently makes available over 8 million apps[3], we believe that the collected dataset is largely representative of this study. From the collected app set, we eventually re-constructed $28K$ app lineages containing around 450 thousand apks: although we could find many more lineages, we focused on cases where there are at least 10 app versions. The purpose of setting this threshold is that we want to mine for code changes among released versions that would result in contrary testing outcomes. Shorter length of lineage would not be able to reflect such changes, and would pose obstacle of filtering out flaky outcomes.

Test Environment Set-up

In order to set up the testing environment for the lineages, we instantiate multiple Android emulators using the same image so that we can do testing parallelly and make full use of multi-core computing machines. Note that we have chosen x86 as ABI (Application Binary Interface) for these emulators. Since we are running these emulators on x86-based servers, choosing x86 over arm64-v8a as ABI makes emulators run substantially faster, because the instruction set is shared among the virtual machine and the host machine. The drawback is that apps making use of the Android Native Development Kit (NDK) are not installable and would report an installation error complaining about non-matching ABI. During our experiment, we have also observed other kinds of installation errors (e.g., the builds are too old w.r.t. API level). In total, around 56% apks are installable and later tested on the emulators. Although the percentage for the set of installable apks over all apks are not ideal, we consider it acceptable since emulators run much faster than real-world Android devices. Moreover, the scalability of emulators makes it capable of running a large collection of lineages in reasonable time budget. During our experiments, we have observed that if one version in a lineage cannot be installed on the emulator, its succeeding releases (app versions) often cannot be installed as well. However, conversely, when a version can be installed, its succeeding releases often can be installed as well. As is illustrated in Figure 5.3 the median of the total number of apks in a lineage should be close to the median of the number of installable apks in a lineage. Note that since every lineage contains at least 10 apks, the minimum number of apks is 10. For better illustration, lineages that contain no installable apks or no crashes are not considered. Outliers are also removed since in rare cases lineages have great length, i.e., the version updates are very frequent.

Test Strategy Application

For the lineages with apps that are installable on our emulators, we launch Monkey as well as Droidbot (using two strategies in the case of this tool) to automatically explore their functionalities, aiming at obtaining runtime crashes. Figure 5.3 presents in the box-plot the distribution of the number of apps that have framework-specific crashes revealed in each lineage. The median number of crashed apks in a lineage is around 3. We then show the results for the 3 testing strategies in Figure 5.4. The *dfs_greedy* strategy of the tool *DroidBot* exposed most lineages with at least one apk having framework-specific crashes. Among those 1160 lineages, there are 371 where successive apks passed the same test inputs and do not crash any more. For the *bfs_greedy*, a similar number of crashed lineages and fixed lineages can be observed. However, with random tests using seed $NO.12$ from

[3] `https://androzoo.uni.lu/`

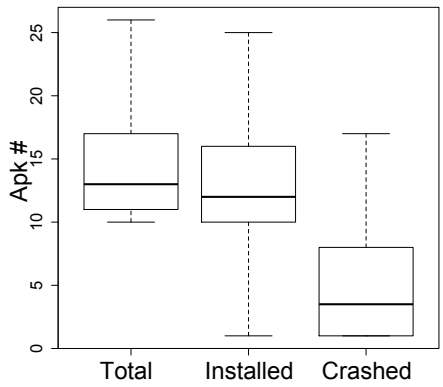


Figure 5.3: Distribution of Total, Installed and Crashed Numbers of Apks in Lineages.

Monkey, we can see that the number of lineages identified to have crashed apks are only half the total number of that of the *dfs_greedy* or *bfs_greedy* strategies of *DroidBot*. This difference in the numbers is because *DroidBot* analyses the *AndroidManifest.xml* and other layout definition files in the apk before generating UI inputs, this gives *DroidBot* higher chance of triggering code execution compared to Monkey. Better coverage, in turn, will result in more crashes being detected. In the meantime, we also see from Figure 5.4 that all 3 strategies would detect similar portion of passed lineages out of lineages with crashes, this indicates that `CraftDroid` is insensitive in the testing tools for mining fixes from the same number of crashed lineages.

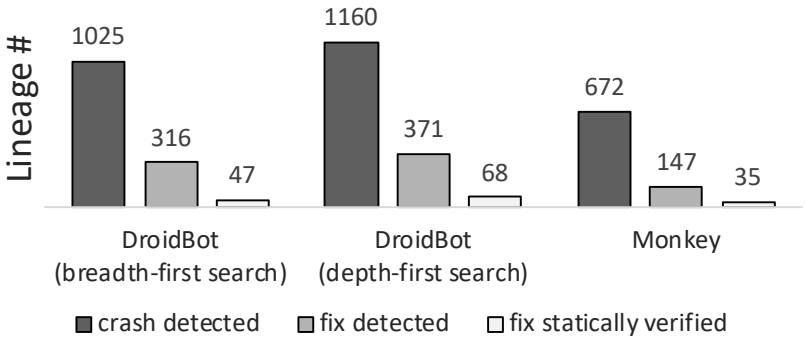


Figure 5.4: Count of Lineages Crashes per Testing Strategy.

Fix Statical Identification

For the lineages selected from the previous steps, we construct a *crashed-fixed* app pair for each lineage. The crashed apk is the last apk in the lineages that has crashed during testing, the fixed apk is the first apk that has passed with the same testing inputs. The reason for the selection is that we consider two adjacent versions will contain the smallest set of code changes related to the app methods that caused the crash. We then use the static analysis tool Soot to decompile both apks and compute diffs on the app methods in the stack trace. For example, from Listing 5.1, the app method that will be compared is *com.whatsapp.RegisterPhone.onCreate*. After static analysis finishes, we succeeded to verify 150 fixes in total, as can be seen in Figure 5.4 ($47 + 68 + 35 = 150$). We then manually inspected these *diffs*, we finally distilled 104 useful fixes. We removed other 46 fixes for mainly two reasons, 1) The API call which throws the exception in the crashed apk is no longer used in the new apk version, 2) The API call appears multiple times, e.g., in a switch-case clause and is uncertain in which call statement the exception was thrown. Although programmatically those two

kinds of code changes are still valid in fixing crashes, we consider them not helpful for the next phase of our approach.

5.4.2 Fix Buckets & Fix Templates (from Phase II)

Table 5.1: Fix Templates

Bucket ID	Acronym	Description	Target Exception
1	Provider Checker	Check if the specified provider is enabled with location manager.	IllegalStateException
2-3	Activity Resolver	Resolve intent for Activity existence before starting new Activity	ActivityNotFoundException
4-6	Resource ID Updater	Switch resource ID with another one	Resouces$NotFoundException
7-10	Lifecycle Verifier	Query the state of the hosting activity upon showing dialog	WindowManager$BadTokenException
11	Sleep not Wait	Call sleep() instead of wait() on Thread	IllegalMonitorStateException
12	Thread Finisher	Switch from call of stop() to interrupt() to end thread	UnsupportedOperationException
13	Redundance Trimmer	Trims redundant call of prepare() on same Thread	RuntimeException
14	State Checker	Check check value of isAlive() on Tread before calling start()	IllegalThreadStateException
15	Package Settler	Set package name for intent upon biding service	IllegalArgumentException
16	Permission Checker	Check permission before sensitive operation	SecurityException
17-19	Range Checker	Check on index range before indexing	(String)/(Array)IndexOutOfBoundsException
20	Emptiness Checker	Check if String is Empty before parsing for int	NumberFormatException
21	Path Verifier	Check the file path existence for database	SQLiteCantOpenDatabaseException
22	Hardware Checker	Check the hosting device has specified hardware feature	RuntimeException
23	Canvas Preconcator	Pass preconcated $Canvas to unLockCanvasAndPost()	IllegalArgumentException
24-31	Nullable Checker	Check for parameters that is not nullable	NullPointerException
32-35	Try-catcher	Surround the statement with try-catch	Exception

In the second phase, as described in Figure 5.1 and explained in SubSection 5.4.2, we group the fixes into buckets and abstract fix templates from each bucket.

Grouping for Fix Buckets

From the previous step, we have confirmed 104 true fixes. We extract features to form a tuple (*Exception*, *Signaler*,*CrashLoc*) from the stack trace of the log of the crashed apk of the *crashed-fixed* pair. When features are the same, fixes are grouped into the same buckets. As a result, we grouped these fixes into 35 buckets. To avoid being too verbose, we are not listing the full detail of the buckets and the features related. We rather summarized the bucket count where same exception classes are thrown in Figure 5.5. The most recurrent exception is *NullPointerException*, taking 8 buckets in total, arising from passing null parameters in API calls. The main reason for the crashes are programs trying to call methods on null object references. The second most common one is *WindowManager$BadTokenException*, with a total of 4 buckets. This exception is thrown when the hosting activity upon which the dialog wants to show its message has entered the finishing state. *RuntimeException* is the super class of the unchecked exceptions, it reflects general problem when running the Android app. *Resources$NotFoundException* is thrown when the correspondent resources, e.g., String, Figure, are not found in the apk. *IllegalArgumentException* is thrown when the passed in parameter for the API cannot be handled by the Android system and will be considered illegal. *ActivityNotFoundException* is thrown when no activity is found to handle intents. Android uses intents to start new activities, both inside the current app and activities in other apps in the device. An implicit intent is used when the current app wants to start the activity from other apps on the device that can perform the required *Action* by the intent. However, there is no guarantee that at least one activity exists on the device that performs the action. Improperly handling these scenarios will cause the *ActivityNotFoundException* being thrown. *SecurityException* is thrown when the required permission has neither been granted to the app at installation time nor runtime. The app *WhatsApp* in the motivation example takes this category. Other exceptions, although occurred only once, also arise from key defects in the app codes.

Abstracting Fix Templates

We then manually extract templates from buckets. Note that similar or same fix templates can be extracted for different buckets. This is especially true when the exception and the *Signaler* is the

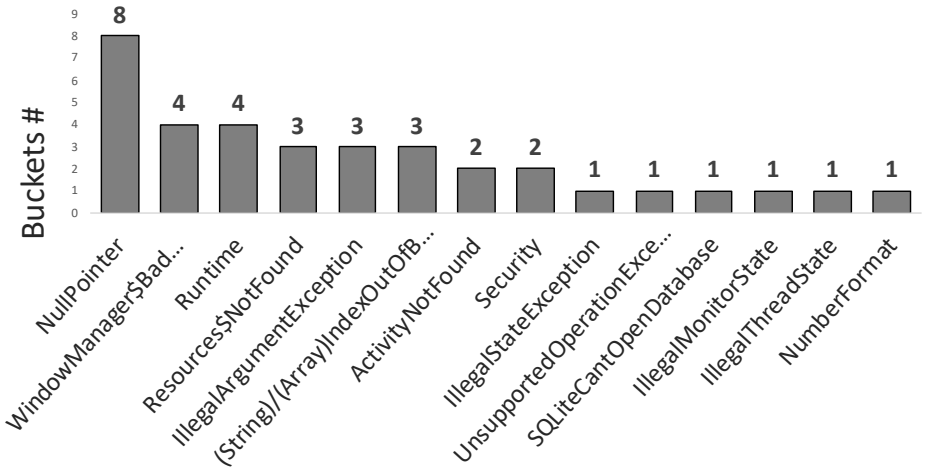


Figure 5.5: Bucket Count for Exceptions.

same. As a result, we are able to settle down 17 fix templates, as listed in Table 5.1. We give an acronym for each fix template for easier reference in the rest of this chapter. *Provider* is a source for providing location information, e.g., network, gps e.t.c.. *Provider Checker* checks on the return value of *isProviderEnabled("provider")* before the app executes *requestLocationUpdates("provider)*. Omitting this when the provider is not enabled will make the system throw *IllegalArgumentException*. The Android system uses *Intent* to start new activities, especially, implicit intents to filter for activities in other apps that can fulfill the required action. However, when no such activity exists in current device, *ActivityNotFoundException* will be thrown. *Activity Resolver* checks for the existence of targeting *Activity* before the app tries to start one. Android apps widely use multi-threading technique to divide light-weight UI response logic and heavy load background logic like downloading. Novice developers will thus be easily confused and make mistakes when handling the threads. We also established 4 templates for fixing thread related crashes. *Sleep not Wait* updates the call from wait() to sleep() on the thread when the app requires a background thread to block other access requests for a certain time period. *Thread Finisher* updates the call from stop() to interrupt() when developer intends to put an end to the thread. *Redundance Trimmer* aims to trim redundant call to prepare() on a thread since it poses *RuntimeException*. *State Checker* aims to check if the thread is alive before calling start() on the thread. *Package Settler* aims to set package name for intent upon biding service, failing this will make the system throw *IllegalArgumentException*. Fix templates 10-15 contain a set of checkers and verifiers for file existence, parameter, permission e.t.c. These are all easily neglected due to uncareful coding. *Canvas Preconcator* aims to preconcate the current canvas instance before calling unLockCanvasAndPost() by calling translate() on the canvas first. The last fix template is *try-catch*. We use *Exception* to summarize that this fix is widely used across multiple buckets by developers. It is often used to deal with corner cases that do not have many occurrences. Developers tend to be lazy for these scenarios. Similar to Tan et al. [223], we consider this template a hard fix, and will not recommend it since it actually suppresses the root cause of the exception.

5.4.3 ReCBench for Evaluating Bug Patches

As explained in SubSection 5.3.3, we take from Phase I the lineages whose testing results contain only crashed apks and propose ReCBench, a benchmark for evaluating patches for crash-inducing bugs. This benchmark contains 200 crashed apks, the crash message filtered out from *LogCat*, and the scripts to automatically reproduce the crashes. The script can be used to re-generate test inputs on the patched apk and filter out log information to decide whether the patch is valid. The dataset is accessible at https://craftdroid.github.io

5.5 Study Results

To evaluate the proposed workflow `CraftDroid` and the fix templates generated, we further address 3 research questions.

- **RQ1:** How do the automatically explored crashes compare with the ones reported in issue-tracking systems?
- **RQ2:** Are state-of-the-art benchmarks capable of evaluating patches generated for crashed Android apps?
- **RQ3:** How good are our mined fix templates in patching crashed Android apps?

5.5.1 Explored Crashes vs. Reported Crashes

In Phase I of `CraftDroid`, we adopted 3 automatic testing strategies to explore inputs that would crash Android apps. However, there is already datasets in the research community that was collected by scanning the issue-tracking systems on open-source App repositories, like the one proposed by Fan et al. [56]. This dataset contains 194 app crashes reported by developers or general users throughout their daily development and usage. We want to compare this dataset with ours to compare the automatically explored crashes and reported crashes in answering 1) Whether our dataset is realistic in reflecting a portion of the crashes cared by users and 2) Whether the automatic testing tools can provide more crash cases omitted by human beings. Please be noted that we utilize our dataset of 104 lineages equipped with fixes, rather than all crashes explored. Similarly, the dataset from Fan et al. also contains crash reporting issues that are closed eventually, i.e., fixed by developers rather than open issues that are not necessarily fixed. The purpose of putting these constraints (i.e., lineages with fixes for `CraftDroid`, and closed issues for Fan et al.) is that we consider these two sets of crashes valid in the sense that they are cared by developers and actually fixed in the later versions and(or) commits. For comparison, we also applied the feature extraction and fix grouping method as presented in Step 4 on the dataset of Fan et al. Recall that the study of Fan et al. collected stack traces in issue tracking systems (although, in the absence of input information, the associated crashes are not reproducible, unlike in `CraftDroid`). As can be seen from Table 5.2, there are in total 9 buckets shared by both datasets. Moreover, `CraftDroid` is able to spot 26 exclusive buckets not reported by issue-tracking systems. Although there are another 117 buckets from the issue-tracking system that we did not cover, given the fact that the authors claimed that they have exhaustively collected all such reported crashes from the online repository collection F-Droid [54] for open source Android apps, we give credit to `CraftDroid` of being scalable and having the potential of exposing more such crashes, e.g., with other testing tools for larger code coverage.

Table 5.2: Buckets Count between Fan et al. and `CraftDroid`

Dataset	Total Bucket	Exclusive Buckets	Shared buckets
Fan et al. [56]	126	117	9
`CraftDroid`	35	26	

Answer to RQ1: `CraftDroid` is able to explore android app crashes that are also reported by issue-tracking systems, with the ability to also cover more crashes that are not yet reported by developers or users.

5.5.2 Benchmarks for Evaluating Patches

Several benchmarks containing Android app crashes have been proposed recently, e.g., Fan et al [56] collected 194 closed issues. Tan et al. also proposed Droixbench [223]. Droixbench is proposed to

evaluate the effectiveness of the automatic patch generation tool Droix. The benchmark contains 24 reproducible app crashes. To reproduce the UI inputs sequences, a runnable script is provided for each crashed apk. However, we found out that Droixbench contains only 9 framework-specific crashes, where the other 15 are app-specific crashes. Since we focus on framework-specific crashes and their fixes, Droixbench does not meet our need. Further, we discover that only 3 out of 9 framework-specific crashes are fixed by Droix, resulting in a fix rate of only 33%. This is much lower than the fix rate of app-specific crashes, where 12 out of 15 crashes are fixed, with the fix rate of 80%. This also supports our observation that framework-specific crashes takes great effort to fix and should draw more attention.

As mentioned in RQ1, Fan et al. [56] also propose a collection of closed issues for framework-specific Android app crashes. We found 3 obstacles in using this dataset. 1) There is no guarantee of apk or version number associated with the issue, people who study this dataset often need to take into consideration the opening time of the issue track, and deduce the version by her/himself. 2) The testing environment, i.e., the specific phone model, the phone states when the crash happens are not easy to re-establish. 3) Most importantly, there are often no clear reproducing steps attached to help reproduce the crash. In most cases, only natural language descriptions are provided. Such shortcomings make it almost impossible to evaluate the patches for the same test inputs.

We thus propose *ReCBench*, a benchmark containing 200 reproducible framework-specific crashes and the scripts to reproduce them. As can be seen in Table 5.3, *ReCBench* contains the higher number of framework-specific crashes among the three benchmarks. Moreover, the total number of app is also 200, which means that it provides higher diversity in terms of the categories of apps. In comparison, the dataset of Fan et al. only contains 44 apps. All the 200 crashes in *ReCBench* are framework-specific. Most importantly, *ReCBench* provides scripts to automatically reproduce the UI inputs that triggered the crashes. Finally, *ReCBench* is largely extensible: in this study, time constraints for ensuring, via extensive experiments, that crashes are reproducible limited the dataset to 200 samples. Our crash exploration phase yielded thousands of crashes which could be later validated and included in *ReCBench*.

Table 5.3: Comparison among benchmarks

Benchmark	App#	Size	Framework	Reproducible
Droixbench	15	24	No	Yes
Fan et al.	44	194	Yes	No
ReCBench	200	200	Yes	Yes

Answer to RQ2: ReCBench contains 200 framework-specific crashes, and stands out in automatic reproducibility, diversity in app categories and the total size of crash collection.

5.5.3 Evaluating Fix Templates on ReCBench

The objective of RQ3 is to assess the quality of the 17 fix templates yielded by `CraftDroid`. To that end, we first consider crash samples from *ReCBench* that are selected based on the below criteria:

1. We focus on crash samples that are relevant to the universe of crashes for which the fix templates were inferred. Concretely, we sample crashes from the 35 buckets yielded in Phase I.
2. Diversity over quantity. We select crashes to consider as many buckets as possible.
3. After a sample of each represented bucket is selected, we consider more samples corresponding to most represented buckets (in this case, *ActivityNotFoundException*-related crashes).
4. To avoid bias, we ensure that we consider crashes for which no fix was ever found after crash exploration.

Table 5.4: Patch Evaluation on RecBench

App Name	Bucket	Applied Template	Fix	Remark
AutoHome	8	LifeCycle Verifier	yes	
PI	8	LifeCycle Verifier	yes	
JadwalKA	12	Thread Finisher	yes	
Fruit Mahjong	12	Thread Finisher	yes	
Flashlight	1	Provider Checker	yes	
areain!	1	Provider Checker	yes	
WordPress	18	Range Checker	no	crash with succeeding API
Android Optimizer	18	Range Checker	yes	
Mine_mine	20	Emptiness Checker	yes	
SetCPU	20	Emptiness Checker	no	String null throw NPE
FingerWQ	23	Canvas Preconcator	no	app method non existing
BTCfx	15	Package Settler	yes	
MapCam	15	Package Settler	yes	
Baby Piano	2	Activity Resolver	yes	
GK in Gujarati	2	Activity Resolver	yes	
Reflection	2	Activity Resolver	yes	
UK Lotto	2	Activity Resolver	yes	
Agile Buddy	2	Activity Resolver	yes	
HiYou Park	33	Try-catcher	yes	
Sohu Weibo	33	Try-catcher	yes	

Due to time and repair execution constraints, we eventually selected 20 crashed apks as shown in Table 5.4. For the 20 crashed apks, 17 were fixed, meaning that the patched apk does not crash in same testing inputs. However, there are still 3 apks that did not pass the test. The original version of *WordPress* crashed because it attempted to get index on a substring that does not show in the target string. Although we managed to do a range check and prevented this inappropriate call, succeeding call to the same API again crashed the app. This indicates that the developers tend to make the same mistakes on the same API usage. The patched *SetCPU* fails because we are trying to check whether the string is empty by calling isEmpty() on the string before it is used to parse for int. However, the fix template did not anticipate that the string itself is never instantiated, so the NPE(*NullPointerException*) was thrown and still crashes the app. *FingerWQ* was not even successfully patched because Soot was not able to create the *Jimple* file of the targeted app code from the apk, so the patching process was not carried. Overall, the 20 crashed apks were intended to be patched thanks to 9 out of our 17 fix templates.

In Listing 5.3, we list 4 patches successfully generated and evaluated to be effective. Since we are using Soot to patch apks, the patches are in fact in *Jimple* code. However, for readability purpose, we illustrate in an equivalent *java* format.

The first app is *AutoHome*, it is a forum app for discussion and information sharing for car fans. On showing the dialog alerting no network connection is present, it encounters *WindowManager$BadTokenException*, it indicates that the activity where the dialog wants to show above has been destroyed. The correct way is to check the activity's lifecycle before calling the show() method, it is therefore fixed by our *Lifecycle Verifier* template. This exception exists in 4 buckets we collected in Section 5.4. The bug reasons as well as the correspondent fix template are also same for the buckets. The second app is a train tracking app which require location update information. However, since network is turned off for the emulators, requesting location updates from network will make the *IllegalArgumentException* being thrown. The correct way is to check the provider before requesting. This app is fixed by our *Provider Checker* template. The third app is a photography app that has over 100k installs from Google Play. We found that it crashes when it tries to bind to the app billing service. The root cause is that it did not set package name of the binder class for the intent instance. It is therefore fixed with correctly setting the package name, with our *Package*

```
//App: AutoHome (forum app for car fans)
//Exception: WindowManager$BadTokenException
//Applied template: Lifecycle Verifier
+ if (!activity.isFinishing){
        alertDialogBuilder.show();
+ }
//App: areain! (train tracking app)
//Exception: IllegalArgumentException
//Applied template: Provider Checker
+ if(locationManager.isProviderEnabled("network")){
     locationManager.requestLocationUpdates("network"
                                                  ,5000L,100.0F,this);
+ }
//App: MapCam (photography app)
//Exception: IllegalArgumentException
//Applied template: Package Settler
  Intent intent = new Intent("com.android.vending
                                  .billing.InAppBillingService.BIND");
+ intent.setPackage("com.android.vending");
  this.bindService(intent, serviceConnection,1);
//App: Fruit Mahjong (gaming app)
//Exception: IllegalArgumentException
//Applied template: Thread Finisher
-  thread.stop();
+ thread.interrupt();
```

Listing 5.3: Fix Examples for Crashed Apps.

Settler. The fourth app is *Fruit Mahjong*, a gaming app. It was fixed by replacing from call of stop() to call of interrupt() to thread to prevent *UnsupportedOperationException* being thrown.

Answer to RQ3: Our mined fix templates are in general efficient in patching crash-inducing bugs caused by wrong usage of framework APIs, although in minor cases they might cause regression.

5.6 Threats to Validity

External validity

Our findings may be biased due to the app dataset (of free apps) that we used for mining as well as the testing tools that were leveraged in `CraftDroid`. Nevertheless, we tried to mitigate these threats by considering the largest repository of Android apps available to researchers. We also tried three different test strategies to improve coverage.

Internal validity

Our method of localizing crash-inducing bugs includes some threats to validity. We utilize stack trace for fault localization, and assume that the crash-inducing bug arises in one of the app methods in the stack trace. However, it is possible that such bug can also reside in app methods that do not show in the stack trace. For example, the *NullPointerException* can be thrown because a field variable is not initialized during the instantiation of the type. In this case, the bug location is different from the crash location. However, we consider our work still valid since it has been reported by this empirical study [244] that 59% to 67% crash-inducing bugs actually reside in stack traces. Although it indicates that we might miss fixes during the mining process, it does not affect the validity of `CraftDroid`.

5.7 Related Work

Android app analysis has been a hot topic for many years. Different approaches, including both static analysis and dynamic analysis, have been proposed to tackle various issues in the mobile realm such as privacy leaks detection [23, 132], repackaged apps identification [144], etc. Our approach, targeting the runtime crashes of Android apps and their potential fixes, has adopted both static and dynamic analysis techniques.

Android Crash Analysis. The most closely related works to ours are by Tan et al. [223] and Fan et al. [56]. Tan et al. [223] present a benchmark of Android app crashes and their potential fixes to the community for exploring automated crash fixes. Their benchmark, namely DroixBench, which is built by mining open-source projects, contains only 24 samples, which are unfortunately not representative to real-world app crashes and fixes since over half of the thrown exceptions are *NullPointerException*. Similarly, Fan et al. [56] have also mined open-source project tracking systems for identifying reported crashes and associated fixes. Both of these two approaches leverage open-source projects to identify crash fixes. Our work, however, is different given that we attempt to identify crash fixes from closed-source apps. We thus complement the state-of-the-art with more dataset of crash fixes and further provide new means to harvest crash fixes in closed-source settings. The fact that a large majority of Android mobile apps are closed-source [11] suggests that `CraftDroid` has a high potential for researchers to mine ever more crash fixes than with the previous approaches.

State-of-the-art works are also interested in reproducing runtime crashes [39, 210, 249]. For example, Xuan et al. [249] leverage mutation testing to reproduce crashes while Soltani et al. [210] leveraging genetic algorithms on the inputs to reproduce crashes. Recently, Moran et al. [183] proposed CRASHSCOPE, a testing tool for automatically discovering crashes, saving log related information, and generating useful reports. Along with the tool, the authors released only 8 crashes from 20 Android apps together with reproducing scripts and detailed reports. Such a tool could, however, be integrated into `CraftDroid` to further enhance the crash exploration. Note that, in our work, we use the same test inputs to feed different lineage app versions in order to locate potential crash *fixes*.

Works that tackle the compatibility of Android apps [140, 141, 240] are also relevant to our study. Indeed, there are many reasons that may cause Android apps to be incompatible to some devices. However, the resulting consequence of incompatibilities is often the same: runtime crashes.

Android App Testing and Analysis. Several automated app testing approaches have been proposed to dynamically analyze Android apps [16, 19, 26, 41, 50, 106, 163, 169, 256]. For example, Mao et al. [169] have introduced an approach that combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation techniques to perform multi-objective automated testing for Android apps. Among the top 1,000 Google Play apps, Sapienz is able to find 558 unique, previously unknown crashes. As summarized in the systematic literature review in Chapter 3, there are more than 100 works proposed by the community to tackle the problem of automated app testing. Our approach is orthogonal to these approaches, where we can leverage more automated testing approaches to further pinpoint runtime crashes of Android apps.

Static analysis has been also a popular technique to dissect Android apps [135, 139, 171, 202]. For example, researchers have used static taint analysis to discovery privacy leaks in Android apps [23] and leveraged model checking techniques to verify Android apps in terms of their security properties [28]. In this work, we leverage basic static analysis techniques to identify crash fixes in Android apps. We believe that more advanced static analysis techniques could be leveraged to improve the accuracy of our crash fix identification approach.

5.8 Summary and Future Work

In this chapter, we target a new research direction attempting to mine Android crash fixes from Android market apps, which usually do not have an open change-tracking or issue-tracking system. We successfully generatred 17 fine-grained fix templates, which were evaluated to be effective in patching 17 out of 20 real-world crashed apps. `CraftDroid` can benefit the automatic program repair (APR) community [124, 127, 153–159] in establishing a new means to augment datasets. `CraftDroid` can also benefit the developer community in recommending effective patches to fix their crashed apps.

As future work, we plan to integrate more automated testing tools so as to enrich our set of fix templates. We also plan to implement analyzers to 1) scan user reviews to retrieve a more accurate set of apps to be tested for runtime crashes and 2) study the GUI changes to filter out false negatives. Finally, beyond the current manual abstraction methods, we aim to apply state-of-the-art automatic fix mining approaches to make our approach fully automatic.

6 Afuera: Automatically Documenting Android Framework APIs for Unchecked Exceptions

Unchecked exceptions are the main reason for framework-specific Android app crashes. In order to prevent developers from introducing crash-inducing bugs into their apps, we should warn them of API usages associated with unchecked exceptions. However, framework API references, and in particular the Android framework, are not providing enough information about unchecked exceptions. To address this issue in this chapter, we propose `Afuera`*, an automated tool that profiles Android framework APIs and provides information on when they can potentially trigger unchecked exceptions. With* `Afuera`*, we confirmed that* 26 739 *unique unchecked exception instances may be triggered by invoking* 5 467 *(24%) Android framework APIs.* `Afuera` *further analyzes the Android framework to inform about which parameter(s) of an API method can potentially be the cause of the triggering of an unchecked exception. Study results suggest that* `Afuera` *has perfect true positive rate, although affected by false negatives due to the limitation of state-of-the-art taint analysis techniques.*

This chapter is based on the work submitted for peer review to the below venue:

- Pingfan Kong, Li Li, Jordan Samhi, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Afuera: Automatically documenting android framework apis for unchecked exceptions. In *peer review for International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021

Contents

6.1 Overview **90**
6.2 Motivation and Background **91**
6.2.1 Motivation 91
6.2.2 Background on Android Framework-specific Exceptions 93
6.3 Study Design for `Afuera` **94**
6.3.1 Module I: Profile UE-API Methods 94
6.3.2 Module II: Pinpoint Parameters 97
6.4 Study Results **99**
6.4.1 Implementation 99
6.4.2 Key Characteristics of UE-API Methods 100
6.4.3 UE-APIs Usage in Real-world Android Apps 101
6.4.4 Effectiveness of `Afuera` Module II 103
6.5 Threats to Validity **104**
6.6 Related Work **104**
6.7 Summary **105**

6.1 Overview

Android leads today the worldwide market of operating systems (including desktop, tablet and mobile) [215]. However, Android app failures are still prevalent, despite generations of Android framework evolution. Consequently, these failures incur enormous debugging time from developers, which could have been devoted to providing better service to app users. In the attempt of mitigating these failures, researchers have worked on a chain of toolboxes for testing [42,218,254], reproducing [184, 270], locating [244] and fixing [223] these failures. Despite the practical value of these *curative* tools, the app failure *prevention* measures are still in urgent demand to reduce the likelihood of releasing failing apps. Unfortunately, app developers are limited in taking precautions against failure-inducing faults during development, since predicting run-time behavior is not an expertise of compile-time code checking tools. Indeed, having such precaution measures would ease the burden of the curative approaches. Although 100% bug free program is a myth, there are several contributions that can be made towards this direction.

Recently, Fan et al. [56] have pointed out in a large-scale study that framework-specific exceptions are the predominant root cause of app failures. These are exceptions thrown by Android framework APIs. An in-depth analysis of their study further shows that over 95% of these framework-specific exceptions are actually unchecked exceptions. Unlike checked exceptions that are checked at compile-time (forcing API method to handle the exception using a `try-catch` block or to declare it using the `throws` keyword), unchecked exceptions are not checked by compilers. Therefore, if the API method does not handle it with a try-catch block, when the exception is thrown at run-time, the app developer should have been prepared to catch it in the code. Otherwise, users will witness an app failure. By examining ReCBench (c.f Chapter 5), we have confirmed that unchecked exceptions are indeed the primary reason for app failures. Unfortunately, unchecked exceptions are not well-documented in the official Android API reference [72]: most of the time, the fact that an API can throw an unchecked exception is not explicitly specified. We, therefore, call for a comprehensive analysis of the framework to yield the complete list of framework APIs that are associated (directly or indirectly) with unchecked exceptions. Such documentation should also specify the types of the unchecked exceptions that may be thrown via these APIs.

Providing documentation and warnings about APIs throwing unchecked exceptions is an important first milestone. Developers, however, require additional support for understanding the conditions under which an unchecked exception will be thrown. Indeed, by thorough examination of the entire Android framework, we found that a significant number of unchecked exceptions are *IllegalArgumentException*. Debugging such exception is often not trivial. One or more arguments passed to the API could be erroneous. Identifying which argument is erroneous is a time and effort consuming task, especially given the Android framework's complexity and the huge number of APIs. Tool support to understand why a given API throws an unchecked exception is thus welcome. Note that a straightforward solution to fight against unchecked exception would consist in protecting each usage of APIs throwing unchecked exceptions with a try-catch block, but this solution is quite unrealistic and discouraged in general [193][1].

In this chapter, we opt for an approach relying on instrumentation and static analysis techniques. In particular, we propose `Afuera`, an automated tool for **A**ndroid **F**ramework **U**nchecked **E**xception a**R**gument **A**nalysis. `Afuera` is also the Spanish word meaning "outside". Metaphorically, we imply that our tool brings the study of failure reasons "outside" the app domain, towards the Android framework space that hosts the app code. `Afuera` is composed of two modules. The first one yields documentation to inform about which unchecked exception(s) could be thrown by a given API method. The second module taints the parameters of API methods to track their flows across the entire Android framework and to check if they reach the condition statement that guard the throw statements of unchecked exceptions. To perform this tracking, `Afuera` first instruments the Android

[1] Unchecked exceptions exist in Java to mark program error state that application code cannot reasonably be expected to recover from or by handling them in any way.

framework to then apply the popular taint-analysis tool FlowDroid [23]. Thanks to this second module, `Afuera` provides additional information about which parameters of an API method could be potentially the cause of the triggering of an unchecked exception.

In this chapter, we make the following contributions to the researcher and developer communities:

- We developed `Afuera`, a static and taint analysis based tool to yield better documentation about unchecked exceptions.

- With `Afuera`, we performed an exhaustive investigation on the entire Android framework. This investigation fills up the documentation's missing piece about which APIs may throw unchecked exceptions. We identified 5 467 APIs that throw unchecked exceptions. These framework APIs account for an astonishing 24% of all the Android framework APIs. We also identified 3 093 "throw statement" of unchecked exceptions accounting for 50 different types of unchecked exceptions. Overall, we count 26 739 pairs (m, ts), where ts is a throw statement of an unchecked exception that is propagated up to the API method m.

- We further applied `Afuera` to identify which API method parameters could be the cause of unchecked exceptions. We analyzed 50 randomly sampled APIs with unchecked exceptions. Evaluation results show that `Afuera` has perfect true positive rate. Although `Afuera` is affected by false positives due to state-of-the-art taint-analysis tools. We further explain the limitations and imply practical usages of `Afuera`.

- We have investigated the usage of such unchecked exception associated APIs in 200 real-world popular Android apps and their history releases. The study suggests that the usage of such APIs are fundamentally widespread. This finding provides insights on why Android app failures continually haunted app developers.

`Afuera` is publicly released at `afuera.github.io`, where the source code of the tool is available as well as the list of API methods and the corresponding unchecked exceptions they can throw.

The rest of chapter is organized as follows: Section 6.2 motivates our work and present some background information. Section 6.3 describes our tool supported approach named `Afuera`. In Section 6.4, we further address 3 research questions. In Section 6.5 and Section 6.6, we summarize the threats to the validity of this study and discuss related literature. Finally, we conclude this chapter in Section 6.7.

6.2 Motivation and Background

6.2.1 Motivation

The Android framework API *startActivityForResult* is a commonly used method. This API is available in several popular Android classes such as *Context*, *Activity*, *Service*, etc. When used in an *Activity*, it allows to start a new *Activity*. Note that an *Activity* is one of the 4 basic components of the Android framework. It is any screen that the user sees. However, junior developers often make the mistake in trying to start an *Activity* that does not exist in the hosting device. In this case, the *ActivityNotFoundException* will be thrown by the API and the app crashes if it is not explicitly caught.

First Motivation: Although the Android official reference [76] related to the API *startActivityForResult* indicates that the API can throw the exception *ActivityNotFoundException*, it is not written that the API can also throw 5 other unique exception types. Figure 6.1 draws the simplified method bodies of the API, the intermediary method *fm_1* on the call chain and the *Signaler* method. Note that a *Signaler* is the method from which the exception is initially signaled, e.g., by a *throw statement*. The *Signaler* method belongs to class *android.app.Instrumentation*. It is an important

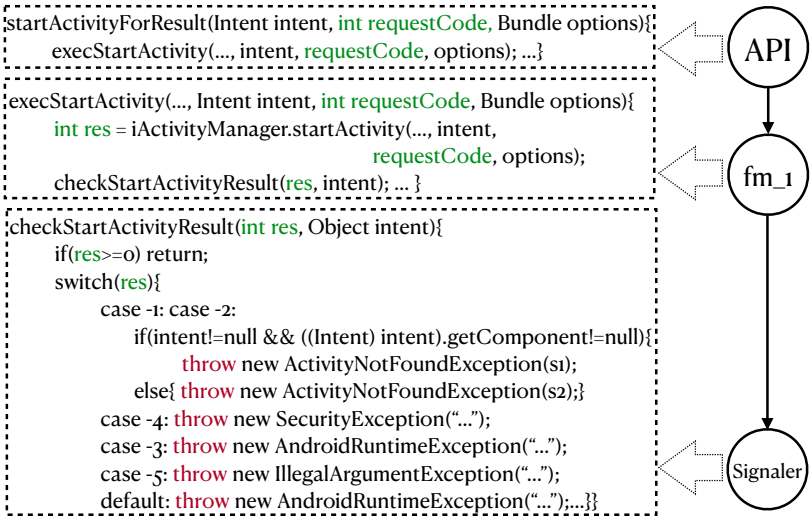


Figure 6.1: Motivating Example.

base class that allows the developer to monitor any interaction between the application and the framework. It also implements methods that will be invoked during the normal app run-time. The *Signaler* contains 6 throw statements in a switch-case block, each marks a unique error state of the app execution. Even for the well-known *ActivityNotFoundException*, there are two distinct throw statements. Developers should fix their apps accordingly based on these two different error messages *s1* and *s2*. This difference is also not clearly specified in the documentation. Moreover, there are 4 other throw statements worthy attention. For instance, *SecurityException* may be thrown with message *"Not allowed to start activity ..."*. The *AndroidRuntimeException* may be thrown with message *"FORWARD_RESULT_FLAG used while also requesting a result"*. Understanding and debugging for these exceptions are equally important, but comparatively harder since they are never mentioned in the documentation.

Moreover, Figure 6.1 is only a simplified version of the call chain from the API to this specific *Signaler*. Since the Android framework is complex, other signalers can exist and thus other exceptions could be thrown by this API without being documented. Actually, based on our evaluation results (c.f Section 6.4.3), we discovered that *NullPointerException* and *SuperNotCalledException* may also be thrown for this API, but from other *Signaler* methods. Due to the incomplete documentation, developers easily overlook these hazards which fail their apps during run-time.

Second Motivation: When faced to *ActivityNotFoundException*, developers are guided by the explicit name of the exception. Indeed, the target *Activity* is not found, and a normal developer should know that the issue is probably due to the *intent* parameter that specifies which *Activity* should be started. However, when faced to one of the other 4 thrown exceptions in the *Signaler*, developers may spend more time to understand the cause of the issue. Since these throw statements are within the switch-case block, the value of integer *res* (marked in green in Figure 6.1) controls which of the 4 exceptions should be thrown. The intuition is that since the second API argument *requestCode* is also an integer, *res* should have got its value from *requestCode*. Sadly, this is not correct. *res* is actually an internal flag that the framework holds to mark the **res**ult of executing the starting *Activity* task. In order to understand the process, the developers need to check the framework method *fm_1*. We can see that the value of *res* is the return value of invoking the *startActivity* method on an instance of *IActivityManager*. However, since all API arguments are passed in to execute this method, the developers further need to explore the method body of *IActivityManager.startActivity*, or even more framework methods to confirm. This task is time- and effort-consuming even for experienced developers.

Hint: First, the Android official documentation may miss many exceptions that are triggered during run-time. Second, when analyzing why these exceptions are thrown, the developers need to examine many framework methods to confirm which erroneous parameter(s) have led to the exception being triggered. This task is resource-intensive. In this work, we aim at enhancing API documentation (w.r.t. unchecked exceptions) and at providing a necessary building block for tool support to facilitate developers' debugging tasks.

6.2.2 Background on Android Framework-specific Exceptions

The Android software stack is composed of several layers on top of a Linux Kernel[2]. To take advantage of the various features provided by the Android platform, app developers interact with this Android software stack mainly via the Java API framework. Most of the Android apps are thus implemented in Java[3]. In its version level 19, the Java API framework contains 5 217 classes, with 22 714 API methods.

Java being central in Android app development, the Android framework shares the exception mechanism [193] of Java. Figure 6.2 presents the class hierarchy (based on inheritance links) of *Throwable* objects in Java. Any *Throwable* object can be thrown by either the virtual machine or by using the Java keyword *throw* during run-time. Figure 6.2 further lists some important sub-classes of *Throwable*. As specified by the Java documentation [193], types that sub-class *RuntimeException* and *Error* (e.g., *AssertaionError* and *ActivityNotFoundException*) are called **unchecked exceptions**. They are marked in bold font. As the name suggests, these exceptions are not checked by compilers during compiler-time. Developers subsequently may overlook these exceptions since they are not warned during development. The motivating example in Section 6.2.1 details such an exception type. The **checked exceptions**, on the contrary, are all instances of the *Throwable* class that are not sub-classes of *RuntimeException* or *Error*. The compiler force checking during compile time. Apps that do not *specify* or *catch* these exceptions will not compile successfully.

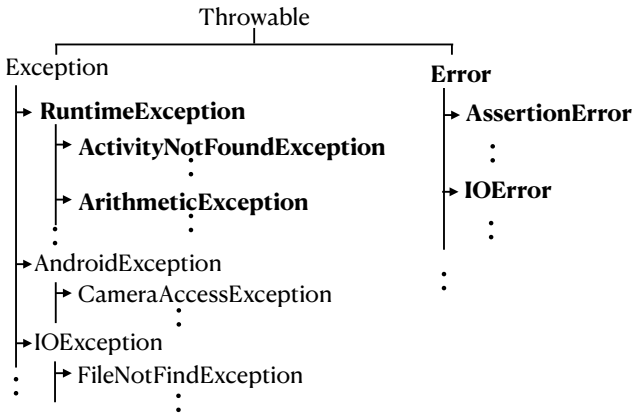


Figure 6.2: Java *Throwable* Type and its Sub-classes.

Figure 6.3 shows a concrete example of code with checked exception. The first code snippet does not compile because the method `m_1` uses the constructor method `new FileReader` which can throw a checked exception `FileNotFoundException`. To fix this program, the developer has two options: either specify the list of exceptions using `throws` (we opt for this option in Figure 6.3), or use a try-catch block (an example of this block given in Figure 6.6a).

Figure 6.4 shows a concrete example of code with unchecked exception. In this case, there is a division by zero in the method `m_2`, which at run-time throws the unchecked exception `ArithmeticException`

[2] `https://developer.android.com/guide/platform`
[3] Kotlin is promoted as an alternative for Android app development in recent years. It is fully inter-operative with Java.

Do not compile

```
m_1(String path) {
  FileReader f = new FileReader(path);
  …}
```

possible correction

Compile

```
m_1(String path) throws IOException{
  FileReader f = new FileReader(path);
  …}
```

`FileReader` can throw a checked exception *`FileNotFoundException`*

Figure 6.3: Checked Exception

and makes the program fails. Note that even if the division can throw an exception, the program in Figure 6.4 can still compile: no additional code is required to protect the "division", simply because the thrown exception is an unchecked exception.

```
m_2() {
  int x = 0; int y= 10;
  int z = y/x;
  …}
```

The program Compile even if at runtime the unchecked exception `ArithmeticException` is thrown.
No additional code is required to protect against this unchecked exception

Figure 6.4: Unchecked Exception

Studies have shown that the unchecked exceptions cause substantially more app failures than the checked exceptions. For instance, by investigating ReCBench in Chapter 5, we have shown that more than 95% of app failures result from unchecked exceptions. We postulate that this high ratio of unchecked exceptions is explained by the fact that developers are unaware of unchecked exceptions, i.e., they do not know whether a given API method can throw or not an unchecked exception that potentially leads to app failure. In the remainder of this chapter, APIs that may potentially throw unchecked exceptions are referred to as **UE-API methods**.

6.3 Study Design for `Afuera`

`Afuera` aims at providing information to developers about UE-API methods, the unchecked exceptions they can potentially throw, and the parameters of these UE-API methods that could be the cause of the triggering of the unchecked exceptions. `Afuera` is composed of two modules as depicted in Figure 6.5. In Module I, we statically analyze the entire Android framework for identifying the UE-API methods, i.e., the API methods that may potentially throw unchecked exceptions. Additional information about the unchecked exceptions, such as the location of their throw statements, is also yielded. Module II instruments all UE-API methods found in Module I and applies a taint analysis tool to statically compute the flows of the parameters of these UE-API methods and check if they reach the condition statement (e.g., if statement) that guards the throw statements of unchecked exceptions. The assumption is that when an Android app fails because of an unchecked exception thrown by a given API method, the debugging effort of its developers will be reduced if they know which parameter(s) of the API method is linked to the throw statement of the unchecked exception.

6.3.1 Module I: Profile UE-API Methods

In this section, we present how to profile the Android framework UE-API methods from the Android framework class files. Concretely, the output of this step is a list of pairs (m, e) where m is a UE-API method that can potentially throw the unchecked exception e. Together with e, additional information is provided about the location in the framework where the exception originates. Module I consists of 4 steps: **Identify**, **Build**, **Locate** and **Propagate**.

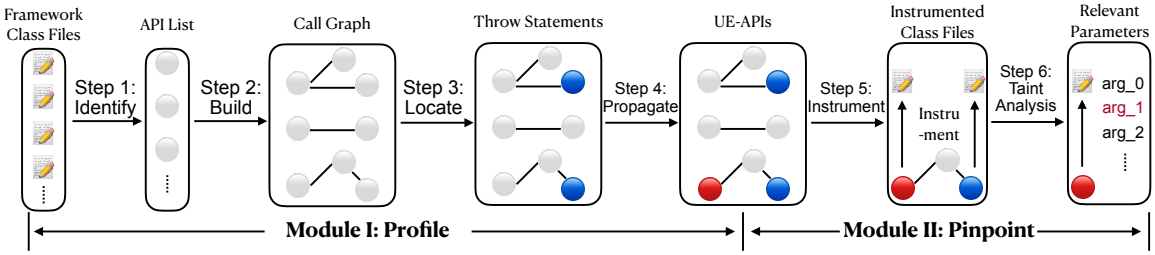


Figure 6.5: Workflow of `Afuera`.

Step 1. Identify Framework APIs: The goal of this step is to build the list of all the Java API methods present in the Android framework. To start the analysis, we first need to retrieve the framework class files. Android is an open framework with the framework source code available to everyone. However, we choose to analyze the compiled class files over the source code. This choice allows us to avoid analyzing unit testing code and potentially dead code that have not been used by the framework if any. We use a small trick to retrieve the framework class files directly from an Android emulator instance. We apply the command *"adb pull /system/framework/framework.jar/"* from the emulator's hosting server to retrieve the compressed framework class file *framework.jar*. The command *"adb"* (Android Debug Bridge) [73] is a versatile command-line tool that allows the host server to exchange information with the emulator instance. After decompressing the file, we obtain the *framework.apk* file. It is actually another compressed package containing all the framework class files. Different from Java bytecode class files that run on JVM (Java Virtual Machine), the Android class files are in Android-specific Dalvik bytecode format that run on DVM (Dalvik Virtual Machine)[4]. Therefore, we use the Dexpler [30] tool to translate Dalvik bytecode classes to the Java bytecode classes. We finally obtain the *"classes.jar"* file.

We explain now how we retrieve the API list from the *"classes.jar"* file. First, we use the *"jar -tf classes.jar"* [194] command to obtain the jar contents, including the list of class files. Second, for each class X with a class file path *x-path*, we use the *"javap -cp classes.jar x-path"* command to get members of X and collect only non-private methods. To ensure that the methods can be invoked from application code, we require each method to be not *abstract*. Also, to guarantee that the method is visible from application code, we only keep *public* methods, and *protected* methods whose declaring classes are not *final*, i.e., can be sub-classed. Finally, these methods of all framework classes will be our list of framework APIs. Note that we consider each API method as unique even in the case of inheritance.

Step 2. Build Call Graphs: The goal of this step is to compute Call Graphs to facilitate the analysis performed in Step 3 and Step 4. In Step 2, we rely on Soot [22], a comprehensive toolbox for static analysis of Java programs. As inputs, Soot requires a *jar* file and list of entry points. In our case, the jar file is the *classes.jar* that is extracted in Step 1. The list of entry points is the list of API methods identified in Step 1. Next, we use Soot's built-in Class Hierarchy Analysis (CHA) algorithm implementation to build the call graphs from each entry point. We use CHA to include as many call edges as possible. Note that the Call Graphs built from each entry point may or may not be connected, hence the plural form of Call Graphs.

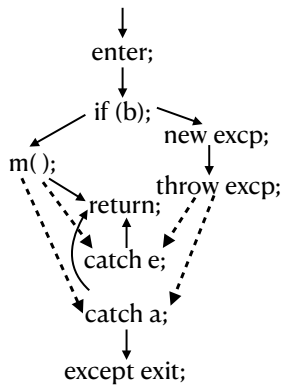
Step 3. Locate Throw Statements: Based on the Call Graphs built from the last step, the goal is now to locate the throw statements whose thrown types are unchecked exceptions. To that end, first, we traverse all the methods (nodes) in the Call Graphs. Then for each method, we compute its Control Flow Graph (CFG) with Soot. CFG is a graphical representation of the control flow of a program method (function) during the program execution. It eases the intra-procedural analysis of methods in languages such as Java. For example, Figure 6.6b is a CFG of method *m_ECF* (Figure 6.6a). We

[4]Since Android API level 21, Android Runtime (ART) fully replaced DVM. However, it continues to use the same input bytecode.

simply consider each node of the CFG and check if it is an instance of a throw statement. After checking Figure 6.6b, we confirm that unit *throw excp* is a throw statement. Then, we further extract the thrown type and check if it sub-classes *java.lang.Error* or *java.lang.RuntimeException* (c.f Section 6.2.2 where we explain that unchecked exceptions are sub-types of these two classes). If the answer is positive, we keep track of the throw statement for propagation analysis in Step 4. For example, since *AssertionError* sub-classes *Error*, *throw excp* is tracked. In Figure 6.5, the methods that contain at least one such throw statements are marked in blue. Recall that these methods are called *Signalers* (c.f Section 6.2.1).

```
void m_ECF(boolean b){
    try{
        if (b){
            m( );
        else{
            excp = new AssertionError(..);
            throw excp;
    }catch(Error e){
        ...
    }catch(AndroidException a){
        ...}}
```

a Method *m_ECF*.



b CFG of *m_ECF*.

Figure 6.6: Exceptional Control Flow.

Step 4. Propagate Unchecked Exceptions:

The objective of Step 4 is to determine if a given unchecked exception (i.e., the located throw statements) can be propagated up to an API method via a chain of method calls. One difficulty is to check if the unchecked exception is not caught by an "intermediary" method in the chain before reaching the API (actually, as shown in Section 6.4, it is not rare, although unexpectedly, to see an unchecked exception "caught in the framework" before reaching an API method).

In order to study if an unchecked exception is caught in these intermediary framework methods, we need to focus on the Exceptional Control Flows (ECF)[5] of any CFG. Figure 6.6a shows the simplified source code of a framework method *m_ECF*, with its CFG depicted in Figure 6.6b. The solid lines in Figure 6.6b represent normal control edges. However, there are also exceptional control edges (marked in dashed lines) from method invocation *m()* and *throw excp* to the two exception catch blocks. Hereafter in this chapter, these blocks will be called *traps*. If *b* is false, *throw excp* is executed. Based on Figure 6.2, the thrown *AssertionError excp* sub-classes the *Error* class. Therefore, this exception will be successfully caught by the first trap. In this case, the method *m_ECF* can finish normally. If method *m()* is executed, it may throw an unchecked exception not known during compile-time. For example, it may throw an *ActivityNotFoundException*. Based on Figure 6.2, since it sub-classes neither of the two trapped exception types, the method *m_ECF* will terminate with *except exit*. In such a case, this uncaught exception will be thrown to the caller to handle. To summarize, the unchecked exception can reach the UE-API method if all intermediary methods on the call chain failed to catch it with the traps.

Algorithm 2 better formalizes Step 4. The input of this algorithm is any throw statement *ts* identified in Step 3 and the exception type *excp* it has thrown. First, we set *ts* to be the analysis *target*. Then we retrieve the ordered list of exception *traps* by analyzing the CFG that *target* belongs to. We then compare the exception type *excp* against each *trap*, if *excp* is sub-classing one of the trapped exception type, the algorithm finishes. If not, however, it means that the thrown exception is not

[5]ECFs allow the program to make jumps that violate the usual call/return stack discipline in response to error conditions [36]. In our case, these conditions are unchecked exceptions. The ECFs are already included in the CFG constructed by Soot.

Data: *ts*: **t**hrow **s**tatement
Data: *excp*: thrown **exce**ption type
Result: *S*: **S**et of UE-API methods

```
1: target ← ts;
2: Traps ← CFG.getTraps(target);
3: for trap ∈ Trap do
4:     if excp instanceof trap then
5:         caught ← true;
6:         break;
7:     end if
8: end for
9: if caught then
10:    return;
11: end if
12: method ← getContainingMethod(target);
13: invs ← CG.getInvocations(method);
14: for inv ∈ invs do
15:    target ← inv;
16:    goto line 2;
17: end for
18: if method ∈ API then
19:    S.add(method);
20: end if
21: return;
```

Algorithm 2: Propagation Algorithm for One Throw Statement

caught internally and the containing *method* may throw the unchecked exception *excp*. Next, we traverse all the Call Graphs retrieved from Step 2, to find *invs*: the set of *inv*ocation statement*s* targeting *method*. Recursively, we repeat the algorithm on each *inv*. Finally, if the *method* is an API, it will be confirmed as a UE-API method. In Figure 6.5, these UE-API methods are marked in red. Based on Algorithm 2, we can conclude that not all throw statements can be successfully propagated to reach a UE-API method, since the unchecked exception can be caught at any level of the Call Graph during the propagation.

6.3.2 Module II: Pinpoint Parameters

In this sub-section, we present how to pinpoint the parameters of UE-API methods that might be linked to an unchecked exception. Module II consists of two steps. It takes as input all UE-API methods as found by Module I, and first performs **Instrumentation** and then **Taint Analysis** to pinpoint the UE-API method's parameter(s) that may be linked to the unchecked exception.

This step can be seen as a pre-processing step in order to perform Step 6. The taint analysis presented in Step 6 is performed by using FlowDroid [23], which is able to detect information flows in the Android apps by precisely modeling the Android components life cycles. In FlowDroid, an information flow is defined as a path in the code from a *source* method to a *sink* method. Both source and sink methods are input parameters of FlowDroid. Our idea is to consider the parameters of a UE-API method as sources and the throw statement in the Signaler method (c.f. Section 6.2.1) as sink. In this way, FlowDroid will be able to inform us about the existence of a path between the source and sink, and thus to indicate that the UE-API method's parameter(s) may be linked to the unchecked exception. Unfortunately, in FlowDroid, a parameter of a method cannot be a source. To overcome this limitation, `Afuera` instruments the API framework in Step 5 by adding dummy methods that will serve as sources and sinks.

Step 5. Instrumentation of APIs and Signalers: Let us consider a pair of a UE-API method and a throw statement of an unchecked exception as yielded by Step 4. The purpose of Step 5 is to insert dummy source and sink methods into the framework to facilitate the taint analysis in step 6.

We detail the instrumentation on a concrete example of UE-API method and *Signaler*. We consider again the example presented in Figure 6.1 (c.f Section 6.2.1) involving the *startActivityForResult* UE-API method.

Listing 6.1 illustrates how `Afuera` instruments the UE-API method and its declaring class. First, since there are three parameters for this *startActivityForResult* UE-API method, `Afuera` inserts three dummy source methods (Lines 9-14) into the class *Activity*, one for each parameter. Each dummy source method takes one of the parameters as argument and returns the argument itself. For example, method *dummySourceArg0* takes the first parameter *intent* as argument and immediately returns the same *intent*. Then `Afuera` inserts the invocation of each of these three dummy source methods at the beginning of the UE-API method body (Lines 4-6) before the original code. Note that with this instrumentation, the values of the three parameters during run-time will not be affected. Therefore, the instrumentation does not affect the originally implemented logic of the UE-API method. Thanks to this instrumentation, the three parameters are now in a source method and they will be "tainted" by FlowDroid.

```
1  package android.app;
2  public class Activity extends ...{
3    public void startActivityForResult(Intent intent, int requestCode, Bundle options){
4  +          intent = dummySourceArg0(intent);
5  +          requestCode= dummySourceArg1(requestCode);
6  +          options = dummySourceArg2(options);
7            .../*original code*/
8            }
9  + private Intent dummySourceArg0(Intent intent){
10 +          return intent;}
11 + private int dummySourceArg1(int requestCode){
12 +         return requestCode;}
13 + private int dummySourceArg2(Bundle options){
14 +         return options;}
15 ...}
```

Listing 6.1: Instrumentation of the UE-API Method. The lines starting with the symbol + are inserted thanks to the instrumentation.

Listing 6.2 shows how to instrument the *Signaler* method and its declaring class. Coincidentally, the name of the *Signaler*'s declaring class is *Instrumentation*, not to be confused with this instrumentation step of `Afuera` we are explaining. Since there are multiple throw statements, we assume that the throw statement for the *IllegalArgumentException* is the targeted unchecked exception. Other throw instances are be instrumented similarly. First, `Afuera` inserts a dummy sink method (Line 11) that takes as parameter the condition variable *res*. Since the throw statement is within a switch-case block, the condition only contains a single integer *res*. If, however, the throw statement would have been guarded by an if-clause, each variable that composes the boolean expression would have been a parameter of the dummy sink method. Note that since we are only creating an information sink, the method body can be empty. Next, we insert the invocation to this dummy sink method immediately preceding the throw statement (Line 7). Similar to Listing 6.1, the instrumentation does not change the original internal logic of the *Signaler* method. By doing this, we have created a dummy information sink to facilitate the next step.

```
1  package android.app;
2  public class Instrumentation {
3  /*package*/ static void checkStartActivityResult(int res, Object intent) {...
4    switch (res) {
5      case −5:
6  +          dummySink(res);
7      throw new IllegalArgumentException(...);
8      case −4: throw new SecurityException(...);
9      ...}}
```

```
10 + static void dummySink(int res){ }
11 ...}
```

Listing 6.2: Instrumentation of the Signaler Method.

Step 6. Taint Analysis: The final step is to run FlowDroid [23] on the instrumented class files to find the links between the UE-API method's parameter(s) and the condition guarding the throw statement. As already explained, the taint analysis tool FlowDroid [23] can only find information flow between two method invocations (source and sink). Therefore, we use the instrumented dummy methods in Step 5 to taint the parameters of the UE-API methods. In practice, we run FlowDroid for each pair (m, ts) where m is a UE-API method and ts a throw statement related to m. For each run, three parameters need to be configured.

1. A set of class files, which is the instrumented *classes.jar* file.
2. A set of source methods and sink methods. In this case, we provide the set of dummy source and sink methods inserted thanks to the instrumentation in Step 5. We only consider the instrumented methods related to m and ts.
3. A set of entry points where the analysis starts. We provide m (i.e., the UE-API method) as entry point.

The outcome of the analysis is a set of paths from a dummy source method to the dummy sink method. Each path indicates that the dummy source has information flow towards the dummy sink, and thus, that the argument of the UE-API method flows until the guard condition of the throw statement. Our assumption is that this information is valuable when developers try to debug an Android app failing because of an unchecked exception.

6.4 Study Results

Our assessment addresses the following research questions:

- **RQ1:** What are the key characteristics of UE-API methods. We divide this RQ into several sub-RQs that are detailed in Section 6.4.2.
- **RQ2:** How widely are UE-APIs used in real-world Android apps?
- **RQ3:** What is the accuracy of `Afuera` in identifying UE-API method parameters that relate to the unchecked exceptions?

Before answering these RQs, we provide details on the implementation that we made towards assessing the `Afuera` approach.

6.4.1 Implementation

We rely on the Soot framework [22] to profile the Android framework UE-API methods (Steps 1-4) and the instrumentation (Step 5). We use FlowDroid [23] to perform the taint-analysis (Step 6). To perform our experiments, we considered the Android framework API level 19, whose last revision was released in year 2013. We did not select newest API levels at the time of the study, since we want to learn the developer usage evolution over the years as detailed in Section 6.4.3. We do not expect any issue of applying `Afuera` on newer API versions.

We retrieve the framework class files from a Nexus 5X emulator instance. The total analysis time for Module I (c.f Section 6.3.1) finishes within 10 minutes. This is substantially faster than collecting unique framework-specific exception instances by applying testing tools on Android apps [56] which might take weeks.

6.4.2 Key Characteristics of UE-API Methods

After applying `Afuera`, we are able to locate 2 218 signalers, i.e., methods which contain at least one statement that throws an unchecked exception. In total, these signalers contain 3 093 throw statements involving unchecked exceptions. We identified 50 types of unchecked exception.

RQ1.1: What are the most prevalent type of unchecked exceptions? Figure 6.7(a) shows the number of signalers according to unchecked exception types. Note that one signaler may throw more than one type of unchecked exception and hence counted more than once. Due to space limitation, only the top 10 is presented. In addition, we truncate exception type name (e.g., use *NullPointer* to replace *java.lang.NullPointerException*). By far, the most prevalent unchecked exception type is *IllegalArgumentException*. This well-motivates the creation of a tool such as `Afuera` since issues related to "arguments" seem to be the most common reason for unchecked exception triggering. The second most common unchecked exception type is *RuntimeException*. It is the super type of a group of unchecked exceptions, describing general problems happening during app runtime. The third most frequent unchecked exception type is *IllegalStateExeption*, indicating that the app execution has entered into an illegal (unexpected) situation.

RQ1.2: From where do the unchecked exceptions originate? Figure 6.7(b) shows the number of signalers distributed over the Android framework packages. Again, names of packages are truncated (e.g., *app* instead of *android.app*). In total, 81 packages contain at least one signaler method. The top 10 packages have small differences in number, but we can notice that several packages related to UI are present (e.g., *android.view*, *android.widget*, *android.renderscript*, *android.graphics*). The *android.app* package encapsulates the overall Android application model and contains most signaler methods.

RQ1.3: Where do the unchecked exceptions exit the framework? Next, we consider the output of Step 4 in `Afuera` to identify the UE-API methods. We found 26 739 unique pairs (m, ts) where ts is a statement that throws an unchecked exception which reaches the UE-API method m. Overall, we count 5 467 UE-API methods. These UE-API methods take up over 24% of all framework APIs. Figure 6.7(c) shows the number of API methods associated with unchecked exception types. Note that one API may throw several unchecked exception instances. To our attention, the order of the top 4 most recurrent unchecked exception types are the same as in Figure 6.7(a). It again confirmes the prevalence of *IllegalArgumentException* and the importance of proposing `Afuera`. Figure 6.7(d) describes the number of UE-APIs in each Android package. In total 94 packages contain UE-APIs. An interesting finding is that the 3 of the top 4 packages *android.widget*, *android.view* and *android.renderscript* are all for the purpose of User Interaction (UI) management.

RQ1.4: Are unchecked exceptions handled in the framework? Not all thrown unchecked exceptions reach a UE-API method because they can be "trapped" in the framework (as explained in Step 4 of `Afuera`). While performing Step 4, we found that 10 138 unchecked exceptions were trapped (c.f Line 5 of Algorithm 2) by intermediary framework methods.

RQ1.5: Are UE-API methods well-documented? We check if the official Android API reference provides information about unchecked exceptions that could be thrown by API methods. To that end, we randomly pick 30 UE-API methods and check their references (i.e., documentation). We confirm that **none** of the 30 sampled UE-APIs' documentation lists unchecked exceptions. We made publicly available[6] the list of all UE-API methods with their corresponding unchecked exception types. This information can be used by developers when they are developing their apps.

Answer to RQ1: UE-API methods exist in large numbers in the Android framework. We identified 5 467 UE-APIs, taking over 24% of all Android framework APIs. Also, 3 of the top 4 packages with most UE-API methods are for UI interaction. The most common unchecked

[6] `https://afuera.github.io`

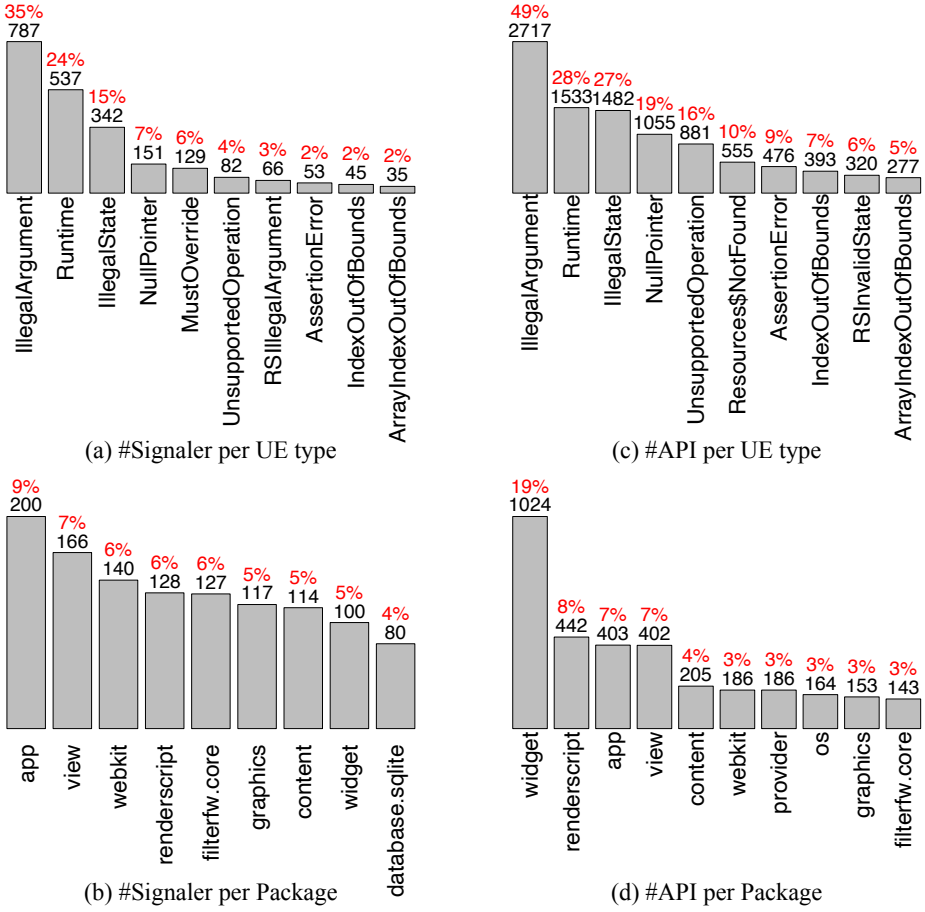


Figure 6.7: Signalers and UE-API methods Statistics.

exception type is the *IllegalArgumentException* (49% of UE-API methods throw this exception). This suggests that the UE-API method's parameter(s) should be the first place to look at when debugging for such unchecked exceptions.

6.4.3 UE-APIs Usage in Real-world Android Apps

To answer RQ2, we first collect the list of all Android apps present in the AndroZoo dataset [11] that have been crawled from Google Play [79]. Then we use the Python based crawler named *Google-Play-Scraper* [178] to collect meta data on app downloads counts. Finally, we randomly selected and downloaded from AndroZoo 200 apps with at least 100 million downloads. For each of these 200 apps, we take care of selecting its most recent apk version.

We use the Soot toolbox to analyze our dataset of 200 apks. More specifically, for each app, we count (1) the number of UE-API method usages, and (2) the number of overall framework API method usages (including UE-API methods). On average, we count 9 367 API usages and 2 895 UE-API usages per app in our dataset. Then, for each app, we group UE-API methods according to the unchecked exception types that they may throw. For instance, we can obtain the group of UE-API methods that can throw *IllegalArgumentException* in a given app. For each group g of each app, we compute the ratio of UE-API methods in g to all UE-APIs. Figure 6.8 (left) presents the distribution of this ratio over the 200 apps in our dataset and for each exception types already listed in Figure 6.7(c). First, we notice that the box plots are all relatively flattened. This result suggests that apps tend to have similar ratio for each unchecked exception type. Second, the "*IllegalArgumentException*

cluster" is by far the cluster with the highest ratio. On average, 65% of UE-API methods can throw *IllegalArgumentException*. The "*IllegalStateException* cluster" is also high, on average over 50% of UE-API methods can throw this unchecked exception. Note that each UE-API may throw multiple unchecked exceptions. Therefore, the added sum may be more than 100% for each app. Similarly, Figure 6.8 (right) presents the box plots over each Android package in the same order as in Figure 6.7(d). The usage ratio of UE-APIs over packages do not diverge much over the 200 apps, except for the packages *widget*, *view* and *os*. Interestingly, the substantially high overall usage ratio of UE-APIs in package *android.view* draws our attention. As evidenced in Android official API reference [72], classes in this package expose basic user interfaces that handle screen layout and interaction with the user.

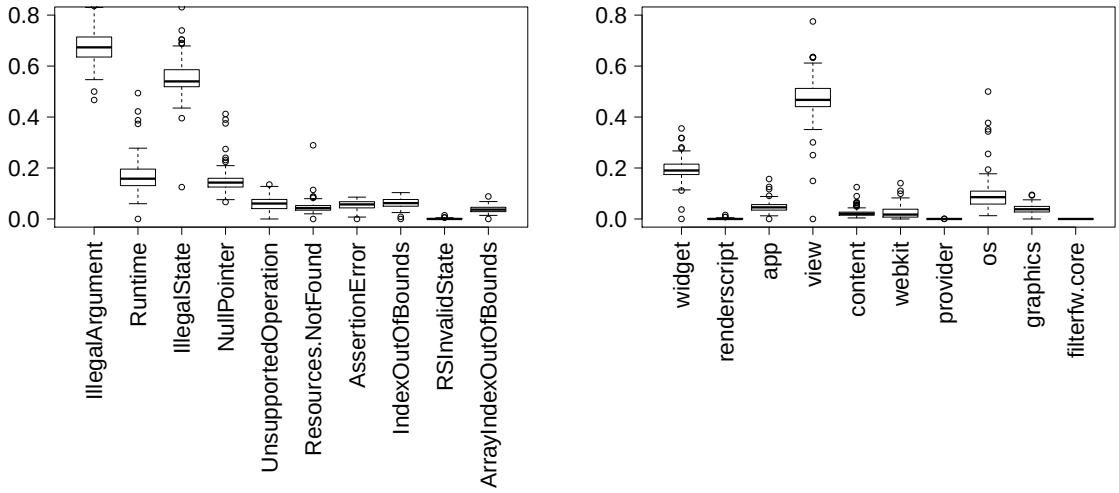


Figure 6.8: Distribution of the UE-API Usages per UE type (left) and per package (right)

UE-API method usages evolution: Since Android apps adapt fast, we are interested in the evolution of the number of UE-API method usages over years. To that end, we retrieved for each app, all its versions present in AndroZoo. Figure 6.9 shows the distribution of the ratio (#UE-API usages/#API usages) for each year. We observe that the UE-API method usage ratio, in average, is rather stable (very slightly decreasing), i.e., roughly over 30% for each year. Note that in the Android Java API framework that we extracted from the selected Android device (i.e., framework.jar), there are 22 714 API methods overall including 5 467 UE-API methods, leading to a 24% ratio of UE-API methods. The small difference between these two ratios (24% vs. 30%) suggests that UE-API methods have been indeed frequently leveraged by Android apps.

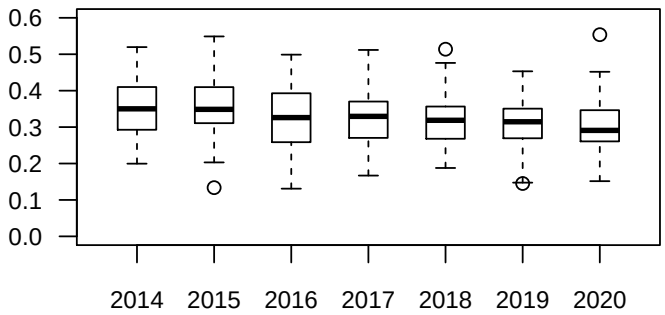


Figure 6.9: UE-API Usage Yearly Evolution.

Answer to RQ2: Developers tend to significantly use UE-API methods. Their usage in Android apps accounts for around 30% of all framework API method used (ratio stable over time).

6.4.4 Effectiveness of `Afuera` Module II

We remind that our assumption is that if a parameter of UE-API method is related to the guarding condition of an unchecked exception that leads to app failure, the likelihood that this parameter is responsible for the fault leading to the exception triggering is high. By using static taint analysis, `Afuera` is able to check if the parameter of UE-API method is actually related to the guarding condition of throw statement of an unchecked exception. However, static analysis is often subject to imprecision. In this RQ3, we evaluate the effectiveness of `Afuera` in performing the above-mentioned taint analysis task. To that end, we randomly select 50 UE-API methods and compare the result of `Afuera` analysis against our manual analysis.

Overall, we have found 5 467 UE-API methods in the Android framework. Among them, 979 have no parameter. We suppose that the unchecked exceptions associated with these UE-APIs are not parameter-dependent. They are often associated with the erroneous program state caused by other API usages in the application space. We leave the study of this supposition for future work. From the 4 488 UE-API methods with at least one parameter, we randomly select 50. If there are multiple unchecked exceptions thrown by the selected UE-API method, we randomly pick one to analyze.

To perform our manual investigation, we inspect the AOSP source code (i.e., the Android framework open source code) with the help of Integrated Development Environments (IDE) such as AndroidStudio [80] or Eclipse [59]. These IDEs help us to track object values throughout the call graph. We use both forward tracking, i.e., tracking taints from parameters, and backward tracking, i.e., tracking taints from throw statement condition objects, to verify the link. Note that this manual verification procedure is widely acknowledged to be effort- and time-consuming [89, 243].

Table 6.1 lists the evaluation results of `Afuera` and compare it with our manual investigation that we consider being the ground truth. In total 16 parameters from 16 UE-API methods are asserted to be relevant to the correspondent unchecked exception condition. To our satisfaction, these assertions are verified to be 100% correct, and thus `Afuera` is highly precise. However, there are 11 parameters wrongly asserted by `Afuera` as irrelevant (false negative results) We then carefully examined the taint-tracking process of FlowDroid and summarized the failure reasons. We found that these false negatives are all due to the common drawback of taint analyzers' incapability of tainting lookup tables. Indeed, as pointed out by Cavallaro et al. [38], such drawback is common among taint analyzers. Due to this limitation, taints in lookup tables are disrupted.

Table 6.1: Confusion Matrix of Evaluation Results.

	truely relevant	truely irrelevant	Total
Assert relevant	16	0	16
Assert irrelevant	11	23	34
Total	27	23	50

To better understand this limitation, Listing 6.3 gives a concrete example of one of the false negative results. Note that the methods in this listing are simplified for clearer illustration. The *validCycle* method (Lines 2-7) is a signaler in package *android.renderscript*. At line 6, an *RSInvalidStateException* is thrown, complaining against an unexpected loop in a node graph. However, FlowDroid asserts that only parameter *original* is related to the condition at Line 5, while totally omitting parameter *target*. The disrupted taint occurs at the method *findNode* (lines 9-11). More specifically, this method compares the *kernelID* between the target node and each node in the graph and returns the node with same *kernelID*. However, this taint is disrupted since there is no tainted value assignment at

line 10. This limitation of FlowDroid deters `Afuera` in spotting all parameters that are related to unchecked exception.

```
1  /* methods belonging to package android.renderscript */
2  private void validateCycle(Node target, Node original) {
3     KernelID kt = getID(target);
4     Node nt = findNode(kt);
5     if (nt.equals(original))
6        throw new RSInvalidStateException("Loops in group not allowed.");
7  }
8  private Node findNode(KernelID k) {
9     for(n : this.mNodes)
10       if (k == n.id) return n;
11 }
```

Listing 6.3: Failed Taint Tracking Example

Answer to RQ3: Manual verification suggests is highly precise when it pinpoints parameters of UE-API methods that are related to unchecked exceptions. However, `Afuera` fails to identify some of the parameters due to limitations in the taint analysis of the FlowDroid state of the art tool. Overall, `Afuera` has an accuracy of $78\% = (16 + 23)/50^a$.

[a] We compute the accuracy with the following formula $(TP + TN)/total$

6.5 Threats to Validity

Our approach only focuses on unchecked exceptions that are explicitly thrown by app code. We do not include those thrown by the Android Runtime. Unchecked exceptions like *NullPointerException* and *ArrayIndexOutOfBoundsException* can be thrown implicitly by the Android Runtime. This is a non-trivial task requiring a series of efforts [35, 108, 227].

Our analysis is focused on Android classes in the Android Open Source Project (AOSP). Therefore, we do not consider some other common packages related to Android apps. In particular, we do not consider *com.google.*∗, the Google Mobile Service (GMS) package. GMS is a commercial software package owned by Google and is not part of the AOSP. Also, Java runtime (JRE) specific packages are not included, since we focus on Android classes. Finally, we exclude third-party packages like *org.json* and *org.w3c.dom* that are included in AOSP. Since these packages are ruled out, the propagation in Step 4 might be incomplete.

6.6 Related Work

Android Framework-specific Failures: Several approaches have been proposed in the literature to address this Android framework-specific app failures. These approaches traditional debugging tasks such as testing [16,19,26,41,42,106,163,169,218,254,256], reproduction [184,270], bug localization [244] and bug fixing [223]. Several works that reason about the failure-inducing bugs are closely related to this chapter. Fan et al. [56, 217] performed a large-scale analysis on framework-specific exceptions in Android apps. They have systematically collected failure cases by testing real-world apps and by crawling issue-tracking systems online. However, the collected cases only reveal a portion of framework-specific exceptions. While our exhaustive investigation can find the complete set of all UE-APIs. Moreover, the authors contributed to distill 11 common fault types by devoting 6 person-months. However, `Afuera` is able to accelerate this procedure by pinpointing parameters involved in app failures. Tan et al. [223] proposed Droix to automatically repair failed Android apps. They perform the fix directly on app apks. We have proposed in Chapter 5 to automatically mine

crash fixes from the app evolution history. These fix templates can facilitate Automatic Program Repair tools such as Droix to fix failed apps. Despite these curative approaches that address the app failures after they occur, preventive measures that scan for defects are still not well studied. Although a general purpose scanner for failure-inducing bugs is not available, some special cases have been studied. For example, Song [212] proposed ServDroid to detect Android *Service* usage inefficiencies. One inefficiency template will cause the *Service* to fail. However, this template is manually abstracted and the method does not scale. More automation is in much need.

Android Framework API Profiling: Approaches for profiling Android API methods mainly study how API usages and API evolution affect the quality of the apps. Li et al. [140] proposes CiD to detect compatibility issues caused by improper API version updates. CiD systematically models the life cycle of the Android APIs and flag usages that can lead to potential compatibility issues. Several issues are confirmed to lead to app failures. Gao et al. [64] studied the security hazards caused by misusing crypto APIs in the apps. However, they found that app developers may even introduce new bugs instead of fixing existing ones by updating to new APIs. Li et al. [142] proposes CDA to have an insight on how app developers should react to the deprecated APIs. They studied the whole life cycle of API deprecation and have listed several key findings. Fazzini et al. [58] provide automatically generated patches to update API usages in real-world Android apps. Their approach is based on code evolution found from other example apps with the same changes. McDonnell et al. [172]. performed a case study to examine the co-evolution behavior of Android API and dependent applications. Their study made use of change-tracking history on GitHub. They further found that API usage adaptation code is more defect prone than the ones without. Linares-Vásquez et al. [152] focused on API usage patterns in the context of Android framework that may induce great energy consumption of Android devices.

Statically Analyzing the Android Framework: Several research works have built on static analysis techniques for Android apps [23, 139, 171, 202]. As highlighted in a systematic literature review [139], most of these works mainly focused on the security and privacy aspects of Android apps. Specifically, tools like FlowDroid [23], IccTA [132] used taint analysis techniques to address private data leaks. To the best of our knowledge, we are the first to apply taint analysis to inspect framework-specific exceptions. Beyond apps, several other works targeted the Android framework, mostly to mine useful information. Bartel et al. [31] statically analyzed the Android framework to extract permission checks for APIs in order to infer the list of permissions required when using a given API method. One main contribution of this work is to build a field-sensitive class-hierarchy analysis (CHA-Android) call graph to prune irrelevant calling edges for a more precise result. Similarly, in our work, we used the basic CHA since we do not want to miss any unchecked exception instance that could happen during run-time. Au et al. [24] proposed PScout, to also identify permission checks in the Android framework. PScout extracts permission specification from the AOSP using static analysis. Cao et al. [37] proposed EdgeMiner in an attempt to mine implicit control flow transitions through the Android framework. Many static analysis tools base their work on computing a precise Control Flow Graph of the apps. However, it is pointed out by Cao et al. that without adding the implicit control flows, these graphs are imprecise. EdgeMiner thus mines this information by directly analyzing the AOSP.

6.7 Summary

We have proposed a tool-supported approach named `Afuera` to automatically retrieve key information on unchecked exceptions associated to Android framework APIs, which we refer to as UE-API methods. After a thorough examination of the entire Android framework, we have found 5 467 APIs that may throw unchecked exceptions. These represent 24% of framework APIs, making unchecked exceptions a concern that can affect any app developer. We have also proposed a taint analysis based approach to pinpoint the API parameters that may be involved in the triggering of the unchecked

exceptions. Overall, our study shows that UE-API methods are prevalent in the Android framework, are frequently used in Android apps, and are often related to UI interactions. Evaluation results on 50 randomly sampled APIs suggest that `Afuera` provides high precision (high true positive rates). Finally, we overview 3 usage scenarios for `Afuera`, targeting at parameter constraint checking, fuzzing for app failures and locating failure-inducing bugs.

7 Conclusions and Future Work

In this chapter, we revisit the main contributions of this dissertation and present potential future research directions.

Contents

7.1 Conclusions **108**
7.2 Future Work **108**
7.2.1 Verification of Parameter Constraints 109
7.2.2 Fuzzing for App Crashes 109
7.2.3 Improve the Accuracy of Bug Locating 109

7.1 Conclusions

In this dissertation, we targeted taming Android app crashes. To this end, we explored two paths: the curative toolchain and the preventive toolchain.

We have contributed 3 parts for the curative toolchain: exposing the app crashes, locating the crash-inducing bugs, and mining fixing templates for the app crashes.

First, we performed a systematic literature review on the topic of Android app testing. Our review has explored 103 papers that were published in major conferences, workshops and journals in software engineering, programming language, and security domain. We have then proposed a taxonomy of the related research exploring several dimensions including the objectives (i.e., what functional or non-functional concerns are addressed by the approaches) that were pursued, and the techniques (i.e., what type of testing methods – mutation, concolic, etc.) that were leveraged. We have further explored the assessments presented in the literature, highlighting the lack of established benchmarks to clearly monitor the progress made in the field. With this SLR, we know how to select the suitable testing tools for exposing Android app crashes: by examining if they require massive instrumentation, if they require source, if they have good test coverage, etc.

Second, we performed an empirical study on the locations of crash-inducing bugs. This study shows that 37% crash types are related to bugs that are outside the stack traces, which imposes challenges to the localization problem. We then proposed ANCHOR, a two-phase categorization and localization tool that is able to generate a ranked list of bug locations for developers to examine. The effectiveness of ANCHOR is assessed with both this empirical dataset and an in-the-wild scenario on a third-party dataset. Our work brings inspiring insights into the crashing faults localization problem for Android app crashes.

Third, we target a new research direction attempting to mine Android crash fixes from Android market apps, which usually do not have an open change-tracking or issue-tracking system. We successfully generated 17 fine-grained fix templates, which were evaluated to be effective in patching 17 out of 20 real-world crashed apps. CraftDroid can benefit the automatic program repair (APR) community [124, 127, 153–159] in establishing a new means to augment datasets. CraftDroid can also benefit the developer community in recommending effective patches to fix their crashed apps.

Finally, for the preventive toolchain, we have proposed a tool-supported approach named `Afuera` to automatically retrieve key information on unchecked exceptions associated to Android framework APIs, which we refer to as UE-API methods. After a thorough examination of the entire Android framework, we have found 5 467 APIs that may throw unchecked exceptions. These represent 24% of framework APIs, making unchecked exceptions a concern that can affect any app developer. We have also proposed a taint analysis based approach to pinpoint the API parameters that may be involved in the triggering of the unchecked exceptions. Overall, our study shows that UE-API methods are prevalent in the Android framework, are frequently used in Android apps, and are often related to UI interactions. Evaluation results on 50 randomly sampled APIs suggest that `Afuera` provides high precision (high true positive rates). `Afuera` is the first step towards assisting Android app developers in programming crash-inducing bug-free Android apps.

To conclude, we have carried out 4 studies that jointly push forward the task of taming Android app crashes.

7.2 Future Work

In this section, we list several future works that are in line with this dissertation.

7.2.1 Verification of Parameter Constraints

By taking advantage of the information yielded by `Afuera`, developers can better protect the usage of UE-API methods, and in particular the usage of specific arguments that could lead to the triggering of unchecked exceptions. When developers are programming a functionality based on a UE-API method, it is important to control the usage of the parameter, e.g., through a value check. However, `Afuera` currently does not support summarizing the constraint on the argument in order to avoid the exception. Trying to automate this function is an important future work.

7.2.2 Fuzzing for App Crashes

`Afuera` informs about which framework API method can throw an unchecked exception and from where this unchecked exception originates (i.e., the throw statement). This information could be used to guide the fuzzing of a given UE-API method with the objective to trigger the unchecked exception. In this way, we could check the robustness of an Android app, i.e., we could check if the app would not fail if the unchecked exception is triggered. Luckily, there are already guided fuzzing testing tools like FuzzDroid [199] which targets at reaching a code position. However, we need to provide customized inputs for FuzzDroid: (1) FuzzDroid targets at a malicious code statement; in our case the target could be the throw statement. (2) FuzzDroid fuzzes a default set of execution environment query APIs; in our case, it could be application space variables that are linked to the condition guarding the throw statement. If the throw statement is successfully reached, and developers do not handle the unchecked exception thrown from the UE-API, the app will fail.

7.2.3 Improve the Accuracy of Bug Locating

As presented in Chapter 4, bug localization [5–7, 110, 126, 128, 236] is a key step that precedes fixing. However, there is still room for improvement. We argue that `Afuera` can be used to better verify the arguments of UE-API methods that are related to unchecked exceptions. However, we showed that 672 UE-APIs have no parameters. In this case, when fixing app failures induced by unchecked exceptions thrown from those UE-API methods that take no argument, developers still need to know which variable(s), in the application space, is/are linked to the condition that guards the throw statement. By following the same approach described in Step 5 of `Afuera`, developers can target any application variable (instead of only UE-API arguments) that they suspect. Then by performing Step 6, `Afuera` will assert whether the suspected variable(s) is linked to the triggering of the unchecked exceptions, and thus to the app failure.

List of papers, tools & services

Papers included in this dissertation:

- Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 2018
- Pingfan Kong, Li Li, Jun Gao, Timothée Riom, Yanjie Zhao, Tegawendé F Bissyandé, and Jacques Klein. Anchor: Locating android framework-specific crashing faults. *arXiv preprint arXiv:2008.01676*, 2020
- Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue- and change-tracking systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–89. ACM, 2019
- Pingfan Kong, Li Li, Jordan Samhi, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Afuera: Automatically documenting android framework apis for unchecked exceptions. In *peer review for International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021

Papers not included in this dissertation:

- Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017
- Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. Negative results on mining crypto-api usage rules in android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 388–398. IEEE, 2019
- Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, 2019
- Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 939–951, 2020

Tools and Datasets:

- **SLR** - `http://lilicoding.github.io/TA2Repo/`
- **ANCHOR** - `https://anchor-locator.github.io`
- **CraftDroid** - `https://craftdroid.github.io`
- **Afuera** - `https://afuera.github.io`

Services (Reviewer):

- IEEE Transaction on Software Engineering
- IEEE Transaction on Reliability
- Springer Journal on Automated Software Engineering

Bibliography

[1] Application fundamentals. https://stackoverflow.com/questions/37769188/how-to-request-android-permission-api-23. Accessed: 2021-03-02.

[2] Application fundamentals. https://stackoverflow.com/questions/39810434/android-api-23-requesting-multiple-permissions. Accessed: 2021-03-02.

[3] Application fundamentals. https://stackoverflow.com/questions/33666071/android-marshmallow-request-permission. Accessed: 2021-03-02.

[4] Supported devices -google play help. https://support.google.com/googleplay/answer/1727131?hl=en-GB. Accessed: 2018-08-24.

[5] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[7] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[8] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *International Symposium on Software Testing and Analysis*, 2015.

[9] Muhammad Adinata and Inggriani Liem. A/b test tools of native mobile application. In *International Conference on Data and Software Engineering*, 2014.

[10] Oum-EI-Kheir Aktouf, Tao Zhang, Jerry Gao, and Tadahiro Uehara. Testing location-based function services for mobile applications. In *Symposium on Service-Oriented System Engineering*, 2015.

[11] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[12] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Ugo Gentile, Gianluca Mele, Roberto Nardone, and Valeria Vittorini. Improving code coverage in android apps testing by exploiting patterns and automatic test case generation. In *International workshop on Long-term industrial collaboration on software engineering*, 2014.

[13] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana. Agrippin: a novel search based testing technique for android applications. In *International Workshop on Software Development Lifecycle for Mobile*, 2015.

[14] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *International Conference on Software Testing, Verification and Validation Workshops*, 2011.

[15] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Nicola Amatucci. Considering context events in event-based testing of mobile applications. In *International Conference on Software Testing, Verification and Validation Workshops*, 2013.

[16] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[17] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Bryan Robbins. Testing android mobile applications: Challenges, strategies, and approaches. *Advances in Computers*, 89(6):1–52, 2013.

[18] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. Mobiguitar: Automated model-based testing of mobile apps. In *IEEE Software*, 2015.

[19] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[20] Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Tan Hee Beng Kuan, and Charlie Soh. Achieving high code coverage in android ui testing via automated widget exercising. In *Asia-Pacific Software Engineering Conference*, 2016.

[21] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[22] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 13–24. IEEE, 2017.

[23] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[24] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[25] Andrea Avancini and Mariano Ceccato. Security testing of the communication among android applications. In *International Workshop on Automation of Software Test*, 2013.

[26] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.

[27] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *International Conference on Automated Software Engineering*, 2016.

[28] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. Towards model checking android applications. *IEEE Transactions on Software Engineering*, 44(6):595–612, 2018.

[29] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[30] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38, 2012.

[31] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[32] Sebastian Bauersfeld. Guidiff - a regression testing tool for graphical user interfaces. In *International Conference on Software Testing, Verification and Validation*, 2013.

[33] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for android applications. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[34] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[35] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and nullpointerexception s for java bytecode. In *International Conference on Formal Verification of Object-Oriented Software*, pages 123–141. Springer, 2011.

[36] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[37] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[38] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Secure Systems Lab at Stony Brook University, Tech. Rep*, pages 1–18, 2007.

[39] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering*, 41(2):198–220, 2015.

[40] Xiangping Chen and Zhensheng Xu. Towards automatic consistency checking between web application and its mobile application. In *International Conference on Software Engineering and Knowledge Engineering*, 2014.

[41] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

[42] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*, 2015.

[43] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing gui event traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[44] Riccardo Coppola, Luca Ardito, and Marco 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*, pages 8–14, 2019.

[45] Antonio Corradi, Mario Fanelli, Luca Foschini, and Marcello Cinque. Context data distribution with quality guarantees for android-based mobile systems. In *Security and Communication Networks*, 2013.

[46] Pedro Costa, Miguel Nabuco, and Ana C. R. Paiva. Pattern based gui testing for mobile applications. In *International Conference on the Quality of Information and Communications Technology*, 2014.

[47] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. Towards mutation analysis of android apps. In *International Conference on Software Testing, Verification and Validation Workshops*, 2015.

[48] Karthikeyan Balaji Dhanapal, K Sai Deepak, Saurabh Sharma, Sagar Prakash Joglekar, Aditya Narang, Aditya Vashistha, Paras Salunkhe, Harikrishna G. N. Rai, Arun Agrahara Somasundara, and Sanjoy Paul. An innovative system for remote and automated testing of mobile phone applications. In *Service Research and Innovation Institute Global Conference*, 2012.

[49] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015.

[50] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, 2018.

[51] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 82–93. ACM, 2016.

[52] Ana Rosario Espada, María del Mar Gallardo, Alberto Salmerón, and Pedro Merino. Runtime verification of expected energy consumption in smartphones. In *Model Checking Software*, 2015.

[53] F-Droid. F-droid. `https://www.f-droid.org/en/`. Accessed: 2021-03-02.

[54] F-Droid. Open source app repository. `https://f-droid.org/en/`.

[55] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 486–497, 2018.

[56] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 408–419. IEEE, 2018.

[57] Guilherme de Cleva Farto and Andre Takeshi Endo. Evaluating the model-based testing approach in the context of mobile applications. In *Electronic Notes in Theoretical Computer Science*, 2015.

[58] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 204–215, 2019.

[59] Eclipse Foundation. Ide eclipse. `https://www.eclipse.org`.

[60] Dominik Franke, Stefan Kowalewski, Carsten Weise, and Nath Prakobkosol. Testing conformance of life cycle dependent properties of mobile applications. In *International Conference on Software Testing, Verification and Validation*, 2012.

[61] Fraunhofer. Codeinspect tool of fraunhofer. `https://codeinspect.sit.fraunhofer.de/`, 2020.

[62] Software Testing Fundamentals. Software testing levels. http://softwaretestingfundamentals.com/software-testing-levels/.

[63] Jerry Gao, Wei-Tek Tsai, Rimi Paul, Xiaoying Bai, and Tadahiro Uehara. Mobile testing-as-a-service (mtaas)–infrastructures, issues, solutions and needs. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 158–167. IEEE, 2014.

[64] Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. Negative results on mining crypto-api usage rules in android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 388–398. IEEE, 2019.

[65] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. On vulnerability evolution in android apps. In *The 40th International Conference on Software Engineering, Poster Track (ICSE 2018)*, 2018.

[66] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Should you consider adware as malware in your study? In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 604–608. IEEE, 2019.

[67] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, 2019.

[68] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 939–951, 2020.

[69] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–29, 2018.

[70] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *International Conference on Software Engineering*, 2013.

[71] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *International Conference on Mobile Software Engineering and Systems*, 2016.

[72] Google. Android api reference. `https://developer.android.com/reference`.

[73] Google. Android debug bridge. `https://developer.android.com/studio/command-line/adb`.

[74] Google. Android manifest overview. `https://developer.android.com/guide/topics/manifest/manifest-intro#compatibility`. Accessed: 2021-03-02.

[75] Google. Android platform architecture. `https://developer.android.com/guide/platform`. Accessed: 2021-03-02.

[76] Google. Api reference to startactivityforresult. `https://developer.android.com/reference/android/app/Activity`.

[77] Google. Application fundamentals. `https://developer.android.com/guide/components/fundamentals`. Accessed: 2021-03-02.

[78] Google. Codenames, tags, and build numbers. `https://source.android.com/setup/start/build-numbers`. Accessed: 2021-03-02.

[79] Google. Google play store. `https://play.google.com/store/apps`.

[80] Google. Ide androidstudio. `https://developer.android.com/studio?hl=es`.

[81] Google. Logcat command-line tool. `https://developer.android.com/studio/command-line/logcat`. Accessed: 2021-03-02.

[82] Google. Request app permissions. `https://developer.android.com/training/permissions/requesting.html`. Accessed: 2021-03-02.

[83] Google. Crashes | android developers. `https://developer.android.com/topic/performance/vitals/crash`, 2020.

[84] Tobias Griebe and Volker Gruhn. A model-based approach to test automation for context-aware mobile applications. In *Annual ACM Symposium on Applied Computing*, 2014.

[85] Tobias Griebe, Marc Hesenius, and Volker Gruhn. Towards automated ui-tests for sensor-based mobile applications. In *International Conference on Intelligent Software Methodologies, Tools and Techniques*, 2015.

[86] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, 148:88–104, 2019.

[87] Chenkai Guo, Jing Xu, Hongji Yang, Ying Zeng, and Shuang Xing. An automated testing approach for inter-application security in android. In *International Workshop on Automation of Software Test*, 2014.

[88] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[89] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. Vida: Visual interactive debugging. In *2009 IEEE 31st International Conference on Software Engineering*, pages 583–586. IEEE, 2009.

[90] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In *European Symposium on Research in Computer Security*, 2015.

[91] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *International Symposium on Software Testing and Analysis*, 2015.

[92] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177. IEEE, 2018.

[93] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130. IEEE Press, 2013.

[94] Clemens Holzmann and Patrick Hutflesz. Multivariate testing of native mobile applications. In *International Conference on Advances in Mobile Computing and Multimedia*, 2014.

[95] ChunHung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[96] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *International Workshop on Automation of Software Test*, 2011.

[97] Yongjian Hu and Iulian Neamtiu. Fuzzy and cross-app replay for smartphone apps. In *International Workshop on Automation of Software Test*, 2016.

[98] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. Automatically verifying and reproducing event-based races in android apps. In *International Symposium on Software Testing and Analysis*, 2016.

[99] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 532–542, 2018.

[100] Junfei Huang. Appacts: Mobile app automated compatibility testing service. In *International Conference on Mobile Cloud Computing, Services, and Engineering*, 2014.

[101] Yuguang Huang and Lei Li. Naive bayes classification algorithm based on small sample set. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 34–39. IEEE, 2011.

[102] Trupti S Indi, Pratibha S Yalagi, and Manisha A Nirgude. Use of java exception stack trace to improve bug fixing skills of intermediate java learners. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 194–198. IEEE, 2016.

[103] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Energy-aware test-suite minimization for android apps. In *International Symposium on Software Testing and Analysis*, 2016.

[104] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 37–48. IEEE, 2016.

[105] Marek Janicki, Mika Katara, and Tuula Pääkkönen. Obstacles and opportunities in deploying model-based gui testing of mobile software: a survey. *Software Testing, Verification and Reliability*, 22(5):313–341, 2012.

[106] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2013.

[107] Bo Jiang, Peng Chen, Wing Kwong Chan, and Xinchao Zhang. To what extent is stress testing of android tv applications automated in industrial environments? In *IEEE Transactions on Reliability*, 2015.

[108] Shujuan Jiang, Wei Li, Haiyang Li, Yanmei Zhang, Hongchang Zhang, and Yingqi Liu. Fault localization for null pointer exception based on stack trace and program slicing. In *2012 12th International Conference on Quality Software*, pages 9–12. IEEE, 2012.

[109] Shujuan Jiang, Hongchang Zhang, Qingtan Wang, and Yanmei Zhang. A debugging approach for java runtime exceptions based on program slicing and stack traces. In *2010 10th International Conference on Quality Software*, pages 393–398. IEEE, 2010.

[110] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[111] Yu Kang, Yangfan Zhou, Min Gao, Yixia Sun, and Michael R Lyu. Experience report: Detecting poor-responsive ui in android applications. In *International Symposium on Software Reliability Engineering*, 2016.

[112] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R Lyu. Diagdroid: Android performance diagnosis via anatomizing asynchronous executions. In *International Conference on Foundations of Software Engineering*, 2016.

[113] Joseph Chan Joo Keng, Lingxiao Jiang, Tan Kiat Wee, and Rajesh Krishna Balan. Graph-aided directed testing of android applications for checking runtime privacy behaviours. In *International Workshop on Automation of Software Test*, 2016.

[114] Hammad Khalid, Meiyappan Nagappan, and Ahmed Hassan. Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps. *IEEE Software*, 2015.

[115] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

[116] Haeng-Kon Kim. Hybrid mobile testing model. In *International Conferences, ASEA and DRBC*, 2012.

[117] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. In *Technical report, EBSE Technical Report EBSE-2007-01*. sn, 2007.

[118] Konstantin Knorr and David Aspinall. Security testing for android mhealth apps. In *Software Testing, Verification and Validation Workshops*, 2015.

[119] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[120] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue- and change-tracking systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–89. ACM, 2019.

[121] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 2018.

[122] Pingfan Kong, Li Li, Jun Gao, Timothée Riom, Yanjie Zhao, Tegawendé F Bissyandé, and Jacques Klein. Anchor: Locating android framework-specific crashing faults. *arXiv preprint arXiv:2008.01676*, 2020.

[123] Pingfan Kong, Li Li, Jordan Samhi, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Afuera: Automatically documenting android framework apis for unchecked exceptions. In *peer review for International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021.

[124] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017.

[125] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791*, 2018.

[126] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325. ACM, 2019.

[127] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: Bug report driven program repair. In *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019.

[128] Sangeeta Lal and Ashish Sureka. A static technique for fault localization using character n-gram based information retrieval model. In *Proceedings of the 5th India Software Engineering Conference*, pages 109–118, 2012.

[129] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using s park. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[130] Li Li. Mining androzoo: A retrospect. In *The Doctoral Symposium of 33rd International Conference on Software Maintenance and Evolution (ICSME-DS 2017)*, 2017.

[131] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC 2015)*, 2015.

[132] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[133] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*, 2014.

[134] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-17)*, pages 136–143. IEEE, 2017.

[135] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2019.

[136] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[137] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[138] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[139] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[140] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163, 2018.

[141] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[142] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Cda: Characterising deprecated android apis. *Empirical Software Engineering*, pages 1–41, 2020.

[143] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.

[144] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017.

[145] Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. Effectively manifesting concurrency bugs in android apps. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 209–216. IEEE, 2016.

[146] Xiujiang Li, Yanyan Jiang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. User guided automation for testing mobile apps. In *Asia-Pacific Software Engineering Conference*, 2014.

[147] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 23–26. IEEE, 2017.

[148] C.-J. Liang, N. Lane, N. Brouwers, L. Zhang, B. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobil app testing through contextual fuzzing. In *International conference on Mobile computing and networking*, 2014.

[149] Milind G Limaye. *Software testing.* Tata McGraw-Hill Education, 2009.

[150] Ying-Dar Lin, Edward T.-H. Chu, Shang-Che Yu, and Yuan-Cheng Lai. Improving the accuracy of automated gui testing for embedded systems. In *IEEE Software*, 2013.

[151] Ying-Dar Lin, Jose F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. On the accuracy, efficiency, and reusability of automated test oracles for android devices. In *IEEE Transactions on Software Engineering*, 2014.

[152] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11, 2014.

[153] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Anil Koyuncu, Kisub Kim, Taeyoung Kim, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*. IEEE, 2019.

[154] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 2018.

[155] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. A closer look at real-world patches. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, pages 275–286. IEEE, 2018.

[156] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113. IEEE, 2019.

[157] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, page 456—467.

[158] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019.

[159] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, pages 658–662, 2018.

[160] Zhifang Liu, Xiaopeng Gao, and Xiang Long. Adaptive random testing of mobile application. In *International Conference on Computer Engineering and Technology*, 2010.

[161] Google LLC. App manifest overview. `https://developer.android.com/guide/topics/manifest/manifest-intro`, 2020. Accessed: 2020-01-26.

[162] Google LLC. Test google play billing. `https://developer.android.com/google/play/billing/billing_testing.html`, 2020. Accessed: 2020-01-26.

[163] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[164] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *International Workshop on Automation of Software Test*, 2012.

[165] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[166] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[167] Amiya K Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *International Conference on Dependable systems and Networks*, 2012.

[168] Luka Malisa, Kari Kostiainen, Michael Och, and Srdjan Capkun. Mobile application impersonation detection using dynamic user interface extraction. In *European Symposium on Research in Computer Security*, 2016.

[169] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2016.

[170] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2016.

[171] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2017.

[172] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE, 2013.

[173] Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260, 2003.

[174] Abel Méndez-Porras, Christian Quesada-López, and Marcelo Jenkins. Automated testing of mobile applications: a systematic map and review. In *XVIII Ibero-American Conference on Software Engineering, Lima-Peru*, pages 195–208, 2015.

[175] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[176] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. Spam filtering with naive bayes-which naive bayes? In *CEAS*, volume 17, pages 28–69. Mountain View, CA, 2006.

[177] Rupert Miller and David Siegmund. Maximally selected chi square statistics. *Biometrics*, pages 1011–1016, 1982.

[178] mingyu.planb@gmail.com. Open source app repository. `https://pypi.org/project/google-play-scraper/`.

[179] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. Sig-droid: Automated system input generation for android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 461–471. IEEE, 2015.

[180] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. Sig-droid: Automated system input generation for android applications. In *International Symposium on Software Reliability Engineering*, 2015.

[181] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of android applications. In *International Conference on Software Engineering*, 2016.

[182] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[183] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 33–44. IEEE, 2016.

[184] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18. IEEE, 2017.

[185] Inês Coimbra Morgado and Ana CR Paiva. The impact tool: Testing ui patterns on mobile applications. In *International Conference on Automated Software Engineering*, 2015.

[186] Inês Coimbra Morgado and Ana CR Paiva. Testing approach for mobile applications through reverse engineering of ui patterns. In *International Conference on Automated Software Engineering Workshop*, 2015.

[187] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 29–35. IEEE, 2012.

[188] Ward Muylaert and Coen De Roover. Untangling composite commits using program slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 193–202. IEEE, 2018.

[189] Vikrant Nandakumar, Vijay Ekambaram, and Vivek Sharma. Appstrument - a unified app instrumentation and automated playback framework for testing mobile applications. In *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2013.

[190] Phu H Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. An extensive systematic review on the model-driven development of secure systems. *Information and Software Technology*, 68:62–81, 2015.

[191] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[192] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press, 2015.

[193] Oracle. Unchecked exceptions — the controversy. `https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html`.

[194] Oracle. Viewing contents of jar file. `https://docs.oracle.com/javase/tutorial/deployment/jar/view.html`.

[195] Šarūnas Packevičius, Andrej Ušaniov, Šarūnas Stanskis, and Eduardas Bareiša. The testing method based on image analysis for automated detection of ui defects intended for mobile applications. In *International Conference on Information and Software Technologies*, 2015.

[196] Ju Qian and Di Zhou. Prioritizing test cases for memory leaks in android applications. In *Journal of Computer Science and Technology*, 2016.

[197] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1033–1044, 2011.

[198] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *International Conference on Software Engineering*, 2016.

[199] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 300–311. IEEE, 2017.

[200] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[201] Sarker T. Ahmed Rumee and Donggang Liu. Droidtest: Testing android applications for leakage of private information. In *International Journal of Information Security*, 2013.

[202] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017.

[203] Ben Sadeh, Kjetil Ørbekk, Magnus M. Eide, Njaal C.A. Gjerde, Trygve A. Tønnesland, and Sundar Gopalakrishnan. Towards unit testing of user interface code for android mobile applications. In *International Conference on Software Engineering and Computer Systems*, 2011.

[204] Sebastien Salva and Stassia R. Zafimiharisoa. Apset, an android application security testing tool for detecting intent-based vulnerabilities. In *International Journal on Software Tools for Technology Transfer*, 2014.

[205] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics*, 2014.

[206] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE, 2010.

[207] Hossain Shahriar, Sarah North, and Edward Mawangi. Testing of memory leak in android applications. In *International Symposium on High-Assurance Systems Engineering*, 2014.

[208] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Nuria Oliver, and Alan Hanjalic. Climf: learning to maximize reciprocal rank with collaborative less-is-more filtering. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 139–146, 2012.

[209] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164, 2009.

[210] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

[211] Kwangsik Song, Ah-Rim Han, Sehun Jeong, and Sung Deok Cha. Generating various contexts from permissions for testing android applications. In *International Conference on Software Engineering and Knowledge Engineering*, 2015.

[212] Wei Song, Jing Zhang, and Jeff Huang. Servdroid: detecting service usage inefficiencies in android applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 362–373, 2019.

[213] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[214] Oleksii Starov, Sergiy Vilkomir, Anatoliy Gorbenko, and Vyacheslav Kharchenko. Testing-as-a-service for mobile applications: state-of-the-art survey. In *Dependability Problems of Complex Information Systems*, pages 55–71. Springer, 2015.

[215] StatCounter Global Stats. Operation system market share worldwide. `https://gs.statcounter.com/os-market-share`.

[216] Ting Su. Fsmdroid: guided gui testing of android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 689–691. IEEE, 2016.

[217] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering*, 2020.

[218] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[219] Guillermo Suarez-Tangil, Mauro Conti, Juan E Tapiador, and Pedro Peris-Lopez. Detecting targeted smartphone malware with behavior-triggering stochastic models. In *European Symposium on Research in Computer Security*, 2014.

[220] Quan Sun, Lei Xu, Lin Chen, and Weifeng Zhang. Replaying harmful data races in android apps. In *International Symposium on Software Reliability Engineering Workshop*, 2016.

[221] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. In *IEEE International Conference on Software Testing, Verification and Validation*, 2011.

[222] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[223] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 187–198. ACM, 2018.

[224] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. Generating test cases to expose concurrency bugs in android applications. In *International Conference on Automated Software Engineering*, 2016.

[225] Vincent F Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 45–57. ACM, 2017.

[226] Mark Turner, Barbara Kitchenham, David Budgen, and OP Brereton. Lessons learnt undertaking a large-scale systematic literature review. In *Proceedings of EASE*, volume 8, 2008.

[227] David M Ungar. Method and apparatus for optimizing the null pointer exception in an object-oriented programming environment with statically typed variables, July 25 2000. US Patent 6,094,664.

[228] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[229] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[230] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial android applications. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2014.

[231] Sergiy Vilkomir and Brandi Amstutz. Using combinatorial approaches for testing mobile applications. In *International Conference on Software Testing, Verification, and Validation Workshops*, 2014.

[232] Nephi A Walton, Mollie R Poynton, Per H Gesteland, Chris Maloney, Catherine Staes, and Julio C Facelli. Predicting the start week of respiratory syncytial virus outbreaks using real time weather variables. *BMC medical informatics and decision making*, 10(1):68, 2010.

[233] Mian Wan, Yuchen Jin, Ding Li, and William G. J. Halfond. Detecting display energy hotspots in android apps. In *International Conference on Software Testing, Verification and Validation*, 2015.

[234] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[235] Jue Wang, Yanyan Jiang, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu. Aatt+: Effectively manifesting concurrency bugs in android apps. *Science of Computer Programming*, 163:1–18, 2018.

[236] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11, 2015.

[237] Afzal Wasif, Torkar Richard, and Feldt Robert. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51:957–976, 2009.

[238] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[239] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[240] Lili Wei, Yepang Liu, and Shing-Chi 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*, pages 226–237. ACM, 2016.

[241] Lili Wei, Yepang Liu, and Shing-Chi 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, ASE 2016*, pages 226–237, 2016.

[242] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190. IEEE, 2014.

[243] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[244] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214. ACM, 2014.

[245] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.

[246] Xiangyu Wu, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. Testing android apps via guided gesture event generation. In *Asia-Pacific Software Engineering Conference*, 2016.

[247] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *IEEE Symposium on Security and Privacy*, 2015.

[248] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

[249] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 910–913. ACM, 2015.

[250] Dacong Yan, Shengqian Yang, and Atanas Rountev. Systematic testing for resource leaks in android applications. In *International Symposium on Software Reliability Engineering*, 2013.

[251] Ping Yan and Zheng Yan. A survey on dynamic mobile malware detection. *Software Quality Journal*, pages 1–29, 2017.

[252] Chao Yang, Guangliang Yang, Ashish Gehani, Vinod Yegneswaran, Dawood Tariq, and Guofei Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *ACM SIGSAC conference on Computer and communications security*, 2016.

[253] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. Intentfuzzer: detecting capability leaks of android applications. In *ACM symposium on Information, computer and communications security*, 2014.

[254] Shengqian Yang, Dacong Yan, and Atanas Rountev. Testing for poor responsiveness in android applications. In *International Workshop on the Engineering of. Mobile-Enabled Systems*, 2013.

[255] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.

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

[257] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104. IEEE, 2012.

[258] Zhen Yang, Xiangfei Nie, Weiran Xu, and Jun Guo. An approach to spam detection by naive bayes ensemble based on decision induction. In *Sixth International Conference on Intelligent Systems Design and Applications*, volume 2, pages 861–866. IEEE, 2006.

[259] Chao-Chun Yeh and Shih-Kun Huang. Covdroid: A black-box testing coverage system for android. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 3, pages 447–452. IEEE, 2015.

[260] Chao-Chun Yeh, Han-Lin Lu, Chun-Yen Chen, Kee-Kiat Khor, and Shih-Kun Huang. Craxdroid: Automatic android system testing by selective symbolic execution. In *International Conference on Software Security and Reliability-Companion*, 2014.

[261] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *International Conference on Software Testing, Verification, and Validation*, 2014.

[262] Samer Zein, Norsaremah Salleh, and John Grundy. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software*, 117:334–356, 2016.

[263] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated test input generation for android: are we really there yet in an industrial case? In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[264] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24(6):3394–3434, 2019.

[265] Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in android applications. In *International Workshop in Automation of Software Test*, 2016.

[266] Pingyu Zhang and Sebastian G. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. In *International Conference on Software Engineering*, 2014.

[267] Tao Zhang, Jerry Gao, Oum-El-Kheir Aktouf, and Tadahiro Uehara. Test model and coverage analysis for location-based mobile services. In *International Conference on Software Engineering and Knowledge Engineering*, 2015.

[268] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. Compatibility testing service for mobile applications. In *Symposium on Service-Oriented System Engineering*, 2015.

[269] Shuai Zhao, Xiaohong Li, Guangquan Xu, Lei Zhang, and Zhiyong Feng. Attack tree based android malware detection with hybrid analysis. In *International Conference on Trust, Security and Privacy in Computing and Communications*, 2014.

[270] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. Recdroid: automatically reproducing android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering*, pages 128–139. IEEE Press, 2019.

[271] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards black box testing of android apps. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 501–510. IEEE, 2015.

[272] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[273] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.

[274] Haowen Zhu, Xiaojun Ye, Xiaojun Zhang, and Ke Shen. A context-aware approach for dynamic gui testing of android applications. In *Computer Software and Applications Conference*, 2015.