Why My App Crashes? Understanding and Benchmarking Framework-specific Exceptions of Android apps

Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su

Abstract—Mobile apps have become ubiquitous. Ensuring their correctness and reliability is important. However, many apps still suffer from occasional to frequent crashes, weakening their competitive edge. Large-scale, deep analyses of the characteristics of real-world app crashes can provide useful insights to both developers and researchers. However, such studies are difficult and yet to be carried out — this work fills this gap. We collected 16,245 and 8,760 unique exceptions from 2,486 open-source and 3,230 commercial Android apps, respectively, and observed that the exceptions thrown from Android framework (termed *"framework-specific exceptions"*) account for the majority. With one-year effort, we (1) extensively investigated these framework-specific exceptions, and (2) further conducted an online survey of 135 professional app developers about how they analyze, test, reproduce and fix these exceptions. Specifically, we aim to understand the framework-specific exceptions from several perspectives: (i) their characteristics (*e.g.*, manifestation locations, fault taxonomy), (ii) the developers' testing practices, (iii) existing bug detection techniques' effectiveness, (iv) their reproducibility and (v) bug fixes. To enable follow-up research (*e.g.*, bug understanding, detection, localization and repairing), we further systematically constructed, *DroidDefects*, the first comprehensive and largest benchmark of Android app exception bugs. This benchmark contains 33 *reproducible* exceptions (with test cases, stack traces, faulty and fixed app versions, bug types, etc.), and 3,696 *ground-truth* exceptions (real faults manifested by automated testing tools), which cover the apps with different complexities and diverse exception types. Based on our findings, we also built two prototype tools: Stoat+, an optimized dynamic testing tool, which quickly uncovered three previously-unknown, fixed crashes in Gmail and Google+; ExLocator, an exception localization tool, which can locate the root causes of specific exception types. Our dataset, benchmark and tools are publicly available on *https://github.com/tingsu/droiddefects*.

Index Terms—Mobile applications, Android applications, empirical study, exception analysis, software testing, bug reproducibility

1 Introduction

Mobile apps have become ubiquitous recently. For example, Google Play, Google's official Android app market, contains over three million apps; over 50,000 apps are continuously published on it [1] each month. To ensure the competitive edge, app developers strive to deliver high-quality apps [2]. One of their primary concerns is to prevent fail-stop errors (*i.e.*, app crashes) from releases [3], [4].

1.1 Motivations

In industry, many testing frameworks (*e.g.*, Robotium [5], Appium [6]) and static checking tools (*e.g.*, Lint [7], FindBugs [8]) are available [9], [10] to improve app quality. However, many released apps still suffer from crashes. Two recent studies [11], [12] discovered hundreds of previously unknown crashes in popular and well-tested commercial apps. This may make developers wondering "*why my app crashes?*". Researchers have proposed a number of testing techniques and tools [13], [14], [15], [16], [17], [18], [19], [20], [21], [11], [22], [12], [23], [24] to reveal app crashes. However, none of them investigated the root causes of these crashes. Without the answer to this question, developers may not know how to effectively avoid and fix these bugs. By analyzing the 272,629 issues mined from 2,174 Android apps hosted on GitHub and Google Code, we find nearly 40% of the reported crash issues remain open/unfixed (filtered by the keywords "crash" or "exception" in their issue descriptions). This situation could compromise the app quality, considering these issues may probably lead to fail-stop errors after releasing. Even worse, due to the lack of understanding of root causes, the follow-up research, *e.g.*, bug detection, localization and repairing, might be constrained. For example, existing fault localization [25] and repairing [26], [27] tools for Android apps are limited to a small set of trivial crash bugs. Thus, it is important to conduct such a study — characterizing the root causes from a large-scale, diverse set of real-world app crashes, and investigating how to effectively detect, reproduce, and fix them. However, such a study is difficult and yet to be carried out, which has motivated this work.

Routinely, when an app crashes, the Android runtime system will dump an exception trace that provides certain clues of the issue (*e.g.*, the exception type, message, and the stack of invoked methods). Based on the architecture layer

- *T. Su is with School of Software Engineering, East China Normal University, China and Department of Computer Science, ETH Zurich, Switzerland. Email: tsuletgo@gmail.com. L. Fan is with College of Cyber Science, Nankai Univerisity, China and Nanyang Technological University, Singapore. S. Chen is with College of Intelligence and Computing, Tianjin University, China and Nanyang Technological University, Singapore. Y. Liu is with School of Computer Science and Engineering, Nanyang Technological University, Singapore. Email: {llfan,chensen,yangliu}@ntu.edu.sg. L. Xu is with Department of Computer Science and Engineering, New York University Shanghai, China. Email: lihua.xu@nyu.edu. G. Pu is with School of Computer Science and Software Engineering, East China Normal University, China.Email: ggpu@sei.ecnu.edu.sg Z. Su is with Department of Computer Science, ETH Zurich, Switzerland. Email: zhendong.su@inf.ethz.ch.*
- *Lingling Fan and Geguang Pu are the corresponding authors.*

throwing the exception, each exception can be classified into one of three categories — *application exception*, *framework exception*[1], and *library exception* (*cf.* Section 2.1). Specifically, we find framework exceptions account for the majority of app crashes, affecting over 75% of the projects (*cf.* Section 3). Thus, we focus on analyzing framework exceptions, and also brief the other two exception types (*cf.* Section 3.1).

1.2 Challenges

We face three key challenges in this study. (1) The first is the *lack of comprehensive dataset*. To enable crash analysis, we need a comprehensive set of crashes from many apps. Ideally, each crash is associated with the exception trace, the buggy code version, the bug-triggering test, and the patch (if exists). However, to our knowledge, no such dataset exists. Despite open-source project hosting platforms (*e.g.*, GitHub and Google Code) maintain issue repositories, we find only a small set of crash issues (∼16%) are accompanied with exception traces. Among them, only a small fraction has clear reproduction steps (with target app versions and environment); even if the issue is closed, the faulty code version may not be linked with the fixed one. (2) The second concerns *difficulties in crash analysis*. Analyzing crashes requires deep knowledge of app logic, Android framework, and even third-party libraries. However, no reliable tool exists that can help our analysis. As a result, the crash analysis requires considerable human expertise and efforts. (3) The third is the *validation of analysis results and findings*. To reduce the threats to validity, we need to consider diverse categories/types of apps, and cross-check our findings by referring to the developers' expertise and experience.

To achieve this study, we made substantial efforts in several aspects. Fig. 1 shows the overview of our study.

1.3 Data Collection and Online Survey

We collected 16,245 unique exception traces from 2,486 open-source (F-Droid) apps as our analysis data (see Fig. 1(a)) by (1) mining the issue repositories; and (2) applying the three state-of-the-art app testing tools (Monkey [28], Sapienz [11], and Stoat [12]). We also run the three testing tools on 3,230 Google Play apps, and collected 8,760 unique exception traces, to complement our analysis data. Moreover, we conducted an online survey, and received 135 app developers' responses about how they analyze, test, reproduce and fix exception bugs to cross-validate our analysis results and gain more insights (see Fig. 1(b)).

1.4 Crash Analysis

We aim to answer the following research questions.

- ***RQ1 (Exception Characteristics)***: *What are the characteristics of these exceptions, e.g., exception categories, distributions, and locations of manifestation?*
- ***RQ2 (Root Causes)***: *What are the root causes of framework exceptions? What are the difficulties app developers face when analyzing them?*
- ***RQ3 (Exception Detection)***: *What tools are commonly used by developers to detect exception bugs? Are they satisfactory?*
- ***RQ4 (Auditing Tools)***: *How effective is the state-of-the-art bug detection techniques in manifesting framework exceptions?*
- ***RQ5 (Exception Reproduction)***: *How is the reproducibility of app exception bugs? Are there any difficulties of reproducing?*
- ***RQ6 (Exception Fixing)***: *How do developers fix framework exceptions? Are there any difficulties app developers face?*

Through these questions, we find framework exceptions account for the majority in both open-source and commercial apps. They have lower issue closing rate[2] (only 53%), compared with application exceptions (67%). Through careful analysis, we distilled 11 common fault categories, which have not been well-investigated before (*cf.* Section 3.2).

Informed by the developer survey, we further audited existing automated bug detection tools on framework exceptions (*cf.* Section 3.4). We find dynamic testing tools can reveal framework exceptions, but are still less effective on certain fault categories. Their testing strategies have a big impact on the detection ability. In addition, these testing tools have low reproducing rates (*cf.* Section 3.5). We also find most exceptions can be fixed by five common practices with small patches (fewer than 20 code lines), but developers face several difficulties in fixing (*cf.* Section 3.6).

1.5 Applications

Based on our study, we made several applications: (1) We constructed *DroidDefects*, the *first comprehensive* and *largest* benchmark of Android app exception bugs. It contains 33 reproducible and 3,696 ground-truth exception bugs, and covers diverse exception types, root causes, app complexities and categories, and relevant bug information. It can help follow-up research, *e.g.*, bug understanding, detection, localization, prediction, and patch generation for Android apps. (2) We optimized *Stoat*[3], a GUI testing tool, by integrating a number of testing strategies, which quickly revealed three previously unknown bugs in Gmail and Google+. (3) We built *ExLocator*[4], an exception localization tool, which can help localize the root causes of specific exception types. (4) We also demonstrated the possibility of enhancing static checking and mutation testing for Android apps.

1.6 Contributions

To summarize, we made the following contributions:

- We conducted the first large-scale study to investigate exception bugs (framework exceptions in particular) of Android apps, and identified 11 common fault categories. The results provide useful insights for both researchers and developers.
- Our study evaluated the state-of-the-art bug detection techniques, reviewed the reproducibility of these exceptions, and investigated common fixing practices. The findings motivate more effective bug detection, reproduction, and fixing techniques.
- We conducted an online survey to understand how developers analyze, test, reproduce and fix crashes. This survey gains more insights from the developers' experiences, and also validates our analysis results.
- We constructed *DroidDefects*, the first comprehensive and largest benchmark of Android app exceptions, to enable follow-up research. We built two prototype tools *Stoat+*

1. For brevity, we use *framework exception* to indicate *framework-specific exception*, which can be any exception thrown from Android framework.

2. The percentage of how many issues has been closed by developers.
3. Stoat is available at *https://github.com/tingsu/stoat*.
4. Exlocator is available at *https://github.com/crashanalysis/ExLocator*.

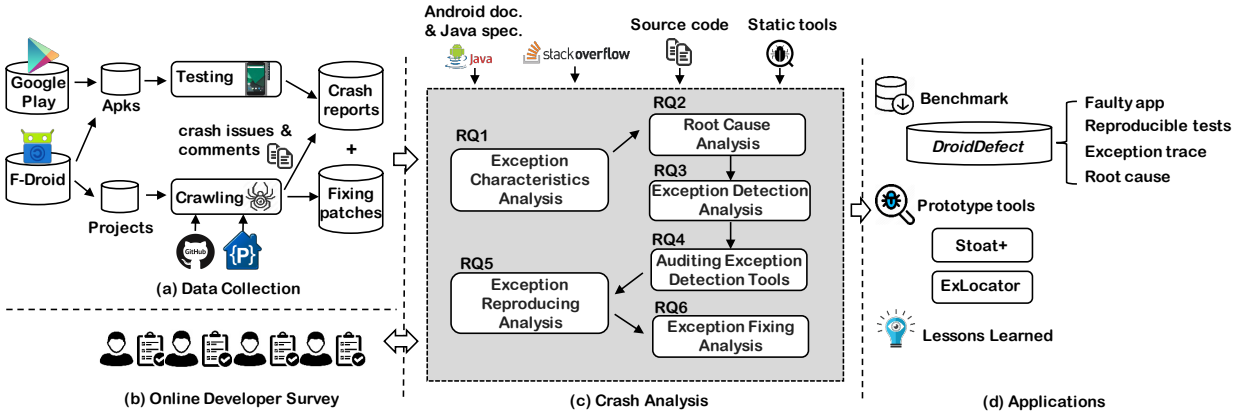

Fig. 1: Overview of our study

and *Exlocator* to improve bug detection and debugging, and summarized several lessons learned.

In our prior work [29], we investigated framework-specific exceptions in Android apps. In this journal version, we have made substantial extensions: (1) We additionally analyzed 8,760 exception bugs from 3,230 commercial apps from Google Play. It provided more observations on the characteristics of exception bugs, and validated the generability of our conclusion (Section 2.2 and 3.1). (2) We conducted an online survey among 135 Android app developers. It provided more insights from the developers' experiences and complemented our analysis results (Section 2.3, Section 3.1∼3.6 (RQ1∼RQ6)). (3) We revisited our research questions (*i.e.*, RQ1, RQ2, RQ4 and RQ6) in depth and analyzed together with the results from the online survey. For example, we additionally investigated the difficulties the developers face when analyzing root causes, the common fix practices, and the reasons of library exceptions, *etc*. (4) We additionally studied two new research questions, *i.e.*, the testing practices of exception bugs by developers (RQ3 in Section 3.3), and the reproducibility of exception bugs from the perspectives of both developers and testing tools (RQ5 in Section 3.5). It reveals the unsatisfactory points of existing testing tools, and the challenges that app developers and state-of-the-art tools face in reproducing exceptions, which have not yet been explored before. (5) We constructed the benchmark repository *DroidDefects*. It now contains 33 reproducible and 3,696 ground-truth exception bugs and the utility program for facilitating other research work. For each bug, we provided the faulty code version, the reproducible test, the exception trace and the explanation of root cause. *DroidDefects* can serve follow-up research work (Section 4). (6) We further illustrated more application domains of our study. We also extended our analysis on the empirical study and analysis results, and concluded with several lessons learned that were not identified before (Section 5 and 6). Importantly, our dataset, benchmark and tools were made publicly available at *https://github.com/tingsu/droiddefects*.

2 Preliminary and Study Preparation

2.1 Android Exception Model

The architecture of Android platform is composed of four layers, *i.e.*, application, framework, library and Linux kernel. Android apps run at the application layer. The Android framework APIs form the building blocks of apps.

```
java.lang.RuntimeException: Unable to resume activity {*}:
java.lang.NumberFormatException: Invalid double: “”
    at android.app.ActivityThread.performResumeActivity(…)
    ….
Caused by: java.lang.NumberFormatException: Invalid double:“”
    at java.lang.StringToReal.invalidReal(StringToReal.java:63)
    at java.lang.StringToReal.parseDouble(StringToReal.java:248)
    ….
```

Fig. 2: An example of RuntimeException trace

To provide different functionalities and services, Android reuses a number of libraries (*e.g.*, Apache, SSL, OpenGL). When an app crashes, a (Java) exception will be thrown from one of these three layers, which corresponds to application, framework or library exception.

Android apps (implemented in Java) inherit the exception model of Java, which has three kinds of exceptions. (1) RuntimeException, the exceptions that are thrown during the normal operation of the Java Virtual Machine when the program violates the semantic constraints (*e.g.*, null-pointer dereferences, divided-by-zero errors). (2) Error, which represents serious problems that a reasonable application should not try to catch (*e.g.*, OutOfMemoryError). (3) Checked Exception (all exceptions except (1) and (2)), these exceptions are required to be declared in a method or constructor's throws clause (statically checked by compilers), and indicate the conditions that a reasonable client program might want to catch. The programmers are responsible to handle RuntimeException and Error by themselves at runtime.

Fig. 2 shows an example of RuntimeException. The bottom part represents the *root exception*, *i.e.*, NumberFormatException, which indicates the root cause. Java uses *exception wrapping*, *i.e.*, one exception is caught and wrapped in another to propagate exceptions. In this case, RuntimeException in the top part wraps NumberFormatException. Note that the root exception can be wrapped by multiple exceptions, and the flow from the bottom to the top denotes the order of exception wrappings. An *exception signaler*, the first called method under the root exception declaration (*e.g.*, invalidReal in this case), is the method that throws the exception. To classify each exception, we referred to Android documentation [30] (API level 18) and the heuristic rules defined by prior work (Table II in [31]) according to the signaler's origin: (1) *Application Exception*: the signaler is defined in the application code. We can recognize it by the application's package name. (2) *Framework Exception*:

the signaler is defined in the Android framework, *i.e.*, from these packages: "android.*", "com.android.*", "java.*", and "javax.*". (3) *Library Exception*: the signaler is defined in the libcore in Android framework (*e.g.*, "org.apache.*", "org.json.*", "org.w3c.*") or third-party libraries used by the app. Note that, in this study, we do not consider native crashes caused by C++ exceptions, and do not consider Java exceptions caused by the bugs of Android framework itself.

2.2 Data Collection

2.2.1 App Subjects

We collected our app subjects from F-Droid [32] and Google Play Store [33]. We chose F-Droid due to three reasons. First, it is the largest repository of open-source Android apps. At the time of our study, it contains over 2,104 unique apps and 4,560 different releases (each app has 1∼3 recent releases), and maintains their metadata (*e.g.*, project addresses, history versions). Second, the apps have diverse categories (*e.g.*, Internet, Personal, Tools), covering different maturity levels of developers, which are the representatives of real-world apps. Third, all apps are hosted on GitHub, Google Code, SourceForge, *etc*, which makes it possible to access their source code and issue repositories. Additionally, we randomly selected 3,230 closed-source apps from Google Play, Google's Android app market, which has millions of commercial apps with diverse categories. We uniformly selected these apps from the top ten categories (*e.g.*, Education, Lifestyle, Business, Tools) [1], and each app has at least *one million installations*. These apps could be regarded as the representatives of commercial apps.

2.2.2 Exception Trace Collection

Table 1 summarizes the statistics of collected exception traces from hosting platforms (GitHub and Google Code) and testing tools. We applied testing tools on both F-Droid apps and Google Play apps to collect exceptions.

Issue Repositories. We collected exception traces from GitHub and Google Code since they host over 85% (2,174/2,549) F-Droid apps. To automate data collection, we implemented a web crawler to automatically crawl the issue repositories of these apps, and collected the issues that contain exception traces. In detail, the crawler visits each issue and its comments to extract valid exception traces. Additionally, it utilizes GitHub and Google Code APIs to collect project information such as package name, issue id, number of comments, open/closed time. We took about two weeks and successfully scanned 272,629 issues from 2,174 apps, and finally mined 7,764 valid exception traces (6,588 unique) from 583 apps.

Automated GUI Testing Tools. To test F-Droid apps (4,560 recent release versions of 2,104 apps) and Google Play apps (3,230 apps), we chose three state-of-the-art Android app testing tools with different strategies: Monkey [28] (random testing), Sapienz [11] (search-based testing), and Stoat [12] (model-based testing). Each tool is configured with default settings and each app is given 3 hours to thoroughly test on a single Android emulator. Each emulator is configured with Jelly Bean Android OS (SDK 4.3.1, API level 18). The evaluation is deployed on three physical machines (64-bit Ubuntu/Linux 14.04). Each machine runs 10 emulators in parallel. Since Sapienz and Stoat leverage code coverage to optimize test generation, we instrumented apps by Emma [34] or Jacoco [35] to collect coverage data.

This data collection phase took 6 months in total, and we finally detected 13,271 crashes (9,722 unique) for open-source apps, and 293,266 crashes (13,764 unique) for commercial apps. During testing, when an app crashes, the exception trace with bug-triggering inputs, screenshots, detection time, *etc*, are recorded to help our analysis.

Notably, for F-Droid apps, we find that the issue repositories of GitHub and Google Code only contain 545 unique crashes that were reported with stack traces, for the 4,560 recent release versions. These crashes only accounts for 5.6% of those detected by testing tools. This indicates these exception traces collected by testing tools can indeed effectively complement the mined exceptions.

2.2.3 Other Resource Collection

To help analysis, we also collected the most relevant posts with the most votes on Stack Overflow by searching key words with "Android", exception types and exception messages. We recorded the creation time, number of votes, number of answers, summary, *etc*. Finally, we mined 15,678 posts of various exceptions.

2.3 Online App Developer Survey

2.3.1 Questionnaire Design

To gain more understanding and validate our own analysis results on exception bugs, we conducted an online app developer survey. This survey aims to solicit Android app developers to share their experience of analyzing, testing, reproducing and fixing exception bugs. Table 2 presents the questionnaire of our study, which includes Q1∼Q17. Specifically, the survey is designed as two parts.

Part I: Background Information. We collected the background information of developers via Q1∼Q4. By these questions, we can filter invalid developers (*e.g.*, the survey only proceeds if the developer is aware of app exceptions), and get the survey results of different developer groups (*e.g.*, groups of developers with different experience levels, different app categories and countries).

Part II: App Exception Experiences and Practices. We collected developers' experiences and practices information via Q5∼Q17. We initially designed a number of questions according to our research questions RQ1∼RQ6, and sent them to three experienced Android app developers (with 5-year+ development experience) from Google, Tecent and Alibaba, respectively, for early feedback. We later refined these questions several rounds, and come up with Q5∼Q17. This design process aims to make the questions intuitive to developers and concentrate on those questions that both developers and researchers really concern.

For the developers who are aware of app exceptions, we provided three examples for each exception category to make sure the developers can fully understand the survey's purpose and related terminologies. Then, we presented Q5∼Q17 to systematically understand the developers' practices from different perspectives. Specifically, we collected information about (1) whether developers have encountered the three exception categories via Q5 and Q6 (*cf.* Section 3.1.1), (2) how developers understand framework exceptions via Q7∼Q9 (*cf.* Section 3.2.3), (3) how developers

TABLE 1: Statistics of collected crashes ("M.": Monkey; "Sa.": Sapienz; "St.": Stoat).

Sources	#Projects	#Crashes	#Unique Crashes
Platforms	2,174	7,764	6,588
(GitHub/Google Code)	(2,035/137)	(7,660/104)	(6,494/94)
F-Droid	2,104	13,271	9,722
(M./Sa./St.)	(4,560 versions)	(3,758/4,691/4,822)	(3,086/4,009/3,535)
Google Play	3,230	293,266	13,764
(M./Sa./St.)		(169,869/58,551/64,846)	(5,634/3,839/4,291)
Total	5,716 (1,792 overlap)	314,301	30,009

detect these exceptions in practice via Q10∼Q12 (*cf.* Section 3.3), (4) how developers reproduce these exceptions via Q13∼Q15 (*cf.* Section 3.5), and (5) how developers fix these exceptions via Q16∼Q17 (*cf.* Section 3.6). In particular, some questions (*e.g.*, Q5, Q6, Q7, Q16) aim to validate our analysis results; some questions (*e.g.*, Q9, Q12, Q13, Q14, Q15, Q17) aim to understand developers' experiences and practices; some questions (*e.g.*, Q7, Q9, Q12, Q14) are given with some options (summarized and refined according to our research experience and discussions with three senior developers), and an "Others" option to allow any additional comments.

2.3.2 Participants

To get sufficient number of responses from developers, we solicited the participants from three channels. First, we contacted 4,428 open-source app developers from GitHub and 1,226 commercial app developers from Google Play by scrawling their emails. Second, we invited the app developers in industry to distribute the survey within their companies and networks. These contacts are from Google, Tencent, Huawei, Alibaba and other IT companies. Third, we recruited app developers from Amazon Mechanical Turk [36] to participate in our survey. We paid 1.5 USD payment for each approved submission. Finally, we received valid responses from 135 professional app developers. Specifically, These developers come from 32 different countries across four different continents (Asia, Europe, North America, Oceania), and develop a diverse categories of apps (22 different categories). Among them, *Business*, *Tools*, *Education*, *Lifestyle*, *Entertainment* are the most popular categories. 10 developers are also involved in banking, insurance, financial apps, which emphasize more on robustness and safety. Among these 135 participants, 25 developers (18.5%) have less than 1-year experience, 67 developers (49.6%) have 1∼3 years' experience, 35 developers (25.9%) have 3∼6 years' experience, and 8 developers (6.0%) have more than 6 years' experience. Most of the developers, *i.e.*, 100 participants (81.5%) have more than 1-year development experience.

TABLE 2: Survey questionnaire of our study

ID	Question *Options/Types*
Part I:	**Background Information**
Q1	Experience in years as an Android developer/tester? *(<1 / 1∼3 / 3∼6 / >6 years)*
Q2	Working place? *(country, company/institution)*
Q3	App category developed? *(e.g., Education, Lifestyle, Business, Entertainment)*
Q4	Awareness of Android app exceptions? *(Yes/No)*
Part II:	**App Exception Experiences and Practices**
Q5	Ever encountered all the three exception categories? *(Yes/No)*
Q6	Pervasiveness of framework-specific exceptions? *(< 10%, 10%∼30%, 30%∼50%, 50%∼70%, > 70%)*
Q7	Ever encountered root causes of app exceptions? *(the 11 fault taxonomies, and "Others")*
Q8	How difficulty of understanding each root cause? *(Difficult / Medium / Easy)*
Q9	Main difficulties of diagnosing root causes? *(e.g., reproduction steps, bug environment, and "Others")*
Q10	Importance of resolving exceptions before release? *(Very important/Important/Normal/Not important/Ignored)*
Q11	Tools/Platforms to reveal app exception bugs? *(e.g., Monkey, UIAutomator, Lint, R&R tools)*
Q12	Unsatisfactory points of existing testing tools? *(e.g., manual efforts, false alarms, inefficiency, and "Others")*
Q13	Failure rate of reproducing exception bugs given reproducing steps? *(< 10% / 10%∼30% / 30%∼50% / >50%)*
Q14	Reasons affecting the reproducibility? *(e.g., concurrency, device models, system settings, and "Others")*
Q15	Pratices/Tools for improving reproducibility? *(open question)*
Q16	Popularity of common fix practices? *(the 5 common fix practices found by our study)*
Q17	Fix rate with only an exception trace? *(< 10%, 10%∼30% / 30%∼50% / 50%∼70% / >70%)*

3 Empirical Study

3.1 RQ1: Characteristics of Exceptions

3.1.1 Exception Category and Distribution

Based on the data collected in Section 2.2, Table 3 lists the exception categories of open-source and closed-source apps, and shows the number of the affected projects, occurrences, number of exception types and issue closing rate. Since Google Play apps do not have publicly available issue repositories, we only collected the closing rate for F-Droid apps. We can see two facts: (1) Framework exceptions are more pervasive and affect most of the apps. For example, 75.3% of open-source apps (revealed by the data of GitHub & Google Code) and 84.5% of closed-source apps (revealed by the data of testing tools) suffer from framework exceptions. In terms of exception occurrences, framework exceptions occupy more than half of all exceptions (50.8% for open-source apps revealed by GitHub/Google Code data, 74.1% for closed-source apps revealed by testing tools). This observation also conforms to the results of our survey question Q5 and Q6: 108 developers (80%), report they have encountered framework exceptions, and 88 developers (57.8%) report, in their experience, framework exception occupies around 30%∼50% (reported by 35 developers) and 50%∼70% (reported by 43 developers) among the three exception categories. (2) The closing rate of framework exceptions is 53%, which is relatively lower than those of the others (67% for application and 57% for library exception).

3.1.2 Locations of Framework Exception Manifestation

To understand framework exceptions, we grouped them by the class names of their signalers. In this way, we got more than 110 groups. To distill our findings, we further grouped these classes into 17 modules by following the insights of popular Android development tutorials [37], [38]. In our context, the classes in one *module* achieve either one general purpose or stand-alone functionality from developers' perspective. For example, we grouped the classes that manage the Android application model (*e.g.*, *Activities*, *Services*) into *App Management* (corresponding to android.app.*); the classes that manage app data from *content provider* and *SQLite* into *Database* (android.database.*); the classes that provide basic OS services, message passing and inter-process communication into *OS* (android.os.*). Other modules include *Widget* (UI widgets), *Graphics* (graphics tools that handle UI drawing), *Fragment* (one special visual element), *WindowsManager* (manage window display), *etc*.

TABLE 3: Statistics of the exceptions crawled from GitHub & Google Code and collected by testing tools on F-Droid and Google Play apps (classified into *Application Exception*, *Framework Exception*, and *Library Exception*, respectively).

Source	Exception Category	#Projects	Occurences	#Types	Closing Rate
F-Droid (GitHub & Google Code)	Application	268 (45.8%)	1552 (23.6%)	88 (34%)	67%
	Framework	441 (75.3%)	3,350 (50.8%)	127 (50%)	53%
	Library	253 (43.2%)	1,686 (25.6%)	132 (52%)	57%
F-Droid (Testing tools)	Application	1,869 (50.9%)	4,017 (41.3%)	35 (35.0%)	-
	Framework	2,400(65.3%)	5,072 (52.2%)	62 (62.0%)	-
	Library	366 (10.0%)	633 (6.5%)	44 (44.0%)	-
Google Play (Testing tools)	Application	389 (23.4%)	1,199 (14.4%)	20 (27.8%)	-
	Framework	1,405 (84.5%)	6,205 (74.1%)	44 (61.1%)	-
	Library	402 (24.2%)	965 (11.5%)	40 (55.6%)	-

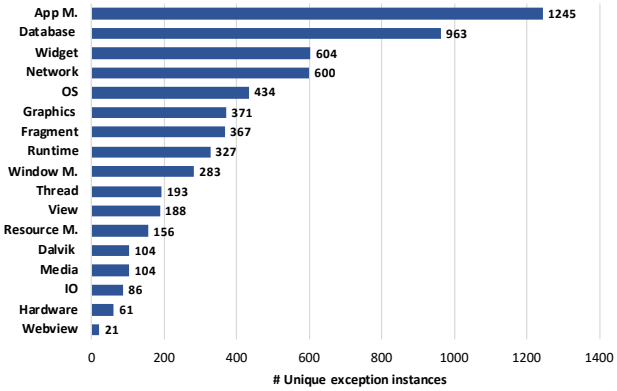

Fig. 3: Exception-proneness of Android framework modules in terms of unique instances (*M.* = Management)

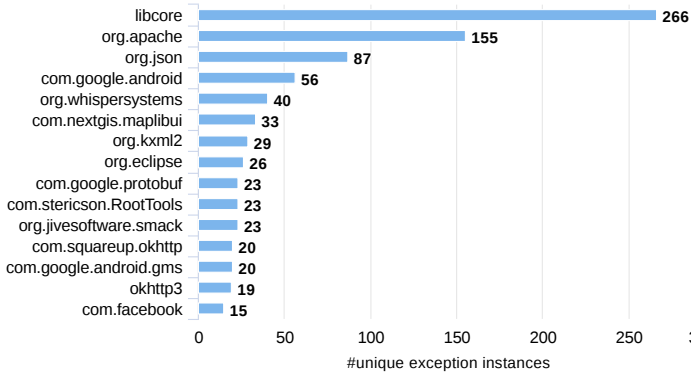

Fig. 4: Top 15 exception-prone libraries in terms of unique instances based on the data in Table 3.

Fig. 3 shows the exception-proneness[5] of Android framework modules in terms of unique exception instances. We find *App Management*, *Database* and *Widget* are the top 3 exception-prone modules. In *App Management*, the most common exceptions are ActivityNotFound (due to no activity is found to handle a given intent) and IllegalArgument exceptions (due to improper registering/unregistering Broadcast Receiver in the activity's callbacks). Surprisingly, although Activity, Broadcast Receiver and Service are the basic building blocks of apps, developers make the most number of mistakes on them.

As for *Database*, CursorIndexOutOfBounds, SQLiteException, SQLiteDatabaseLocked account for the majority, which reflect the various mistakes of using *SQLite*, the default database of Android. As for the other modules, we find: (1) improper use of ListView with Adapter throws a large number of IllegalState exception (account for 47%) in *Widget*; (2) In *OS*, SecurityException, IllegalArgument, NullPointer are the most common ones. (3) improper use of *Bitmap* causes OutOfMemoryError (48%) in *Graphics*; (4) improper handling callbacks of *Fragment* brings IllegalState (85%) in *Fragment*; improper showing or dismissing dialogs triggers BadTokens (25%) in *WindowManager*.

3.1.3 Locations of Library Exception Manifestation

To investigate the library exception, we used the exception data collected in Table 3. We grouped these exceptions by the class names of their signalers, and integrated the exceptions that are thrown from the same library. We finally got 100+ exception-prone libraries. Fig. 4 shows the top 15 libraries in terms of number of unique exception occurrences.

We find *libcore*, *org.apache*, and *org.json* are the three most exception-prone libraries, which are in fact the most basic ones and more frequently used than the others.

We further randomly selected 10 library exceptions from each of these top 15 libraries, and analyzed the root causes. We find that although these libraries provide different functionalities, their exceptions still have some common root causes. For example, most of exceptions are due to the misuse of APIs, *e.g.*, giving incorrect parameter values/formats, failing to validate specific resources (*e.g.*, network) before use. Some exceptions are caused by the API incompatibility issues [39] between the Android SDK/app and the library version, lack of specific hardware support or permissions [40]. Only a small portion of exceptions are due to the bugs of libraries themselves. These observations reveal that library exceptions do share similarity with framework exceptions (detailed in Section 3.2) in terms of common root causes. The Android framework can be actually viewed as a basic "library" that forms the building blocks of Android apps. In this paper, *we focus on investigating framework exceptions*. Different apps may use different libraries. Thus, giving a thorough analysis of library exceptions is not possible in this work alone. Thus, we leave it as future work. We have not given the manifestation locations of application exceptions, since these exceptions can be thrown from arbitrary locations at the app code level. We inspected a number of application exceptions, but most of them were generic programming errors. Thus, we do not give further exploration on application exception in this study.

Answer to RQ1: *Framework exceptions are more pervasive than the other two exception categories, among which App Management, Database and Widget are the three most exception-prone modules for developers. Library exceptions are similar with framework exceptions in the terms of root causes.*

5. In our context, exception-proneness indicates how often developers may misuse specific framework or library functionalities, and does not indicate the correctness of Android framework or libraries themselves. Specifically, these misuses manifest themselves as exceptions.

3.2 RQ2: Taxonomy of Framework Exceptions

This section characterizes the framework exceptions and classifies them into different categories based on their root causes. According to ISTQB [41], "Root cause is a source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed." Specifically, in our context, we define *root cause*, from the view of developers, is the *initiating* cause [42] of either a condition or a causal chain that leads to a visible exception bug. Section 3.2.1 explains how we analyze and abstract these framework exceptions into different categories. Section 3.2.2 illustrates these categories with concrete examples.

3.2.1 Exception Analysis Method

First, we collected 8,243 framework exceptions and partitioned them into different *exception buckets*. Each bucket contains the exceptions that share the similar root cause. Specifically, we used the exception type, message and signaler to approximate the root cause. We also removed app specific information in the exception message to scale the partition. For example, the exception in Fig. 2 is labeled as (NumberFormatException, "*invalid double*", invalidReal). Here, we removed the empty string from the original exception message. We finally got 2,016 buckets, and the top 200 buckets contain over 80% of all exceptions. The remaining buckets have only 5 exceptions or fewer in each of them. Therefore, we focus on the top 200 buckets.

Second, we randomly selected a number of exceptions from each bucket, and used three complementary resources to facilitate root cause analysis: (1) *Exception-Fix Repository*. We set up a repository that contains pairs of exceptions and their fixes. In particular, (i) from 2,035 Android apps hosted on GitHub, we mined 284 framework exception issues that are closed with corresponding patches. To set up this mapping, we checked each commit message by identifying the keywords "fix"/"resolve"/"close" and the issue id. (ii) We manually checked the remaining issues to include valid ones that are missed by the keyword rules. We finally got 194 valid issues. We investigated each exception trace and its patch to understand the root causes. (2) *Exception Instances Repository*. From the 9,722 exceptions detected by testing tools (see Table 3), we filtered out framework exceptions, and linked each of them with its exception trace, source code, bug-triggering inputs and screenshots. When an exception type under analysis is not included or has very few instances in the exception-fix repository, we referred to this repository to facilitate analysis by using available reproducing information. (3) *Technical Posts*. For each exception type, we referred to the posts from Stack Overflow collected in Section 2.2.3 when needing more information from developers and validating our understanding.

Finally, we analyzed 86 distinct exception types, which covers 84.6% of all framework exceptions[6], and distilled 11 common fault categories. Specially, we abstracted the common faults by the three steps. First, we read the official Android documentation and popular developer tutorials to identify and understand Android's important mechanisms

6. We found 13.2% of all exceptions are NullPointerException, which are caused by null pointer dereferences and highly related to the specific logic of each app. Thus, we did not inspect this generic exception type in our analysis.

```
class DataRetrieverTask extends AsyncTask<String, ...> {
  private BankEditActivity context;
  protected Void doInBackground(final String... args) {
    ... //update bank info via the remote server
  }
  protected void onPostExecute(final Void unused) {
    ... //show the update progress
    AlertDialog.Builder builder = new  AlertDialog.Builder(context);
    ... //set dialog message
    AlertDialog alert = builder.create();
+   if(!context.isFinishing()) {
      alert.show();
+   }
  }}
```

Fig. 5: Bankdroid Issue #471 (Simplified)

(*e.g.*, activity lifecyle, single-GUI-thread model), components (*e.g.*, activity, service, thread, database), and features (*e.g.*, XML-based UI design, API compatibility). Second, we inspected each exception bug to understand its own root cause by using the resources stated above. Third, we abstracted the root cause into which mechanism it violates, or which component or feature it fails in. By these information, we classified an exception into one specific fault category, which is named after specific mechanism errors (*i.e.*, Component Lifecycle Error, UI Update Error, Framework Constraint Error), component usage errors (*i.e.*, Concurrency Error, Database Management Error), feature errors (*i.e.*, API Updates and Compatibility, Memory/Hardware Error, XML Design Error) or generic errors (Resource Not Found Error, API Parameter Error, Indexing Error).

3.2.2 Taxonomy

• ***Component Lifecycle Error***. Each Android component has its own lifecycle and is required to follow the prescribed lifecycle paradigm, which defines how the component is created, used and destroyed [43]. For example, Activity provides six core callbacks to allow developers to be aware of its current state. If developers improperly handle the callbacks or miss state-checking before some tasks, the app can be fragile considering the complex environment interplay (*e.g.*, device rotation, network interruption). *Bankdroid* [44] (Fig. 5) is a Swedish banking app. It utilizes a background thread DataRetrieverTask to perform data retrieval, and pops up a dialog to inform that the task is finished. However, if the user presses the back button on BankEditActivity (which starts DataRetrieverTask), the app will crash when it tries to pop up a dialog. The reason is that the developers fail to check BankEditActivity's state (in this case, *destroyed*) after the background task is finished. The bug triggers a BadTokenException and was fixed in revision 8b31cd3 [45]. Besides, Fragment [46], a reusable class implementing a portion of Activity, has much more complex lifecycle. It provides 12 core callbacks to manage its state transition, which makes lifecycle management more challenging, *e.g.*, state loss of Fragments, attachment loss from its activity.

• ***UI Update Error***. Android enforces the single GUI thread model. A UI thread is in charge of dispatching events and rendering user interface. Each app owns one UI thread and should offload intensive tasks to background threads to ensure responsiveness. *cgeo* [47] (Fig. 6) is a popular full-featured client for geocaching. When refreshing cacheList (cacheList is associated with a ListView via an ArrayAdapter), the developers query the database and substitute this list with new results (via clear() and addAll()) in doInbackground. However, the app crashes when it tries to

```
private List<Geocache> cacheList = new ArrayList<>();
private CacheListAdapter adapter =
  ... // adapter binds cacheList and ListView
  new AsyncTask<Void, Void, Void>() {
   protected Void doInBackground(final Void... params){
      //run in the background thread
    final Set<Geocache> cacheListTmp = ... //query database
-   if (CollectionUtils.isNotEmpty(cacheListTmp)){
-     cacheList.clear();
-     cacheList.addAll(cacheListTmp);
-   }
  }}
```

Fig. 6: cgeo Issue #4569 (Simplified)

```
public class GSMService extends LocationBackendService{
 protected Thread worker = null;
 ... //start the service
 worker = new Thread() {
   public void run() {
+    Looper.prepare();
     final PhoneStateListener listener =
       new PhoneStateListener() {
         ... //callbacks to monitor phone state change
     };
  }
  worker.start();
}
```

Fig. 7: Local-GSM-Backend Issue #2 (Simplified)

refresh the list. Because cacheList is maintained by the UI thread, which internally checks the equality of item counts between ListView and cacheList. But when a background thread modifies cacheList, the checking will fail and an exception will be thrown. The developer fixed it by moving the refreshing operations into onPostExecute, which instead runs in the UI thread (in revision d6b4e4d [48]).

• ***Framework Constraint Error***. Android defines a number of constrains when using its framework to build an app. For example, *Each Handler [49] instance must be associated with a single thread and the message queue of this thread [50]*. Otherwise, a runtime exception will be thrown. *Local-GSM-Backend* [51] (Fig. 7), a popular cell-tower based location lookup app, uses a thread worker to monitor the changes of telephony states via PhoneStateListener. However, the developers are unaware that PhoneStateListener internally maintains a Handler instance to deliver messages [52], which requires setting up a message loop in worker. They later fixed it by calling Looper#prepare() (in revision 07e4a759 [53]). Other constraints include performance consideration (avoid performing network operations in the main UI thread [54], permission consideration (require runtime permission grant for dangerous permissions [55] since Android 6.0, otherwise SecurityException) and *etc*.

• ***Concurrency Error***. Android provides a number of asynchronous programming constructs, *e.g.*, AsyncTask, Thread, to concurrently execute intensive tasks. However, improper handling them may cause data race [56] or resource leak [57], and even app crashes. Nextcloud Notes [58], a cloud-based notes-taking app, automatically synchronizes local and remote notes. It attempts to re-open an already-closed database, causing app crash [59]. The exception can be reproduced by executing two steps repeatedly: (1) open any note from the list; (2) close the note as quickly as possible by pressing back-button. The app creates a new NoteSyncTask every time when a note sync is requested, which connects with the remote sever and updates the local database by calling updateNote(). However, when there are multiple update threads, such interleaving may happen and crash the app: *Thread A* is executing the update, and *Thread B* gets the reference of the database; *Thread A* closes the database after the task is finished, and *Thread B* tries to update the closed database. The developers fixed this exception in revision aa1a972 [60] by leaving the database unclosed (since SQLiteDatabase already implemented thread-safe database access mechanism).

```
 public void onCreate(SQLiteDatabase db) {
   ... //create database tables
   db.execSQL(CREATE_FRIENDS_TABLE);
 }
 public void onUpgrade(SQLiteDatabase db, int oldVersion,
   int newVersion) {
   // upgrade database
   if (oldVersion < 5) { ... }
   if (oldVersion < 6) {
-    db.execSQL("create table temp_table as
           select * from " + TABLE_FRIENDS);
-    db.execSQL("drop table " + TABLE_FRIENDS);
+    db.execSQL(CREATE_FRIENDS_TABLE);
     ...
  }}
```

Fig. 8: Atarashii Issue #82 (Simplified)

TABLE 4: Statistics of 11 common fault categories, sorted by *closing rate* (collected from GitHub) in descending order ("Occ.": Occurrences, "S.O.": Stack Overflow).

Category (Name for short)	#Occ.	#S.O. posts	Closing Rate
API Updates and Compatibility (**API**)	68	60	93.3%
XML Layout Error (**XML**)	122	246	93.2%
API Parameter Error (**Parameter**)	820	819	88.5%
Framework Constraint Error (**Constraint**)	383	1726	87.7%
Others (Java-specific errors)	249	4826	86.1%
Index Error (**Index**)	950	218	84.1%
Database Management Error (**Database**)	128	61	76.8%
Resource-Not-Found Error (**Resource**)	1303	7178	75.3%
UI Update Error (**UI**)	327	666	75.0%
Concurrency Error (**Concurrency**)	372	263	73.5%
Component Lifecycle Error (**Lifecycle**)	608	1065	58.8%
Memory/Hardware Error (**Memory**)	414	792	51.6%

• ***API Updates and Compatibility***. Android features fast API updates. For example, Service should be started explicitly since Android 5.0; the change of the comparison contract of Collections#sort() [63] since JDK 7 crashes many apps due to the developers are unaware of this. It also has device fragmentation issues, which were already investigated by prior work [64], [65]. For example, problematic deriver implementation, non-compliant OS customization, and peculiar hardware configuration may cause compatibility issues.

• ***Memory/Hardware Error***. Android devices have limited resources (*e.g.*, memory). Improper using of resources may cause app crashes. For example, OutOfMemoryError occurs if loading too large Bitmaps; RuntimeException appears when MediaRecorder#stop() is called without valid au-

dio/video data received.

• ***XML Design Error***. Android supports UI design and resource configuration in the form of XML files. Although IDE tools have provided much convenience, mistakes still exist, *e.g.*, misspelling custom UI control names, forgetting to escape special characters (*e.g.*, "$", "%") in string texts, failing to specify correct resources in colors.xml and strings.xml.

• ***Resource Not Found Error***. Android apps heavily use external resources (*e.g.*, databases, files, sockets, third-party apps and libraries) to accomplish tasks. Developers make this mistake when they fail to check their availability.

• ***API Parameter Error***. Developers make such mistakes when they fail to consider all possible input contents or formats, and feed malformed inputs as the parameters of APIs. For example, they directly use the results from SharedPreference or database queries without any checking.

• ***Indexing Error***. Indexing error happens when developers access data, *e.g.*, *database*, *string*, and *array*, with a wrong index value. One typical example is the CursorIndexOutOfBounds exception caused by accessing database with incorrect cursor index.

3.2.3 Understanding Root Causes from Developers

To further validate the results of root cause analysis, we surveyed the developers with three questions. In the first question (Q7), we aimed to check the correctness and completeness of root causes. We listed the 11 root causes (accompanied with 2∼3 issue examples) that can cause framework exceptions, and asked developers to choose anyone that he or she has ever encountered. We also provided an additional option "*Others*" for developers to fill in any root causes we may have missed in our study. In the second question (Q8), we aimed to understand how difficult the developers may feel when resolving the exceptions with these root causes (including the effort to inspect the exception message, understand the root case, and locate the faulty code). We gave them the three options, *i.e.*, *Difficult*, *Medium*, and *Easy*, to rate each root cause. In the third question (Q9), we aimed to understand the difficulties of diagnosing root causes. We gave the four options, *i.e.*, *understand exception type and message*, *get the reproduction steps* (the user actions to trigger the exception), *get the bug environment* (*e.g.*, app version, device info), *understand the principles or usages of specific Android APIs*, and an additional option "*Others*".

The responses of the first question support our root cause analysis. All of the 11 root causes were encountered by the developers. Specifically, Framework Constraint Error (encountered by 62 developers (45.9% of all developers)), API Updates and Compatibility Error (60 developers (44.4%)), Lifecyle Error (53 developers (39.3%)), UI Update Error (53 developers (39.3%)) are the four most commonly encountered errors reported by developers. This finding conforms to our analysis results. In Table 4, "#Occ." denotes the exception occurrences of each root cause among the 8,243 framework exceptions. We can see, besides those "trivial" errors such as Resource-Not-Found Error, Index Error and API Parameter Error, app developers are indeed more likely to make Android specific errors, *e.g.*, Lifecycle Error, Memory/Hardware Error, Framework Constraint Error. Some developers also mentioned some exception instances in the "*Others*" option. For example, one developer mentioned

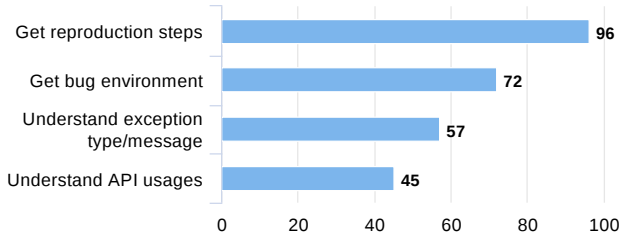

Fig. 9: Difficulties of root cause analysis

improperly using of Android APIs, which was categorized into the API Parameter Error category; another developer mentioned *not properly handling the state of the listeners for sensors*, which was categorized into the Framework Constraint Error. Additionally, 42 developers (31.1% of all developers) mentioned Android system errors (*i.e.*, the bugs of Android framework itself) can also lead to framework exceptions, which is indeed true but out of our scope.

In the second question, we find developers have different assessments on the difficulties of these root causes according to their experience. Resource-Not-Found Error, API Parameter Error, Index Error, and XML Error were the top four most *Easy* errors rated by 50.4%, 48.1%, 44.4%, 43% of all developers, respectively, since these errors are usually induced by trivial human mistakes and easy to fix. On the other hand, Memory/Hardware Error, Concurrency Error, and API Updates and Compatibility Error were the top three most *Difficult* errors rated by 46.7%, 34.8%, 29.6% developers, respectively, because these errors are notoriously difficult to debug [56], [66]. As for Database Management Error, UI Update Error, Framework Constraint Error, Lifecycle Error, almost half of participants, *i.e.*, 51.8%, 48.1%, 46.7%, and 46.7% of all developers, respectively, rated them as *Medium*. This finding also conforms to our observation on Stack Overflow. In Table 4, "#S.O. posts" counts the number of Stack Overflow posts on discussing these faults. We can see developers indeed discuss more on Android Framework Constraint Error and Lifecycle Error.

Fig. 9 shows the responses for Q9. We can see 96 developers (71.1% of all developers) reached the consensus that the most difficult point is to get the reproduction steps, which is quite crucial for diagnosing the root cause. The second difficult point, mentioned by 72 developers (53.3%), is to get the bug environment. 57 developers (42.2%) confirmed the exception type and message sometimes also bring confusions, while 45 developers (33.3%) reported some specific Android APIs usages or features also affect the understanding of root causes. We received 5 answers from the "Others" option, but all of them can be grouped into the previous four difficulties due to similarity. Thus, we believe these four difficulties are the most typical ones.

Answer to RQ2: *We distilled 11 fault categories of framework exceptions. Developers make more mistakes on Lifecycle Error, Memory/Hardware Error and Framework Constraint Error. Developers feel it difficult to resolve Concurrency Error, Memory/Hardware Error, and API Updates and Compatibility Error. Getting reproduction steps and bug environment are the two most difficult problems when diagnosing root causes.*

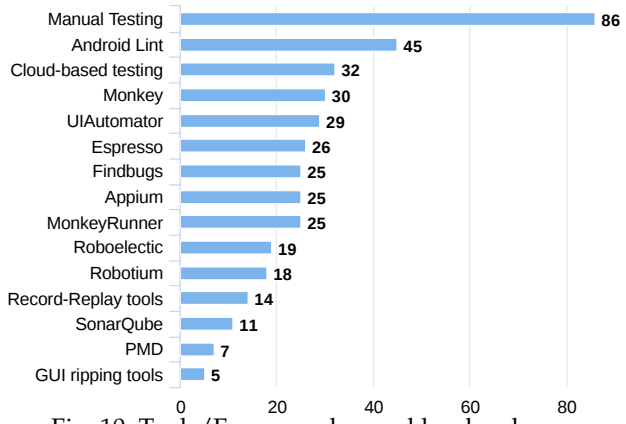

Fig. 10: Tools/Frameworks used by developers

3.3 RQ3: Detecting Exception Bugs

This section investigates the testing practices against exception bugs from developers' perspective. Different from prior surveys [9], [10], [67] on how developers test Android apps, our investigation focuses on how developers detect these exception bugs that can lead to crashes. Specifically, we aim to understand (1) the importance of detecting exception bugs, (2) the commonly-used tools to detect exception bugs, and (3) the unsatisfactory points of these tools. This section motivates our deep investigation on these bug detection tools in Section 3.4 (RQ4) and Section 3.5 (RQ5).

3.3.1 Tools for Detecting Exception Bugs

For the question Q10 "*Do you think it is important to detect (and resolve) exception bugs before releasing your apps*?", the responses were very consistent: 56.3% developers chose *Very Important*, 34.8% developers chose *Important*, and 8.9% developers chose *Normal*. This result indicates that detecting exception bugs is indeed one of top priorities for developers.

In practice, many bug detection tools or frameworks are available to help detect potential app exceptions. Fig. 10 shows the tools that are used by app developers to test or check exception bugs (the responses of Q11). These tools can be categorized into different groups by their principles. For example, Monkey [28] is a random fuzzing tool that tests apps by emitting a stream of random input events; MonkeyRunner [68] is an API-based testing tool that tests apps/devices from functional or framework level. Other tools include unit/integration testing frameworks (*e.g.*, Roboelectic, Espresso, UIAutomator), script-based testing frameworks (*e.g.*, Robotium, Appium), R&R (record & replay)-based tools, cloud-based testing service (*e.g.*, Google Firebase, Microsoft Xamarin) and static checking tools (*e.g.*, Findbugs, Android Lint, PMD, SonarQube).

We can see manual testing is still the most preferable way of 86 developers (63.7%) to find exception bugs. Android Lint is the most commonly-used tool by 45 developers (33.3%) to automatically scan app bugs, which is more popular than other static checking tools (*i.e.*, FindBugs, PMD, SonarQube). 74 developers (54.8%) preferred using AndroidJUnitRunner-based unit and integration testing frameworks (*e.g.*, Espresso and UIAutomator), and 32 developers (23.7%) resorted to cloud-based testing services (*e.g.*, Google Firebase). We also notice only a few (5 developers) use automated GUI ripping tools. Sapienz [11] and Stoat [12], the two state-of-the-art tools, were used. Different from all the other tools, these GUI ripping tools are developed and maintained by researchers to achieve automated app testing.

3.3.2 Unsatisfactory Points of Existing Tools

In the survey, we further asked the developers Q12 "*which points do you think the tools you used are still not satisfactory for detecting exception bugs?*". From the responses, we have several findings. (1) 67 developers (49.6%) complained about the demanding human efforts required by *manually writing tests and setting up the testing environment*. Manual testing and those non-fully automatic testing methods (*e.g.*, MonkeyRunner, AndroidJUnitRunner-based and script-based testing frameworks and R&R tools) all need manual efforts. (2) The inefficiency of uncovering exception bugs is another major concern of 64 developers (43.7%). They reported some tools either *cost too much testing time* (*e.g.*, R&R tools) or *miss bugs* (*e.g.*, Android Lint and other static checking tools). (3) 56 developers (41.5%) complained that *even if the tool finds an exception, the generated test cannot guarantee to reproduce the bug*. This indicates the bug reproducibility problem of mobile apps. Monkey and cloud-based testing service are the two typical methods that have this issue. For example, a Monkey test is a stream of low-level events (based on the device screen coordinates), which may probably fail to reproduce the bug if the screen size changes. Section 3.5 gives a deep investigation of this problem. (4) 47 developers (34.8%) mentioned that the static checking tools (*e.g.*, Lint) and R&R tools can bring false alarms, *i.e.*, *the reported issues are not real bugs*. This issue usually wastes developers' time for inspecting them. (5) 43 developers (31.9%) reported that *some tools fail to consider various environment* (*e.g.*, screen rotation, network stability, different geographic locations, heavy memory/CPU usage), which are quite crucial for testing the usability and robustness of mobile apps. (6) 42 developers (31.1%) hoped the testing or checking tools could generate tests for verification or generate more readable tests for debugging. For example, some developers desired to get more readable tests from Monkey. Developers have not provided other comments in the "Others" option.

Answer to RQ3: *Most developers agree detecting exception bugs is crucial, however, manual testing is still the most preferable testing method. Although different bug detection tools are used, developers still have unsatisfactory points,* e.g., *high manual efforts, insufficient bug detection, low reproducibility rate, many false positives, lack of considering environment etc.*

3.4 RQ4: Auditing Automatic Bug Detection Tools

Informed by the study of RQ3, this section aims to investigate the effectiveness of bug detection tools. As revealed by RQ3, most of the bug detection tools require human assistance (*e.g.*, writing tests). We note two groups of tools, *i.e.*, dynamic testing and static analysis tools, can fully automate app exception checking. However, our previous investigation on the four static analysis tools, *i.e.*, Lint, FindBugs, PMD, SonarQube, shows these tools are almost ineffective in detecting framework exceptions due to the lack of specific checking rules [29]. Unfortunately, these tools have not provided handy APIs or command line options to accept customized checking rules, and require considerable code-level extensions. Thus, we decided not to

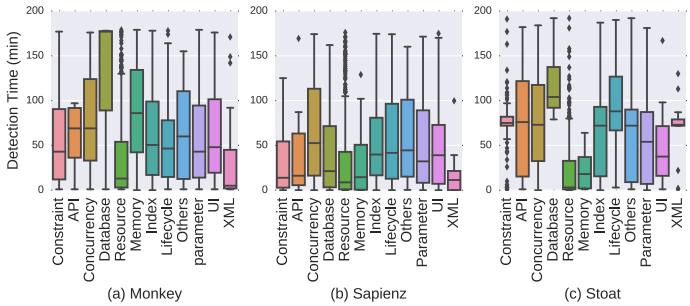

Fig. 11: Detection time of exceptions by each tool

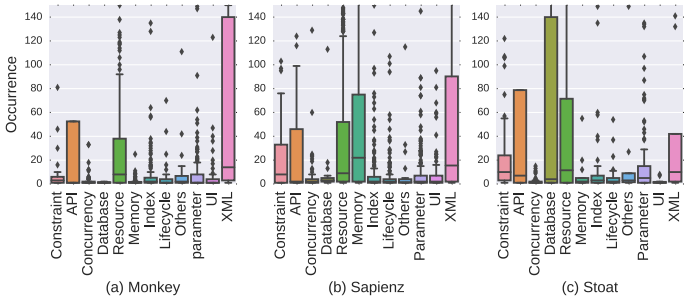

Fig. 12: Occurrences of exceptions by each tool

include them in this evaluation, otherwise the results could be unfair. Section 5 discusses plausible ways of improving static analysis tools. We do not consider the cloud-based testing services as well, which are pay-by-use and not convenient to conduct large-scale evaluation on thousands of apps. Therefore, we only focus on dynamic testing tools, and evaluate them on the framework exceptions categorized in Section 3.2. We selected 3 state-of-the-art dynamic testing tools, *i.e.*, Monkey [28], Sapienz [11], and Stoat [12]. The survey in Section 3.3 shows these tools are used by a number of real app developers (35 developers ever used). More importantly, recent studies [69], [70] show, these tools are proved to be the most effective on both open-source and commercial apps, and have found hundreds of previously-unknown crash bugs in well-tested apps.

We applied dynamic testing tools on each of 2,104 apps with the same configurations in Section 2.2.2. We observed that they could detect many framework exceptions. To understand their abilities, we used two metrics[7]. (1) *detection time* (the time to detect an exception). Since one exception may be found multiple times, we used the time of its first occurrence. (2) *Occurrences* (how many times an exception is detected during a specified duration). Fig. 11 and Fig. 12, respectively, show the detection time and occurrences of exceptions by each tool grouped by the fault categories.

From Fig. 11, we can see the abilities of these tools vary across different fault categories. But we also note some obvious differences. For example, following the guidelines of statistical tests [71], we used Mann-Whitney U test [72], a non-parametric statistical hypothesis test for independent samples, to compare the detection time of some specific fault categories across three tools. We find Sapienz is better at database errors (*i.e.*, use significantly less testing time) than Monkey (*p-value*=0.02, and standardized effect size is *medium* (0.41)) and Stoat (*p-value*=$0.05*10^{-4}$, and standardized effect size is *large* (0.65)). One important reason is that Sapienz implements a strategy, *i.e.*, fill strings in EditTexts, and then click "OK" instead of "Cancel" to maximize code coverage, which is more likely to trigger database operations. Monkey and Sapienz, respectively, are better at lifecycle errors than Stoat (*p-value*s are, respectively, 0.002 and 0.001, and standardized effect sizes are, respectively, *medium* (0.35) and *small* (0.25)). Because both Monkey and Sapienz emit events very quickly without waiting for the previous ones to take effect, *e.g.*, open and quickly close an activity without waiting for the activity finishes its task.

In addition, we note concurrency errors are non-trivial for all three tools, *i.e.*, Monkey, Sapienz and Stoat. But their detection times are not significantly different according to our statistical test. The medians of their detection times are, respectively, 52, 69 and 58 minutes. In Fig. 12, the occurrences of API compatibility, Resource-Not-Found and XML errors are much more than those of many other fault categories across three tools. It indicates these errors are easier to be repeatedly detected. But, on the other hand, Concurrency, Lifecyle, UI update errors are more difficult to be repeatedly detected, regardless of the testing strategies of these tools. The main reason is that these errors contain more non-determinism (interacting with threads).

After an in-depth inspection, we find that some Database errors are hard to trigger because the app has to construct an appropriate database state (*e.g.*, create a table or insert a row, and fill in specific data) as the precondition of the bug, which may take a long time. As for Framework Constraint Error, some exceptions require special environment interplay. For example, InstantiationException of Fragment can only be triggered when a Fragment is destroyed and recreated. To achieve this, a testing tool needs to change device rotation at an appropriate timing (when the target Fragment is on the screen), or pause and stop the app by switching to another one, and stay there for a long time (let Android OS kill the app), and then return back to the app. Concurrency bugs (*e.g.*, data race) are hard to trigger since they usually need right timing of events.

Answer to RQ4: *Dynamic testing tools are less effective in detecting concurrency, database and lifecycle errors. Different testing strategies have a big impact on the bug detection ability against different types of framework exceptions. More effective dynamic testing strategies are demanded to help detect framework exceptions.*

3.5 RQ5: Reproducibility of Exception Bugs

This section investigates the reproducibility of exception bugs, which is crucial for bug diagnosing and fixing. Android apps are event-centric programs and run in complex environment. Typically, the bug-triggering inputs are described as a few reproducing steps (in the form of natural language by humans or event sequences generated by testing tools) and contextual conditions (*e.g.*, device models, network status, and other device settings [73]). Prior work improves the reproducibility of crash bugs by augmenting bug reports [74], [75], [76], [77], translating a bug report (written in natural language) into an executable UI test [78], [79], and leveraging crowd-sourced monitor-

7. We do not present the results of trace length, since we find the three tools cannot dump the exact trace that causes a crash. Instead, they output the whole trace, which cannot reflect their detection abilities.

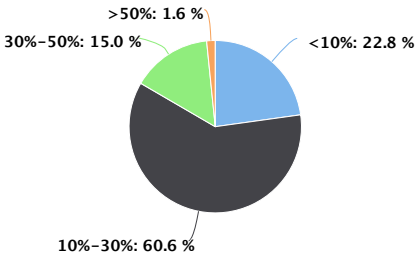

Fig. 13: Percentage of cases in failing to reproduce exceptions even if the reproduction steps are given.

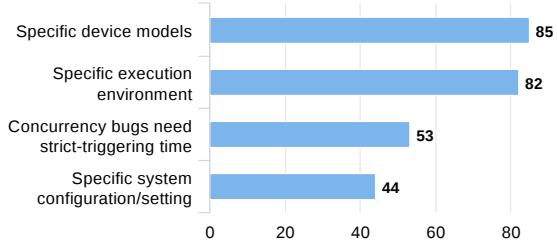

Fig. 14: Difficulties of reproducing exceptions.

ing [80]. However, to our knowledge, no previous efforts has investigated the reproducibility of exception bugs, from these two perspectives: (1) how do app developers, and (2) how do automatic testing tools, perform in reproducing bugs, which this section will explore.

3.5.1 Perspective of App Developers

In our survey, we asked developers Q13 "*In your experience, given the reported reproducing steps, how much percentage of cases in which you still* ***cannot*** *reproduce the crash exception?*". We gave them the four options, *i.e.*, *<10%*, *10%∼30%*, *30%∼50%*, and *>50%*. Fig. 13 shows the responses. We find 82 developers (60.6%) reported they fail to reproduce 10%∼30% exception bugs, which were not ignorable. Further, 20 developers (15%) could not successfully reproduce 30%∼50% exception bugs, and 2 developers even could not reproduce over 50% exceptions. 31 developers (22.8%) chose the option *<10%*. Further, from the responses of 43 senior developers (with over 3 years working experience), we find only 11 of them (25.6%) choose the option *<10%*, which indicates experienced developers also face difficulties in reproducing bugs. Based on the above observations, although developers in fact are quite familiar with their own apps and implementations, we can see reproducing exception bugs is still difficult for human developers, even if the reproduction steps are given.

We further asked developers Q14 "*If you cannot reproduce the crash exception, in your experience, which reasons may affect the reproducibility*?". Five options are provided: (A) *concurrency or asynchronous bugs (e.g., data race)*, (B) *specific running environment (e.g., low memory, external file access, usage of specific third-party library)*, (C) *specific device models (e.g., framework API version, OS customization)*, (D) *specific system configurations or settings (e.g., WiFi/4G, GPS on/off, enable/disable specific developer options)*, and (E) Others (for any developers' comments). The first four options were distilled from three sources: (1) the app developers' comments and discussions from GitHub issue repositories when they resolve bug reports, (2) our own experience of reproducing bugs during our own research [29], [81], [79], and (3) the previous work on bug reproduction [76], [80], [78].

Fig. 14 shows the results. 85 developers (63%) selected (C). They indicated different API versions and vendor models could affect the reproducibility because the platform where the apps are developed is usually different from the one where the apps are used. 82 developers (60.7%) chose (B). They indicated some specific execution environment (*e.g.*, heavy system load, external file access, *etc*) may affect the reproducibility. The developers felt difficult to record and restore the exact environment when the app crashes. 53 developers (39.3%) selected (A), since some concurrency bugs require specific thread scheduling and strict timing [81]. 44 developers (32.6%) reported missing "specific system configurations or settings" in the reproduction steps could also affect the reproducibility. For example, some bugs can only be manifested with mobile data instead of WiFi.

We further asked app developers an open question Q15 "*How do you improve the reproducibility of exception bugs during your development*?". 12 app developers answered this question. They added customized logging interfaces to gain important running information, or used some off-the-shelf crash reporting systems, *e.g.*, ACRA [82], Google Firebase Crashlytics [83], Splunk MINT [84], to collect raw analytics. Specifically, these crash reporting systems (integrated as app plugins) collect the contextual environment (*e.g.*, SDK, OS, app version, hardware model, memory usage), the exception traces, the steps leading to crash (usually in the form of screenshots) to facilitate crash analysis. However, these developers still felt quite challenging to faithfully reproducing exception bugs the users experience in vivo.

3.5.2 Perspective of Testing Tools

To investigate how testing tools perform in reproducing bugs, we chose two Android GUI testing tools, *i.e.*, Sapienz and Stoat. As stated in Section 3.4, these two tools are now the state-of-the-art in finding crash bugs. Specifically, to record & replay the tests, we used Android Monkey script [85] for Sapienz, and UIAutomator script [86] for Stoat. When an app crashes during the testing, we will record the exception trace, and the corresponding crash-triggering test (*i.e.*, the event sequences that led to the crash).

To mitigate test flakiness [87], [88], [89], we deployed the reproducing process on two physical machines, each of which ran 6 emulators with the exact same environment and configurations as the previous testing process in Section 3.4. In addition, we ran each test for five times, and recorded how many times the exception bugs could be triggered. The machine state was cleared between each test run. If the exactly same exception (with the same exception type and stack trace) was triggered among the 5 runs, we regarded the test as a valid one that can *faithfully* reproduce the crash. In total, we replayed the tests of 4,009 and 3,535 exception bugs (including all the three exception categories) found by Sapienz and Stoat, respectively. The whole reproducing process took two months. Note that we have not included Monkey in this investigation, since we find the tests of Monkey are very flaky[8].

8. Our preliminary investigation reveals Monkey's tests are much more flaky than those of Stoat and Sapienz. We find most of Monkey's tests have thousands of events, while those of Sapienz and Stoat have merely hundreds or tens of events, respectively.

TABLE 5: Statistics of reproducible exceptions across the three exception categories.

Tool	#Total	#Application	#Framework	#Library
Sapienz	279	82 (6.5%)	169 (7.2%)	28 (6.9%)
Stoat	269	189 (9.6%)	76 (5.3%)	4 (3.2%)

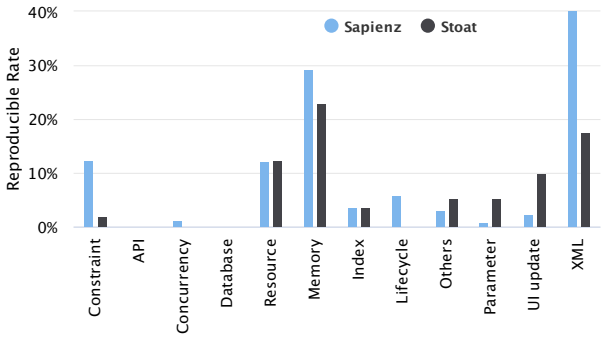

Fig. 15: Reproducibility rate of the 11 root causes of framework exceptions *w.r.t.* Sapienz and Stoat.

Finally, Sapienz and Stoat triggered 15.7% (629/4,009) and 28.2% (996/3,535) of all exception bugs, respectively, by replaying the recorded tests. However, among these triggered exceptions, only 279 exceptions of Sapienz (6.9%, including 82 application exceptions, 169 framework exceptions, and 28 library exceptions) and 269 exceptions of Stoat (7.6%, including 189 application exceptions, 76 framework exceptions, and 4 library exceptions), respectively, were faithfully reproduced. Obviously, the reproducibility rate of exception bugs were quite low. In the remaining cases that triggered exceptions, we find the tests either triggered (1) the exceptions with different types, or (2) the exceptions with the same types but different stack traces. We further inspected a number of those "unfaithfully" reproduced exceptions (*i.e.*, the cases in (2)), and found some of them actually triggered the same bugs but the stack traces were slightly different from the expected ones.

Table 5 shows the numbers of reproducible exceptions across the three exception categories, respectively. In the parentheses, the percentage numbers indicate the ratios of reproducible exceptions among all exceptions of that category. We can see the reproducibility rates of these three exception categories do not have much differences, although Stoat has lower rates on framework and library exceptions, compared to Sapienz. Fig. 15 shows the reproducibility rates of the 11 root causes of framework exceptions. We can see that both Stoat and Sapienz can reproduce more exceptions of Resource-Not-Found, Memory/Hardware, and XML Layout errors (over 10%), while neither of them has good performance at Concurrency, API Update and Compatibility, and Database Management errors.

Overall, the reproducibility of exception bugs is low for both Sapienz and Stoat. We further investigated the reasons behind, and observed the three main difficulties.

• ***Test dependency.*** Both Sapienz and Stoat only record the current test that triggers the exception. However, many exceptions can only be manifested under specific preconditions, which need to be created by some previous tests. As a result, only replaying the current test may fail the reproduction. Simply recording all the previous tests is ineffective, while selectively recording the necessary tests *w.r.t.* the exception is nontrivial.

• ***Timing of Events.*** The execution timing of events are crucial for manifesting some types of exceptions. For example, concurrency bugs require critical timing of events, so as to create specific thread scheduling [81]. In other scenarios, due to the latency of network or computation, some UI widgets may not be quickly ready for executing the next event — causing the ignorance of the next event. Such ignorance may have negative effect on the execution of the whole event sequence, leading to totally different execution paths and results. Thus, to improve reproducibility, the tests should contain timing control operations.

• ***Specific Running Environment or Configurations.*** Triggering some exceptions require specific running environment, *e.g.*, the existence of specific files on the SD card. For example, one of the reasons for OutOfMemoryError is that the app tries to load a large-size file from the SD card. Without this file, such exceptions could not be reproduced. Some exceptions can only be triggered under specific sytem configurations, *e.g.*, disabling network access or granting the permission of using camera.

Answer to RQ5: *Reproducing exceptions is difficult for developers, and also challenging for automated testing tools. Specific device models, specific execution environment, concurrency issues, specific system configurations are the four main difficulties rated by developers. The reproducibility rates of Sapienz and Stoat are quite low (only 6.9% and 7.6%, respectively). Test dependency, timing of events, and specific running environment are the three main observed challenges for testing tools to faithfully reproduce exceptions..*

3.6 RQ6: Fixing Patterns and Characteristics

This section uses the exception-fix repository constructed in RQ2 (194 instances) to investigate the common practices of developers to fix framework exceptions. We categorized their fixing strategies by (1) the types of code modifications (e.g., modify conditions, reorganize/move code, tweak implementations); (2) the issue comments and patch descriptions. We finally summarized five common fix patterns, which can resolve over 90% of the issues in the repository. We further presented Q16 to the developers, and asked them to choose which fix practice they have ever used to fix framework exceptions. Fig. 18 shows the responses. We detail these fix practices as follows, which are ordered by the popularity from the most to the least.

• ***Work in Right Callbacks***. Inappropriate handling lifecycle callbacks of app components (*e.g.*, Activity, Fragment, Service) can severely affect the robustness of apps. The common practice to fix such problems is to work in the right callback. For example, in Activity, putting BroadcastReceiver's register and unregister into onStart() and OnStop() *or* onResume() and OnPause() can avoid IllegalArgument; and committing a FragmentTransaction before the activity's state has been saved (*i.e.*, before the callback onSaveInstanceState()) can avoid state loss exception [90], [91].

• ***Refine Conditional Checks***. Missing checks on API parameters, activity states, index values, database versions, external resources can introduce unexpected exceptions. Developers usually fix them via adding appropriate conditional checks. For example, Fig. 17 (a) checks cursor index to fix CursorIndexOutOfBound, Fig. 17 (b) checks the state of the activity attached by a Fragment to fix IllegalState. Most

```
// MozStumbler revision 6adbfe5
public class ServiceBroadcastReceiver extends BroadcastReceiver{
  public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    ... // handle the intent
    if (mMainActivity != null) {
-    mMainActivity.updateUI();
+    mMainActivity.runOnUiThread(new Runnable() {
+       public void run() {
+         mMainActivity.updateUI();
+       }
+     });
   }}}
```

Fig. 16: Example fixes by moving code into correct thread

```
(a) qBittorrent-Controller revision 8de20af
  Cursor cursor = contentResolver.query(...);
- cursor.moveToFirst();
+ if( cursor != null && cursor.moveToFirst()) {
    int columnIndex = cursor.getColumnIndex(filePath);
     ... // get the result from the cursor
+ }

(b) WordPress revision df3392f
  public class AbstractFragment extends Fragment{
  protected void showError(int messageId) {
+   if(!isAdded()) { return; }
    FragmentTransaction ft = getFragmentManager()...
     ... //commit a transaction to show a dialog
 }}
```

Fig. 17: Example fixes by adding conditional checks

exceptions from *Parameter Error*, *Indexing Error*, *Resource Error*, *Lifecycle Error*, and *API Error* were fixed by this strategy.

• ***Move Code into Correct Thread***. Messing up UI and background threads may incur severe exceptions. The common practice to fix such problems is to move related code into correct threads. Fig. 16 fixes CalledFromWrongThread by moving the code of modifying UI widgets back to the UI thread (via Activity#runOnUiThread()) that creates them. Similar fixes include moving the showings of Toast or AlertDialog into the UI thread instead of the background thread since they can only be processed in the Looper of the UI thread [92], [93]. Additionally, moving extensive tasks (*e.g.*, network access, database query) into background threads can resolve the exceptions NetworkOnMainThread and "Application Not Responding" (ANR) [94].

• ***Change APIs or Design Patterns***. Developers may fix an exception by using other APIs to achieve similar functionalities. For example, they will replace depreciated APIs with newly imported ones. Sometimes, they directly change the design pattern to avoid exceptions, which cannot be easily fixed in the original design.

• ***Optimize data storage and manipulations***. To resolve other exceptions, developers have to carefully adjust implementation algorithms, *e.g.*, optimize data storage and manipulations. For example, to fix OutOfMemory caused by loading Bitmap, the common practice is to optimize memory usage by resizing the original bitmap [95]; to fix data race exceptions, the common practice is to adopt mutex locks (*e.g.*, add *synchronized* to allow the execution of only one active thread) or back up the shared data [96].

To further understand the characteristics of developer fixes, we grouped these issues by their root causes, and computed (1) the number of code lines[9] the developers changed to fix this issue (Fig. 19), and (2) the issue closing

9. To reduce "noises", we excluded comment lines (*e.g.*, "//..."), annotation lines (*e.g.*, "@Override"), unrelated code changes (*e.g.*, "import *.*", the code for new features).

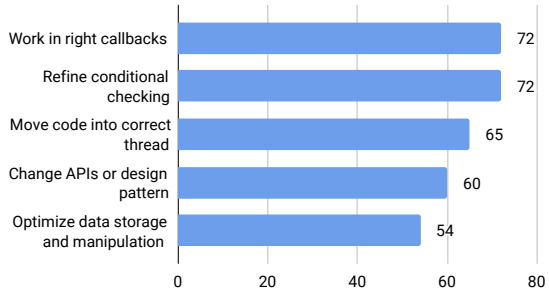

Fig. 18: Popularity of common fix practices by developers.

rate (column "Closing Rate" in Table 4). We can see that the fixes for *Parameter Error*, *Indexing Error*, *Resource Error*, and *Database Error* require fewer code changes (most patches are fewer than 20 lines). Because most of them can be fixed by refining conditional checks. We can also note *UI Update Error*, *API Updates and Compatibility Error*, *Concurrency Error*, *Memory/Hardware Error* and *XML Design Error* require larger code patches. Because fixing these issues usually require more manipulations on UI components, API compatibility, threads, memory and GUI design resources, respectively.

Further, by investigating the discussions and comments of developers when fixing, we find three important reasons that reveal the difficulties they face.

• ***Difficulty of Reproducing and Validation***. One prominent difficulty is how to reproduce exceptions and validate the correctness of fixes [76]. Most users, testing tools or platforms do not report complete reproducing steps/inputs and other necessary information (*e.g.*, exception trace, device model, code version) to developers. In most bug reports, we find only an exception trace is provided.

We surveyed the developers with Q17 "*In your experience, how much percentage of exceptions you are able to fix if you are only provided with an exception trace?*". We find only 12 developers (8.9%) reported they could fix over 70% exception bugs (only three developers say they could fix more than 90% exceptions). 27 developers (20.0%) selected 10∼30% exceptions, 54 developers (40.0%) selected 30∼50% exceptions, and 39 developers (28.9%) selected 50∼70% exceptions, respectively. We can see fixing exceptions could be rather difficult if only exception traces are available. In other cases, reproducing and validating non-deterministic exceptions (*e.g.*, concurrency errors) could be harder. After fixing these issue, developers choose to leave the app users to validate before closing the issue. As shown in Table 4, concurrency errors have low fixing rate.

• ***Inexperience with Android System***. A good understanding of Android system is essential to correctly fix exceptions. As the closing rates in Table 4 indicate, developers are more confused by Memory/Hardware Error, Lifecycle Error, Concurrency Error, and UI Update Error. We find some developers use simple *try-catch* or compromising ways (*e.g.*, use commitAllowingStateLoss to allow activity state loss) as workarounds. However, such fixes are often fragile.

• ***Fast Evolving APIs and Features***. Android is evolving fast. As reported, on average, 115 API updates occur each month [97]. Moreover, feature changes are continuously introduced. However, these updates or changes may make apps fragile when the platform they are deployed is different from the one they were built; and the developers are

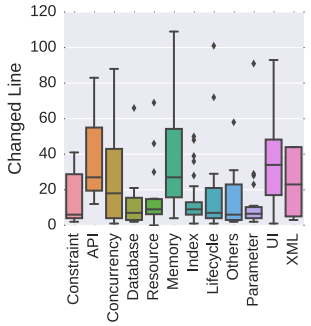

Fig. 19: Fixing in terms of number of changed lines.

confused when such issues appear. For example, Android 6.0 introduces runtime permission grant — If an app uses dangerous permissions, developers have to get permissions from users at runtime. However, we find several developers choose to delay the fixing since they have not fully understand this new feature.

Answer to RQ6: *Working in the right callbacks, using correct thread types, refining conditional checks, changing APIs or design patterns, optimizing data storage or manipulation are the five common fix practices. When developers fix framework exceptions, UI Update Error, API Updates and Compatibility Error, Concurrency Error, Memory/Hardware Error and XML Design Error require larger code patches. Meanwhile, reproducing exceptions and validating the fixes, understanding different mechanisms in Android system and adapting to fast-evolving Android APIs and features are the three main difficulties that developers face during fixing.*

4 THE BENCHMARK *DroidDefects*

Based on the data and analysis results in Section 3, this section aims to construct a benchmark of exception bugs for Android apps. This benchmark can facilitate follow-up research (*e.g.*, static fault analysis [98], fault localization [25], program repair [26]), and help measure effectiveness of proposed techniques in a controlled and systematic way.

However, constructing such a benchmark is non-trivial. First, for Android apps, most bug-triggering tests are reported in natural language, which describe the specific user actions to manifest the defects. These tests cannot be directly executed against the app to validate bugs [74]. Automatically translating these tests to executable ones are extremely difficult [78], [79]. Second, Android system and its apps are evolving fast, and use a diverse set of third-party libraries and different build systems (e.g., *Gradle*, *Ant*). These dependencies make it rather difficult to fully automate the build process, and usually involve considerable human efforts to resolve issues. Third, GUI tests can be notoriously flaky [87], [88], [89], which may not be able to deterministically manifest the defects. Due to these challenges, we cannot follow prior benchmarking methods [99], [100] to automate the construction. To bridge the gap, we made tremendous efforts to construct this benchmark.

4.1 Android App Defect Scenario

Our bug repository, *DroidDefects*, now only considers *reproducible*, *crash* defects that are the bugs of apps themselves. Other defects like Android system bugs [101], third-party library bugs [102], device fragmentation bugs [64], and non-crashing bugs (*e.g.*, performance and energy bugs [103], resource and memory leaks [57], [104], GUI failures [105], [106], security bugs [107], [108]) are not considered. To characterize an Android app bug in our context, we define the defect scenario as follows, which includes

- *A complete app project with one specific defect*, which incorporates the source code, the dependency libraries and the build scripts (e.g., *Gradle* or *Ant*). The project can be successfully compiled into an *apk* file for running on an emulator or a real device; and the defect can be deterministically reproduced by one or more tests.
- *An exception stack trace*, which is induced by the defect. It provides certain clues of the defect (e.g., the exception type, message, and the invoked methods). In particular, it tracks the sequence of called methods up to the point where the exception is thrown.
- *A bug-triggering test and its environment*, which can deterministically manifest the defect of the app, given the specific environment (*e.g.*, API version, system configuration). The test usually is composed of a sequence of user actions and/or system events. The test can be written in the form of natural language, JUnit-based test scripts (*e.g.*, Espresso [109], UIAutomator [86]) or low-level events (*e.g.*, Monkey scripts [85]).
- Optionally, *a developer-written repair or patch*, which fixes the faulty behavior *w.r.t.* the defect. It can be used for understanding the defect.

4.2 Artifacts of *DroidDefects*

DroidDefects contains three main artifacts: (1) dataset of reproducible defects, (2) dataset of ground-truth defects, and (3) utility scripts.

Dataset of reproducible defects. This dataset now contains 33 reproducible defects from 29 Android apps, and covers 26 distinct exception types. This dataset helps researchers to understand the characteristics of different app exceptions, and enables detailed analysis. Although this dataset is relatively small, but it covers different types of exception bugs from the 11 common root causes, and provides with detailed reproducibility and root cause information. All these information has never been considered in those previously constructed dataset [69], [27], [78], [26]. We will continuously evolve and enhance this dataset to include more exception instances, although our experience indicate this process requires tremendous manual efforts [79]. Section 4.3.1 gives the setup details. For each defect, it provides:

- **Source code of faulty app version**, the complete source code of the faulty app version with the build scripts and the compiled *apk* file.
- **Reproducible tests**, the test cases that can deterministically manifest the defect (written in natural language).
- **Exception trace**, the exception trace *w.r.t.* the defect.
- **Root cause analysis**, the explanation of the defect.

In the current version of dataset, we have not yet included non-deterministic defects (*e.g.*, data race bugs [66], [56]), since they require specific timing controls.

Dataset of ground-truth defects. This dataset provides 3,696 distinct real faults from 821 apps, which cover all the 11 root causes summarized in Section 3.2. For each fault, we provide the app project source code, the executable *apk* file and the exception trace. This dataset can be used to evaluate the

TABLE 6: Statistics of ground-truth defects *w.r.t.* 11 common root causes of framework exceptions.

Category (Name for short)	#Defects	#Apps
API Updates and Compatibility (**API**)	33	16
XML Layout Error (**XML**)	66	30
API Parameter Error (**Parameter**)	675	181
Framework Constraint Error (**Constraint**)	168	95
Index Error (**Index**)	551	183
Database Management Error (**Database**)	51	15
Resource-Not-Found Error (**Resource**)	1,238	286
UI Update Error (**UI**)	170	53
Concurrency Error (**Concurrency**)	241	71
Component Lifecycle Error (**Lifecycle**)	301	160
Memory/Hardware Error (**Memory**)	123	63
Others (Java-specific errors)	79	40
Total	3,696	821

effectiveness of the fault detection, localization or repairing techniques at the large-scale. Section 4.3.2 details the setup.

Utility scripts. The utility scripts contain the APIs to run existing tools, including dynamic testing tools (Monkey, Sapienz, and Stoat) and static analysis tools (Lint and FindBugs). This can ease the setup of evaluation. For example, researchers can evaluate his/her newly-proposed testing tool with the three state-of-the-art ones on our dataset via calling dedicated APIs.

4.3 Benchmark Setup Details

Apps. To construct *DroidDefects*, we chose to use open-source apps since the availability of source code enables detailed analysis. We chose app subjects from F-Droid. As discussed in Section 2.2.1, F-Droid apps are the representatives of real-world apps and most of them are maintained on GitHub.

4.3.1 Setup of reproducible defects

Selection Criteria. To construct a comprehensive dataset, we used *exception types* as the main guidance. Specifically, we purposely selected a number of typical Android app defects to cover each exception type from each root cause group (summarized in Section 3.2), respectively.

Source of defects. We mainly collected Android app defects from GitHub issue repositories, since these defects may be reported with the reproduction steps and other information. We also referred to the defects used by recent literature [78], [26]. We have not considered the defects from testing tools in Section 3.4, since as revealed in Section 3.5, the reproducibility rates of generated tests are very low.

Manual Validation. To collect valid defects from GitHub issue repositories, we reused the dataset of app exceptions collected in Section 2.2.2. We used the keywords "crash/stop", "reproduce", "replicate", "version" to further filter the exceptions, and only considered the issues submitted in recent years. We constrained our focus on these keywords since we hoped to focus on those fail-stop defects[10] with clear reproduction steps on the specific app versions, which are quite important for manual validation and reproduction. We selected recent issues by considering Android apps could have outdated dependencies. Finally, we got 448 issues. However, by randomly inspecting some filtered issues, we note there were still many invalid ones (*e.g.*, the reproduction steps are incomplete, the keyword "version" did not match with the "app version", *etc*).

10. Note that not all exceptions can trigger app crashes, *e.g., caught exception*s or system warning exceptions (*e.g.,* the WindowLeaked exception only gives resource-leak warning without failing apps).

Next, three authors of this paper spent one month to manually validate and reproduce these issues. Specifically, we worked in the following steps. First, we randomly sampled some candidate issues for each exception type. Second, to get the faulty code version V_{bug}, we either (1) checked out the code commit right preceding the fixed version V_{fix} if the bug fix is explicitly mentioned, or (2) checked out V_{bug} according to the specified app version or the issue submission time. Third, we built the app into an executable *apk* via build scripts or Android Studio. Last, we installed the app on an Android device to replay the described reproduction steps and observe whether the exact exception will be thrown. In our experience, various reasons may fail the above reproduction process. For example, the compilation may fail due to outdated dependencies; the exception cannot be manifested due to incomplete reproduction steps or environment issues. Therefore, if we could not successfully reproduce an exception within one hour, we resorted to the other candidates. Finally, we got 33 reproducible defects.

4.3.2 Setup of ground-truth defects

To construct a large dataset of ground-truth defects, we leveraged the exceptions revealed by dynamic testing tools in Section 3.4. Table 6 shows the statistics of this dataset *w.r.t.* the root causes of framework exceptions. In total, we collected 3,696 framework exceptions across 11 common root causes, which were discovered in 821 unique Android apps. To facilitate the use, we characterized the complexity of each faulty app by number of lines, number of methods, number of activities, and number of classes, and the diversity by the app category. Finally, we got 3,696 ground-truth defects.

5 Applications of Our Study

5.1 Improving Exception Detection

Dynamic Testing. Enhancing testing tools to detect specific errors is very important. For example, (1) *Generate meaningful as well as corner-case inputs to reveal parameter errors*. We find random strings with specific formats or characters are very likely to reveal unexpected crashes. For instance, Monkey detects more SQLiteExceptions than the other tools since it can generate strings with special characters like "'" and "%" by randomly hitting the keyboard. When these strings are used in SQL statements, they can fail SQL queries without escaping. (2) *Enforce environment interplay to reveal lifecycle, concurrency and UI update errors*. We find some special actions, *e.g.*, change device orientations, start an activity and quickly return back without waiting it to finish, put the app at background for a long time (by calling another app) and return back to it again, can affect an app's internal states and its component lifecycle. Therefore, these actions can be interleaved with normal UI actions to effectively check robustness. (3) *Consider different app and SDK versions to detect regression errors*. We find app updates may introduce unexpected errors. For example, as shown in Fig. 8, the changes of database scheme can crash the new version since the developers have not carefully managed database migration from the old version. (4) *More advanced testing criteria* [110], [111] are desired to derive effective tests.

Static Analysis. Incorporating new checking rules into static analysis tools to enhance their abilities is highly valuable. We find FindBugs and SonarQube have not included any

Android-specific checking rules, while PMD defines three rules [112], although these tools all support checking Android projects. Lint defines 281 Android rules [113] but only covers a small portion of framework-specific bugs [29]. However, there are plausible ways to improving these tools. For example, to warn the potential crash in Fig. 7, static analysis can check whether the task running in the thread uses Handler to dispatch messages, if it uses, Looper#prepare() must be called at the beginning of Thread#run(); to warn the potential crash in Fig. 5, static analysis can check whether there is an appropriate checking on activity state before showing a dialog from a background thread. In fact, some work [98] already implements the lifecycle checking in Lint.

Demonstration of Usefulness. We implemented Stoat+, an enhancement version of Stoat [12] with two new strategies. These two strategies include eight enhancement cases: (1) *five* specific input formats (*e.g.*, empty string, lengthy string, null) or characters (*e.g.*, "'", "%") to fill in EditTexts or Intent's fields; (2) *three* specific types of *environment-interplay* actions mentioned in Section 5.1. These two strategies were implemented in the MCMC sampling phase of Stoat, and randomly inject these specific events into normal GUI tests to improve fault detection ablitiy (see Section 4.4 in [12]). We applied Stoat+ on dozens of most popular apps (*e.g.*, Facebook, Gmail, Google+, WeChat) from Google Play, and each app was tested for ten hours on a Google Pixel 3 device. At last, we successfully detected 3 previously unknown bugs in Gmail (one parameter error) and Google+ (one UI update error and one lifecycle error). All of these bugs were detected in the latest versions at the time of our study, and have been reported to Google and got confirmed. The detailed issue reports were available at the Stoat's website [114]. However, these bugs were not found by Monkey and Sapienz, while other testing tools, *e.g.*, CrashScope [115] and AppDoctor [116], only consider 2 and 3 of these 8 enhancement cases, respectively.

5.2 Enabling Exception Localization

We find developers usually take days to fix a framework exception. Thus, automatically locating faulty code and proposing possible fixes are highly desirable. Our study can shed light on this goal.

Demonstration of Usefulness. We built a framework exception localization tool, ExLocator, based on Soot [117], which takes as input an APK file and an exception trace, and outputs a report that explains the root cause of this exception. It currently supports 5 exception types from UI Update, Lifecycle, Index, and Framework Constraint errors. In detail, it first extracts method call sequences and exception information from the exception trace, and classifies the exception into one of our summarized fault categories according to the root exception and signalers. As shown in Section 3.2, these specific exception types have obvious fault patterns (*e.g.*, incorrect handling background threads). Exlocator utilizes these patterns and data-/control-flow analysis to locate the root cause. More technical details can be found in our descendant tool APEchecker [81], which automatically localizes UI update errors. The report gives the lines or methods that causes the exception, the description of the root cause and possible fixing solutions, and closely related Stack Overflow posts. From our benchmark *DroidDefects*, we randomly selected 6 exception cases for each of five supported exception types. At last, we got 30 exception cases in total. ExLocator was successfully able to locate 28 exceptions out of 30 (93.3% precision) by comparing with the patches by developers. By incorporating additional context information from Android framework (*e.g.*, which framework classes use Handler), our tool successfully identified the root causes of the remaining two cases. However, all previous fault localization work [118], [119], [25], [120] can only handle generic exception types.

5.3 Enhancing Mutation Testing

Mutation testing is a widely-adopted technique to assess the fault-detection ability of a test suite, as well as to guide test case generation and prioritization [121]. One crucial step of applying mutation testing in a new application domain (*e.g.*, Android apps) is to design specific mutation operators, which can represent typical programming faults, in addition to those generic mutation operators. For example, a number of mutation testing tools for Java programs (*e.g.*, Pit [122] and Major [123]) are available, but they do not include any Android-specific mutation operators. As a result, they may generate trivial mutants that may directly crash themselves when startup or cannot be complied into executables.

We identified 75 different exception instances (with unique exception types and messages) from the data in Table 4. But we find existing mutation operators [124], [125], [126], [127], [128] designed for Android apps only cover a few of these instances. Specifically, only 4 mutation operators (*i.e.*, *Intent Playload Replacement*, *Actvity/Service Lifecycle Method Deletion*, Fail on Back) of Deng *et al.*'s 17 operators [125], [126] may help reveal some specific framework exceptions (*e.g.*, lifecycle-related issues). Their remaining operators focus on detecting UI, event handling and energy failures instead of fatal crashes. MDroid+ [127] proposes 38 operators, but can only cover 8 exception instances in our study. Based on the results of our study, researchers could add more mutation operators. For example, we can delete Activity state checking statements from those methods running in background threads to inject Lifecycle errors (see Fig. 5); we can also remove specific statements (*e.g.*, app state storage) from some Fragment's lifecycle callbacks (*e.g.*, onSaveInstanceState) to inject state loss errors [90], [91]; we can also change some data access from UI threads to background threads to inject UI update errors (see Fig. 6). We can also inject many Framework constraint errors (*e.g.*, see the example in Fig. 7). All these generated mutants can be successfully compiled and only detectable at runtime with specific GUI tests. Thus, more mutation operators can be introduced for framework exception types to improve mutation testing of Android apps.

6 Discussion

6.1 Lessons Learned

We have learned several lessons from this study. We summarize them to inspire practitioners and researchers, and motivate future work.

Automatically reproducing exceptions need more research efforts. Reproducing exceptions is very important for bug diagnosing and fixing. First, in practice, only (incomplete) reproduction steps (written in natural language) or ex-

ception traces are available to developers. Although some tools [75], [115], [78], [79] have been developed to improve or automate bug reproduction, their effectiveness and usability are still limited. CrashScope [115] improves the reproducibility by recording more contextual information of bug-triggering event sequences. However, it still cannot handle exception bugs caused by inter-app communications. Yukusu [78] translates a bug report written in natural language into executable test cases. However, according to our replication of their evaluation, we find Yukusu still focuses on creating test cases instead of reproducing the expected bugs. RecDroid [79] is a further step of Yukusu, which aims to automatically reproduce the expected crash bugs from a bug report. However, it cannot cover all types of exception bugs (*e.g.*, concurrency bugs) and its ability is limited by its predefined grammar patterns. Thus, how to effectively and faithfully reproducing the intended bug described in a bug report still requires more research efforts. Second, how to reproduce an exception with a short event trace is also important [129], [130]. Existing testing tools (*e.g.*, Monkey and Sapienz) usually generate quite long traces which are flaky and not suitable for reproducing. However, when applied for GUI programs, existing test reduction techniques, *e.g.*, delta debugging [129], [131], still have high performance overhead. Thus, how to efficiently reduce bug-triggering tests is still an open problem. Third, if only an exception trace is available, effective techniques for locating faulty code and then generating the bug-triggering tests at the UI level are quite useful for bug reproduction. However, existing fault localization techniques for Android apps [25] are far from mature, and limited to trivial types of exceptions. Little research efforts has been done to link the app logic code with UI widgets for interactive debugging of Android apps. This deserves more research efforts.

Effective bug detection tools are in great demand. Both dynamic and static bug detection techniques are needed to effectively reveal as many exception bugs as possible before app release. First, Android apps could be complicated and have different types of bugs, and different testing strategies could have very different performance in detecting exceptions. Thus, one plausible idea is to combine the strengthens of these strategies together, *e.g.*, combining random testing and systematic GUI exploration [70], or using static analysis to guide dynamic testing [81]. Second, static analysis tools could include more specific rules to check potential bugs and keep update with the evolution of Android system. The rules that are closely related to Android programming errors (*e.g.*, Component Lifecycle Error, Framework Constraint Error, UI Update Error) could have higher fault detection abilities. Third, bug detection tools should improve their usability. For example, dynamic testing tools should provide mechanisms to automatically bypass user logins or accept user-provided account information, otherwise, they are likely trapped at the login pages. Other tool features are also very useful, *e.g.*, leveraging user-provided oracles, generating more readable and less flaky tests, reducing number of false positives. These can improve bug detection and reproduction, and save debugging efforts.

Better documentation and technical tutorials are needed. A comprehensive and intuitive technical documentation is very important for developers to quickly understand Android system and avoid programming errors. However, during this study, we find this issue is still prominent. For example, we notice developers are more capable of fixing trivial errors (*e.g.*, Parameter Error, Index Error) according to their Java programming knowledge, but takes more time and needs more discussions when fixing such Android-specific issues as Component Lifecycle, Memory/Hardware, Concurrency, and UI Update errors. However, some sophisticated mechanism are not well documented in the official Android documentation. One typical example is about the state loss issue when handling Activity and Fragment lifecycle [90]. Junior developers have to refer to those technical posts from experienced developers.

Second, we find some developers cannot quickly get familiar with the newly-introduced features. We observe some developers chose to delay the upgrading of their apps to new Android platforms. For example, Android introduces runtime permission granting since API 23; and supports Kotlin since API 27. Better documentation and training courses should be continuously updated to help developers gain more understanding of new mechanisms, and let them know the feature evolution of Android system.

6.2 Threats to Validity

External Validity. First, our selected apps may not be the representatives of all possible real-world apps. To counter this, we collected all 2,486 apps from F-Droid at the time of our study, which is the largest database of open-source apps, and covers diverse app categories. We also collected a diverse set of 3,230 closed-source Google Play apps as subjects. Second, our mined exceptions may not include all possible exceptions. To counter this, we mined the issue repositories of 2,174 apps on GitHub and Google Code; and applied testing tools on 5,334 unique apps, which leads to total 30,009 exceptions. To our knowledge, this is the largest study for analyzing Android app exceptions.

Internal Validity. First, our exception analysis may not be absolutely complete and correct. For completeness, (i) we investigated 8,243 framework exceptions, and carefully inspected all common exception types. (ii) We surveyed previous work [132], [133], [15], [19], [134], [116], [56], [21], [31], [11], [98], [12], [135] that reported exceptions, and observed all their exception types and patterns were covered by our study. For correctness, we cross-validated our analysis on each exception type, and also referred to the patches from developers and Stack Overflow posts. More importantly, we surveyed 135 professional app developers to gain more understandings and insights to validate our analysis. Second, the classification of app exceptions (Section 3.1) and the taxonomy of root causes (Section 3.2) may be subjective. This may affect the validity of some analysis results. To counter this, we carefully analyzed these exceptions based on our understanding, and the knowledge from Android documentation and development tutorials.

Construct Validity. The online developer survey may have some limitations. The designed questions may not fully cover all aspects, and affect the validity of our conclusions drawn from this survey. But we tried our best to design appropriate questions, and refined these questions according to early feedback from three experienced Android developers and our own long-time research experience of

inspecting developers activities on GitHub. In the survey, we also provided some questions with open options to receive any comments from developers, which complemented our provided options. Our constructed benchmark may also subject to construct validity. To counter this, we manually verified all reproducible cases. For ground-truth cases, we also automatically checked the validity of exception traces.

7 Related Work

A number of fault studies exist in the literature for Android apps from different perspectives, *e.g.*, performance [103], energy [136], compatibility issues [64], [137], permission issues [138], memory leak [139], [140], GUI failures [105], [106], [141], resource usage [57], [104], API stability [97], security [142], [143], [144] *etc*. However, none of these work particularly focuses on app crashes and exceptions, which is the main goal we target at in this work.

Hu *et al.* [132] make one of the first attempts to analyze functional bugs for Android apps. They manually classify 8 bug types (*e.g.*, *Activity* errors, *Event* errors, *Type* errors) from 158 bug reports of 10 apps. Other efforts include [134], [31], which however have different goals compared to our study: Coelho *et al.* [31] analyze exception traces to investigate the bug hazards of exception-handling code (*e.g.*, cross-type exception wrapping), Zaeem *et al.* [134] study 106 bugs of 13 apps to generate testing oracles for a specific set of bug types. However, none of them give a large-scale and comprehensive analysis in this direction, and the validity of their conclusions is also unclear.

Linares-Vásquez *et al.* [127] recently also analyze a large number of android app bugs. But our study is significantly different from theirs. First, we focus on analyzing crash bugs caused by framework exceptions, while they focus on designing mutation operators to evaluate the effectiveness of test suites. Second, we give a much more comprehensive, deep analysis on the root causes, exception detection, reproduction and fixing.

Based on our dataset and analysis results, we constructed the benchmark *DroidDefects*. Although prior work also construct some benchmarks of Android app faults, our benchmark is more systematic in the number of faults, exception types and root causes. For example, *AndroTest* [69], [145] is a dataset of 68 apps collected from early research work [133], [13], [15], [18], to evaluate the fault detection abilities of Android app testing tools. But these subjects are randomly selected from F-Droid without any specific selection criteria. Many of these apps are quite out-of-date and error-prone. *DroidBugs* [27], [146], the only available dataset for automated program repair of Android apps, merely contains 13 bugs from 5 apps. This dataset is introductory, and has not provided any information about bug types.

Researchers have also constructed benchmarks for other bug types. MUBench [147] is a benchmark of 89 API misuses mined from 33 real-world projects, including Android. AppLeak [148] is a benchmark of 40 resource leak bugs in Android apps, which contains the faulty apps, bug-fixed versions (when available), and reproducible test cases. Mostafa *et al.* [149] study behavioral backward incompatibilities of Java software libraries, including Android. They archived a number of backward incompatibility faults. In contrast, our work focus on exception bugs, and covers diverse categories and root causes.

8 Conclusion

In this paper, we conducted the first large-scale, comprehensive study to understand framework exceptions of Android apps, which account for the majority of app exception bugs. Specifically, we investigated framework exceptions from several perspectives, including exception characteristics, root causes, testing practice of developers, abilities of existing bug detection tools, exception reproducibility and common fix practices. To validate and generalize our analysis results, we considered both open-source and commercial apps, and further conducted an online developer survey to gain more insights from the developers' knowledge and experiences. Through this study, we constructed *DroidDefects*, the first comprehensive and largest benchmark of exception bugs, to enable follow-up research; and built two prototype tools, Stoat+ and ExLocator, to demonstrate the usefulness of our findings. We pointed a number of research directions that deserve more research efforts.

Acknowledgments

We would like to thank the constructive and valuable comments from the TSE reviewers. We also appreciate the Android app developers who participate in our online survey, and share us many experience and feedback in this field. This work was partially supported by SNSF Spark Project CRSK-2_190302; partially supported by NSFC Project 61632005 and 61532019; partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001, and NRF Investigatorship NRFI06-2020-0022.

References

[1] "Number of Android applications," http://www.appbrain.com/stats/number-of-android-apps.
[2] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *ICSE*, 2019, pp. 596–607.
[3] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.
[4] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *ESEM*, 2016, pp. 29:1–29:10.
[5] "Robotium," http://www.robotium.org.
[6] "Appium," http://appium.io/.
[7] "Android Lint," https://developer.android.com/studio/write/lint.html.
[8] "FindBugs," http://findbugs.sourceforge.net/.
[9] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *ICST*, 2015, pp. 1–10.
[10] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *ICSME*, 2015, pp. 352–361.
[11] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *ISSTA*, 2016, pp. 94–105.
[12] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *ESEC/FSE*, 2017, pp. 245–256.

[13] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *FSE*, 2012, p. 59.

[14] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using java pathfinder," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[15] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *ESEC/FSE*, 2013, pp. 224–234.

[16] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *OOPSLA*, 2013, pp. 641–660.

[17] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *FASE*, 2013, pp. 250–265.

[18] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *OOPSLA*, 2013, pp. 623–640.

[19] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: segmented evolutionary testing of Android apps," in *FSE*, 2014, pp. 599–609.

[20] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *MobiSys*, 2014, pp. 204–217.

[21] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[22] T. Su, "FSMdroid: Guided GUI Testing of Android Apps," in *ICSE*, 2016, pp. 689–691.

[23] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lu, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *ICSME*, 2017, pp. 103–114.

[24] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond gui testing for android applications," in *ASE*, 2017, pp. 27–37.

[25] H. Mirzaei and A. Heydarnoori, "Exception fault localization in android applications," in *MOBILESoft*, 2015, pp. 156–157.

[26] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *ICSE*, 2018, pp. 187–198.

[27] L. Azevedo, A. Dantas, and C. G. Camilo-Junior, "Droidbugs: An android benchmark for automatic program repair," *CoRR*, vol. abs/1809.07353, 2019.

[28] "Monkey," http://developer.android.com/tools/help/monkey.html.

[29] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *ICSE*, 2018, pp. 408–419.

[30] "Android Developers Documentation," https://developer.android.com/reference/packages.html.

[31] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *MSR*, 2015, pp. 134–145.

[32] "F-Droid," https://f-droid.org/.

[33] "Google Play Store," https://play.google.com/store/apps.

[34] "EMMA," http://emma.sourceforge.net/.

[35] "JaCoCo," http://www.eclemma.org/jacoco/.

[36] "Amazon Mechanical Turk," https://www.mturk.com.

[37] "CodePath Android Cliffnotes," http://guides.codepath.com/android.

[38] "Advanced Android Development," https://developer.android.com/courses/advanced-training/overview.

[39] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *CCS*, 2017, pp. 2187–2200.

[40] G. Nunez, "Party pooper: Third-party libraries in android," 2011.

[41] I. G. W. Group, "Standard glossary of terms used in software testing," *International Software Testing Qualifications Board*, pp. 1–25, 2015.

[42] "Root cause," https://en.wikipedia.org/wiki/Root_cause.

[43] "Activity Lifecycle," https://developer.android.com/guide/components/activities/activity-lifecycle.html.

[44] "Bankdroid," https://github.com/liato/android-bankdroid.

[45] "Bankdroid revision 8b31cd3," https://github.com/liato/android-bankdroid/commit/8b31cd36fab5ff746ed5a2096369f9990de7b064.

[46] "Fragments," https://developer.android.com/guide/components/fragments.html.

[47] "c:geo," https://github.com/cgeo/cgeo.

[48] "c:geo revision d6b4e4d," https://github.com/cgeo/cgeo/commit/d6b4e4d958568ea04669f511a85f24ac08f524b6.

[49] "Handler," https://developer.android.com/reference/android/os/Handler.html.

[50] "Looper," https://developer.android.com/reference/android/os/Looper.html.

[51] "Local-GSM-Backend," https://github.com/n76/Local-GSM-Backend.

[52] "PhoneStateListener," http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.3.1_r1/android/telephony/PhoneStateListener.java#PhoneStateListener.0mHandler.

[53] "Local-GSM-Backend revision 07e4a759," https://github.com/n76/Local-GSM-Backend/commit/07e4a759392c6f2c0b28890f96a177cb211ffc2d.

[54] "NetworkOnMainThreadException," https://developer.android.com/reference/android/os/NetworkOnMainThreadException.html.

[55] "Requesting Permissions at Runtime," https://developer.android.com/training/permissions/requesting.html.

[56] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for android applications," in *OOPSLA*, 2015, pp. 332–348.

[57] Y. Liu, J. Wang, L. Wei, C. Xu, S. Cheung, T. Wu, J. Yan, and J. Zhang, "Droidleaks: a comprehensive database of resource leaks in android apps," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3435–3483, 2019.

[58] "Nextcloud Notes," https://github.com/stefan-niedermann/nextcloud-notes.

[59] "Nextcloud Notes Issue," https://github.com/stefan-niedermann/nextcloud-notes/issues/199.

[60] "Nextcloud Notes revision," https://github.com/stefan-niedermann/nextcloud-notes/pull/212/commits/aa1a97292b5f7511473282cc40f23e786f019d7f.

[61] "Atarashii," https://github.com/AnimeNeko/Atarashii.

[62] "Atarashii revision b311ec3," https://github.com/AnimeNeko/Atarashii/commit/b311ec327413aa4ef4aaabb8a8496c61d342cfe9.

[63] "JDK 7 Compatibility Issues," http://kb.yworks.com/article/550/.

[64] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *ASE*, 2016, pp. 226–237.

[65] L. Wei, Y. Liu, S.-C. Cheung, H. Huang, X. Lu, and X. Liu, "Understanding and detecting fragmentation-induced compatibility issues for android apps," *IEEE Transactions on Software Engineering*, 2018.

[66] C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn, "Race detection for event-driven mobile applications," in *PLDI'14*, 2014, pp. 326–336.

[67] M. L. Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *ICSME*, 2017, pp. 613–622.

[68] "MonkeyRunner," https://developer.android.com/studio/test/monkeyrunner/.

[69] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *ASE*, 2015, pp. 429–440.

[70] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *ASE*, 2018, pp. 738–748.

[71] A. Arcuri and L. C. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test., Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.

[72] "Mann-Whitney U test," https://en.wikipedia.org/wiki/Mann-Whitney_U_test.

[73] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in android testing: they matter," in *A-Mobile*, 2019, pp. 1–6.

[74] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *ESEC/FSE*, 2015, pp. 673–686.

[75] ——, "FUSION: a tool for facilitating and augmenting android bug reporting," in *ICSE*, 2016, pp. 609–612.

[76] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *ICST*, 2016, pp. 33–44.

[77] M. White, M. L. Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk, "Generating reproducible and replayable bug reports from android application crashes," in *ICPC*, 2015, pp. 48–59.

[78] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *ISSTA*, 2019, pp. 141–152.

[79] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *ICSE*, 2019, pp. 128–139.

[80] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *MOBILESoft*, 2016, pp. 88–99.

[81] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *ASE*, 2018, pp. 486–497.

[82] "ACRA: Application Crash Reports for Android," https://github.com/ACRA/acra.

[83] "Google Analytics for Firebase," https://firebase.google.com/products/analytics/.

[84] "Monitor the performance and usage of your Android, iOS apps with Splunk Enterprise," https://mint.splunk.com/.

[85] "MonkeyScript," https://android.googlesource.com/platform/development/+/android-4.2.2_r1/cmds/monkey/src/com/android/commands/monkey/MonkeySourceScript.java.

[86] "UIAutomator," https://developer.android.com/training/testing/ui-automator.

[87] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: models, tools, and controlling flakiness," in *ICSE*, 2013, pp. 1479–1480.

[88] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014, pp. 643–653.

[89] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *ICSME*, 2018, pp. 534–538.

[90] "Fragment Transactions and Activity State Loss," http://www.androiddesignpatterns.com/2013/08/fragment-transaction-commit-state-loss.html.

[91] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in android applications," in *OOPSLA*, 2016, pp. 864–880.

[92] "NextGIS Mobile revision 2ef12a7," https://github.com/nextgis/android_gisapp/commit/2ef12a75eda6ed1c39a51e2ba18039cc571e5b0e.

[93] "WordPress revision 663ce5c," https://github.com/wordpress-mobile/WordPress-Android/commit/663ce5c1bbd739f29f6c23d9ecacbd666e4f806f.

[94] "Keeping Your App Responsive," https://developer.android.com/training/articles/perf-anr.html.

[95] "Managing Bitmap Memory," https://developer.android.com/topic/performance/graphics/manage-memory.html.

[96] "MPDroid Issue," https://github.com/abarisain/dmix/issues/286.

[97] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *ICSM*, 2013, pp. 70–79.

[98] S. GRAZIUSSI, "Lifecycle and event-based testing for android applications," Master's thesis, School Of Industrial Engineering and Information, Politecnico, 9 2016.

[99] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[100] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015.

[101] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *ISSRE*, 2010, pp. 249–258.

[102] J. Kochhar, J. Keng, and T. Biying, "An empirical study on bug reports of android 3rd party libraries," *Singapore Management University*, 2013.

[103] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *ICSE*, 2014, pp. 1013–1024.

[104] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *FSE*, 2016, pp. 396–409.

[105] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *ISSTA*, 2015, pp. 83–93.

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

[107] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, "Are mobile banking apps secure? what can be improved?" in *FSE*, 2018, pp. 797–802.

[108] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *ICSE*, 2020, pp. 596–607.

[109] "Espresso," https://developer.android.com/training/testing/espresso/.

[110] N. P. B. Jr., "Data flow oriented UI testing: exploiting data flows and UI elements to test android applications," in *ISSTA*, 2017, pp. 432–435.

[111] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A survey on data-flow testing," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 5:1–5:35, Mar. 2017.

[112] "PMD rules," https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/android.html.

[113] "Android Lint Checks," http://tools.android.com/tips/lint-checks.

[114] "Stoat," https://github.com/tingsu/Stoat.

[115] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: a practical tool for automated testing of android applications," in *ICSE*, 2017, pp. 15–18.

[116] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with AppDoctor," in *EuroSys*, 2014, pp. 18:1–18:15.

[117] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[118] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *ISSTA*, 2009, pp. 153–164.

[119] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang, "A debugging approach for java runtime exceptions based on program slicing and stack traces," in *QSIC*, 2010, pp. 393–398.

[120] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *ISSTA*, 2014, pp. 204–214.

[121] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[122] "PIT," http://pitest.org/.

[123] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *ISSTA'14*, 2014, pp. 433–436.

[124] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *ICST*, 2015, pp. 1–10.

[125] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information & Software Technology*, vol. 81, pp. 154–168, 2017.

[126] L. Deng, J. Offutt, and D. Samudio, "Is mutation analysis effective at testing android apps?" in *QRS*, 2017, pp. 86–93.

[127] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *ESEC/FSE*, 2017, pp. 233–244.

[128] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. L. Vásquez, G. Bavota, C. Vendome, M. D. Penta, and D. Poshyvanyk, "Mdroid+: a mutation testing framework for android," in *ICSE*, 2019, pp. 33–36.

[129] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI event traces," in *FSE*, 2016, pp. 422–434.

[130] W. Choi, K. Sen, G. C. Necula, and W. Wang, "Detreduce: minimizing android GUI test suites for regression testing," in *ICSE*, 2019, pp. 445–455.

[131] B. Jiang, Y. Wu, T. Li, and W. K. Chan, "Simplydroid: efficient event sequence simplification for android application," in *ASE*, 2017, pp. 297–307.

[132] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *AST*, 2011, pp. 77–83.

[133] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *ASE*, 2012, pp. 258–261.

[134] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *ICST*, 2014, pp. 183–192.

[135] L. Fan, S. Chen, L. Xu, Z. Yang, and H. Zhu, "Model-based continuous verification," in *APSEC*, 2016, pp. 81–88.
[136] A. Banerjee, H.-F. Guo, and A. Roychoudhury, "Debugging energy-efficiency related field failures in mobile apps," in *MOBILESoft*, 2016, pp. 127–138.
[137] H. Huang, L. Wei, Y. Liu, and S. Cheung, "Understanding and detecting callback compatibility issues for android applications," in *ASE*, 2019, pp. 532–542.
[138] A. Sadeghi, R. Jabbarvand, and S. Malek, "Patdroid: Permission-aware gui testing of android," in *ESEC/FSE*, 2017, pp. 220–232.
[139] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *HASE*, 2014, pp. 176–183.
[140] G. Santhanakrishnan, C. Cargile, and A. Olmsted, "Memory leak detection in android applications based on code patterns," in *i-Society*, 2016, pp. 133–134.
[141] J. Hu, L. Wei, Y. Liu, S. Cheung, and H. Huang, "A tale of two cities: how webview induces bugs to android applications," in *ASE*, 2019, pp. 702–713.
[142] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *SEC*, 2011, pp. 21–21.
[143] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: Automated generation of android phishing apps," *IEEE Transactions on Dependable and Secure Computing*, 2019.
[144] C. Tang, S. Chen, L. Fan, L. Xu, Y. Liu, Z. Tang, and L. Dou, "A large-scale empirical study on industrial fake apps," in *ICSE-SEIP*, 2019, pp. 183–192.
[145] "AndroTest," http://bear.cc.gatech.edu/~shauvik/androtest/.
[146] "DroidBugs," https://github.com/I4Soft/DroidBugs.
[147] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *MSR*, 2016, pp. 464–467.
[148] O. Riganelli, D. Micucci, and L. Mariani, "From source code to test cases: A comprehensive benchmark for resource leak detection in android apps," *Software: Practice and Experience*, vol. 49, no. 3, pp. 540–548, 2019.
[149] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of java software libraries," in *ISSTA*, 2017, pp. 215–225.

Ting Su is a postdoc scholar in Department of Computer Science, ETH Zurich, Switzerland, and will join East China Normal University as a Professor in Fall 2020. Previously, he was a visiting scholar of UC Davis, USA from 2014 to 2015. He received his B.S. in software engineering and Ph.D. in computer science from School of Software Engineering, East China Normal University, Shanghai, China, in 2011 and 2016, respectively. His research focuses on software testing and validation, and was recognized with three ACM SIGSOFT Distinguished Paper Awards (ICSE 2018, ASE 2018, ASE 2019). He has published broadly in top-tier programming language and software engineering venues, including PLDI, ICSE, FSE, ASE, and CSUR. More information is available at http://tingsu.github.io/.

Lingling Fan is a Research Fellow in School of Computer Science and Engineering, Nanyang Technological University, Singapore, and will join Nankai University in Fall 2020. She received her Ph.D and B.Eng. degrees in computer science from East China Normal University, Shanghai, China in June 2019 and June 2014, respectively. She had been a Research Assistant in Cyber Security Lab of NTU (2017-2019). Her research focuses on program analysis and testing, software security analysis and big data driven analysis, and has published in top-tier venues of software engineering and security (including ICSE, ASE, ESEC/FSE, S&P, TDSC, etc.) She got an ACM SIGSOFT Distinguished Paper Award at ICSE 2018. More information is available on https://lingling-fan.github.io/..

Sen Chen received his Ph.D. degree in computer science from School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in June 2019. Currently, he is a Research Assistant Professor in School of Computer Science and Engineering, Nanyang Technological University, Singapore, and will join College of Intelligence and Computing, Tianjin University as a tenured Associate Professor. Previously, he was a Research Assistant of NTU from 2016 to 2019 and a Research Fellow from 2019-2020. His research focuses on security and software engineering. He has published broadly in top-tier security and software engineering venues, including ICSE, ESEC/FSE, ASE, TSE, S&P, TDSC, etc. More information is available on https://sen-chen.github.io/.

Liu Yang received his Bachelor and PhD degrees in computer science from National University of Singapore (NUS) in 2005 and 2010, and continued with his postdoctoral research in NUS. He is now an Associate Professor in Nanyang Technological University. His current research focuses on software engineering, formal methods and security, and particularly specializes in software verification using model checking techniques, which led to the development of a state-of-the-art model checker, Process Analysis Toolkit. More information is available at http://www.ntu.edu.sg/home/yangliu/.

Lihua Xu is Associate Professor of Practice in Computer Science at NYU Shanghai. She received both her Masters and PhD degree in computer science from the University of California, Irvine. Her research interests are in software engineering and mobile security, with a focus on improving software quality via program analysis. She has published in top-tier venues such as ICSE, FSE, ASE, CCS, and MobiCom. Her recent work in software analysis received the 2018 ACM SIGSOFT Distinguished Paper Award. She is a recipient of the "Best New Investigator" award at the 2006 Grace Hopper Women in Computing conference.

Geguang Pu is a Professor in School of Computer Science and Software Engineering, East China Normal University. His research interests include program testing, analysis and verification. He served as PC members for international conferences such as SEFM, ATVA, TASE etc. He was a co-chair of ATVA 2015. He has published over 70 publications on the topics of software engineering and system verification (including ICSE, FSE, IJCAI, FM, ECAI, CONCUR etc). He completed his Ph.D. in Mathematics at Peking University in 2005, and received a B.S. in Mathematics from Wuhan University in 2000.

Zhendong Su is a Professor in Computer Science at ETH Zurich, where he specializes in programming languages, software engineering, and computer security. He received both his M.S. and Ph.D. degrees in Computer Science from the University of California at Berkeley, and both his B.S. degree in Computer Science and B.A. degree in Mathematics from the University of Texas at Austin. More information is available at https://people.inf.ethz.ch/suz/.