# Large-Scale Analysis of Framework-Specific Exceptions in Android Apps

Lingling Fan[1,2], Ting Su[2*], Sen Chen[1,2], Guozhu Meng[3,2]
Yang Liu[2], Lihua Xu[1*], Geguang Pu[1*], Zhendong Su[4]
[1]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[4]Department of Computer Science, University of California, Davis, USA
{ecnujanefan,tsuletgo,ecnuchensen}@gmail.com,{gzmeng,yangliu}@ntu.edu.sg
lhxu@cs.ecnu.edu.cn,ggpu@sei.ecnu.edu.cn,su@cs.ucdavis.edu

## ABSTRACT

Mobile apps have become ubiquitous. For app developers, it is a key priority to ensure their apps' correctness and reliability. 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 guide developers, or help improve testing and analysis tools. However, such studies do not exist — this paper fills this gap. Over a four-month long effort, we have collected 16,245 unique exception traces from 2,486 open-source Android apps, and observed that *framework-specific exceptions* account for the majority of these crashes. We then extensively investigated the 8,243 framework-specific exceptions (which took six person-months): (1) identifying their characteristics (*e.g.*, manifestation locations, common fault categories), (2) evaluating their manifestation via state-of-the-art bug detection techniques, and (3) reviewing their fixes. Besides the insights they provide, these findings motivate and enable follow-up research on mobile apps, such as bug detection, fault localization and patch generation. In addition, to demonstrate the utility of our findings, we have optimized Stoat, a dynamic testing tool, and implemented ExLocator, an exception localization tool, for Android apps. Stoat is able to quickly uncover three previously-unknown, confirmed/fixed crashes in Gmail and Google+; ExLocator is capable of precisely locating the root causes of identified exceptions in real-world apps. Our substantial dataset is made publicly available to share with and benefit the community.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

## KEYWORDS

Empirical study, mobile app bugs, testing, static analysis

**ACM Reference Format:**
Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, Zhendong Su. 2018. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180222

## 1 INTRODUCTION

Mobile applications have gained great popularity in recent years. For example, Google Play, the most popular Android market, has over three million apps, and more than 50,000 apps are continuously published on it each month [6]. To ensure competitive edge, app developers and companies strive to deliver high-quality apps. One of their primary concerns is to prevent fail-stop errors, such as app crashes from occuring in release versions.

Despite the availability of off-the-shelf testing platforms (*e.g.*, Roboelectric [76], JUnit [50], Appium [7]), and static checking tools (*e.g.*, Lint [31], FindBugs [23], SonarQube [82]) [51, 53], many released apps still suffer from crashes — two recent efforts [63, 85] discovered hundreds of previously unknown crashes in popular and well-tested commercial apps. Moreover, researchers have contributed a line of work [3, 5, 10, 17, 43, 44, 60, 61, 63, 83–85, 88, 91] to detect app crashes, but none of them have investigated the root causes. It leaves developers unaware of how to avoid and fix these bugs, and hinders the improvement of bug detection, fault localization [48, 66, 81, 90], and fixing [25] techniques. As observed by our investigation on 272,629 issues from 2,174 Android apps hosted on Github and Google Code, developers are unable to resolve nearly 40% reported crashes,[1] which greatly compromises app quality.

This situation underlines the importance of characterizing a large number of diverse real-world app crashes and investigating how to effectively detect and fix them. However, such a study is difficult and yet to be carried out, which has motivated this work.

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 invoked methods). Each exception can be classified into one of three categories — *application exception*, *framework exception*, and *library exception* — based on which architecture layer threw the exception. In particular, our

[1]Filtered by the keywords "crash" or "exception" in their issue descriptions.

*Ting Su, Lihua Xu and Geguang Pu are the corresponding authors. Lingling Fan and Ting Su contributed equally to this work.


*ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180222

study focuses on framework exceptions, which account for majority of app crashes (affecting over 75% of the projects), as revealed by our investigation in Section 4.1.

We face two key challenges in carrying out the study. The first is the *lack of comprehensive dataset*. To enable crash analysis, we need a comprehensive set of crashes from a large number of real-world apps. Ideally, for each crash, it includes exception trace, buggy source code, bug-triggering inputs, and the patches (if exists). However, to the best of our knowledge, no such dataset is publicly available. Despite open-source project hosting platforms maintain issue repositories, such as Github, our investigation reveals only a small set of crash issues (16%) are accompanied with exception traces. Among them, even if the issue is closed, it is not necessarily associated with the buggy code version. The second concerns *difficulties in crash analysis*. Analyzing crashes needs understanding of the application logic as well as the Android framework (or libraries). It is also necessary to cross-validate the root causes (*e.g.*, reproducing crashes, investigating knowledge from developers). However, no reliable tool exists that can facilitate our analysis.

To overcome these challenges and conduct this study, we made substantial efforts. We have collected 16,245 unique exception traces from 2,486 open-source Android apps by (1) mining their issue repositories hosted on Github and Google Code; and (2) applying state-of-the-art app testing tools (Monkey [34], Sapienz [63], and Stoat [85]) on their recent versions (corresponding to 4,560 executables) to complement the mined data. The whole data collection process took four months. We identified 8,243 unique framework exceptions, and spent nearly six person-months carefully investigating these crashes by examining the source code of apps and the Android framework, fixes from developers, bug reports from testing tools, and technical posts on Stack Overflow. We aim to answer the following research questions:

- ***RQ1***: *Compared with other exception categories, are framework exceptions recurring that affect most Android apps?*
- ***RQ2***: *What are the common faults made by developers that cause framework exceptions?*
- ***RQ3***: *What is the current status of bug detection techniques on detecting framework exceptions? Are they effective?*
- ***RQ4***: *How do developers fix framework exceptions? Are there any common practices? What are the difficulties for fixing?*

Through answering the above questions, we aim to characterize Android app crashes (caused by framework exceptions in particular) and provide useful findings to developers as well as researchers. For example, our investigation reveals framework exceptions are indeed recurring. Moreover, they require more fixing efforts (on average 4 days per exception) but have lower issue closing rate (only 53%) than application exceptions (67%). Through careful inspection, we distilled 11 common faults that developers are most likely to make, yet have not been well-investigated by previous work [18, 45, 92].

We further evaluate the detection abilities of current dynamic testing and static analysis techniques on framework exceptions. We are surprised to find static analysis tools are almost completely ineffective (only gives correct warnings on 4 out of total 77 exception instances), although there are some plausible ways to improve them. Dynamic testing tools, as expected, can reveal framework exceptions, but still far from effective on certain fault categories. Their testing strategies have a big impact on the detection ability. In addition, we find most exceptions can be fixed by four common practices with small patches (less than 20 code lines), but developers still face several challenges during fixing.

Our findings enables several follow-up research, *e.g.*, bug detection, fault localization, and patch generation for android apps. To demonstrate the usefulness of our findings, we have optimized Stoat, a dynamic testing tool, and implemented ExLocator, an exception localization tool, for android apps. The results are promising: Stoat quickly revealed 3 previously unknown bugs in Gmail and Google+; ExLocator is able to precisely localize the root causes of identified exceptions in real apps.

To summarize, this paper makes the following contributions:

- To our knowledge, we conducted the first large-scale study to characterize framework-specific exceptions in Android apps, and identified 11 common fault categories that developers are most likely to make. The results provide useful insights for developers and researchers.
- Our study evaluated the state-of-the-art exception detection techniques, and identified common fixing practices of framework exceptions. The findings shed light on proposing more effective bug detection and fixing techniques.
- Our findings enable several follow-up research with a large-scale and reusable dataset [21] that contains 16,245 unique exception traces from 2,486 open-source apps. Our prototype tools also demonstrate the usefulness of our findings.

## 2 PRELIMINARY

### 2.1 Existing Fault Study

Researchers have investigated Android and Symbian OSes' failures [62] and Windows Phone app crashes [75]. As for the bugs of Android apps, a number of studies exist in different aspects: performance [55], energy [11], fragmentation [89], memory leak [78, 79], GUI failures [1, 4], resource usage [54, 56], API stability [64], security [20, 65] and *etc*. However, none of them focus on *functional bugs*, which are also critical to user loyalty and app success. Our work focuses on this scope.

One of the first attempts at classifying functional bugs is from Hu *et al.* [45]. They classify 8 bug types from 10 apps. Other efforts [18, 92], however, have different goals: Coelho *et al.* [18] analyze exceptions to investigate the bug hazards of exception-handling code (*e.g.*, cross-type exception wrapping), Zaeem *et al.* [92] study bugs to generate testing oracles for a specific set of bug types. None of them give a comprehensive analysis, and the validity of their conclusions are unclear. Therefore, to our knowledge, we are the first to investigate Android app crashes, and give an in-depth analysis.

Our study focuses on the framework-specific exceptions (*framework exception* for short throughout the paper) that can crash apps, *i.e.*, those exceptions thrown from methods defined in the Android framework due to an app's violation of constraints enforced by the framework. Note we do not consider the framework exceptions caused by the bugs of the framework itself. We do not analyze application exceptions (leave this as our future work) and library exceptions (since different apps may use different third-party libraries whose analysis requires other information).

## 2.2 Exception Model in Android

Android apps are implemented in Java, and thus inherit Java's exception model. Java 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 references, divided-by-zero errors). (2) `Error`, which represents serious problems that a reasonable application should not try to catch (*e.g.*, `OutOfMemeoryError`). (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. For `RuntimeException` and `Error`, the programmers themselves have to handle them at runtime.

Figure 1 shows an example of `RuntimeException` trace. The bottom part represents the *root exception*, *i.e.*, `NumberFormatException`, which indicates the root cause of this exception. Java uses *exception wrapping*, *i.e.*, one exception is caught and wrapped in another (in this case, the `RuntimeException` of the top part), to propagate exceptions. Note 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* is the method (`invalidReal` in this case) that throws the exception, which is the first method call under the root exception declaration .

```
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)
    ….
```

**Figure 1: An example of `RuntimeException` trace**

# 3 OVERVIEW

Figure 2 shows the overview of our study. We select F-droid [41] apps as our subjects (Section 3.1), and use two methods, *i.e.*, mining issue repositories and applying testing tools, to collect exception traces (Section 3.2). We investigate exception traces and other resources (*e.g.*, Android documentation, app source code, Stack Overflow posts) to answer RQ1~RQ4 (Section 4). This study enables several follow-up research detailed in Section 5.

## 3.1 App Subjects

We choose F-droid, the largest repository of open-source Android apps, as the source of our study subjects, since it has three important characteristics: (1) F-droid contains a large set of apps. At the time of our study, it has more than 2,104 unique apps and 4,560 different app versions, and maintains their metadata (*e.g.*, source code links, release versions). (2) The apps have diverse categories (*e.g.*, Internet, Personal, Tools), covering different maturity levels of developers, which are the representatives of real-world apps. (3) All apps are open-source and hosted on Github, Google Code, SourceForge and *etc*, which makes it possible for us to access their source code and issue repositories for analysis.

## 3.2 Data Collection

Table 1 summarizes the statistics of the collected exception traces. We also collect other data for analysis from Stack Overflow and static analysis tools. The details are explained as follows.

**Table 1: Statistics of collected crashes**

| Approach | #Projects | #Crashes | #Unique Crashes |
|---|---|---|---|
| Hosting Platforms | 2174 | 7764 | 6588 |
| (Github/Google Code) | (2035/137) | (7660/104) | (6494/94) |
| Testing Tools | 2104 | 13271 | 9722 |
| (Monkey/Sapienz/Stoat) | (4560 versions) | (3758/4691/4822) | (3086/4009/3535) |
| Total | 2486 (1792 overlap) | 21035 | 16245 |

***Github and Google Code***. 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 Testing Tools.*** We set up as follows: (1) We chose three state-of-the-art Android app testing tools with different testing techniques: Google Monkey [34] (random testing), Sapienz (search-based testing), and Stoat (model-based testing). (2) We selected all the recent release versions (total 4,560 versions of 2,104 apps, each app has 1~3 recent release versions) maintained by F-droid as testing subjects. Each tool is configured with default setting and given 3 hours to thoroughly test each version on a single Android emulator. Each emulator is configured with KitKat 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. (3) We collect coverage data by Emma [77] or JaCoCo [42] to enable the testing of Sapienz and Stoat.

The evaluation took four months, and finally detected total 13,271 crashes (9,722 unique). In detail, Monkey detected 3,758 crashes (3,086 unique), Sapienz 4,691 crashes (4,009 unique), Stoat 4,822 crashes (3,535 unique). During testing, we record each exception trace with bug-triggering inputs, screenshots and detection time and *etc*, to help our analysis. Further. we find the issue repositories of Github/Google Code only record 545 unique crashes for these recent versions, which accounts for only 5.6% of those detected by testing tools. This indicates these detected exception traces can effectively complement the mined exceptions.

***Stack Overflow.*** According to exception traces mined from the two sources above, we also collect the most relevant posts on Stack Overflow by searching posts with key word "Android", exception types and detailed descriptions. We record information like create time, number of answers, question summary. We mined totally 15,678 posts of various exceptions.

***Static Analysis Tools.*** We also collect data from four state-of-the-art static analysis tools (Lint, PMD, FindBugs, SonarQube), which either serves as a plug-in of Android Studio or supports Android projects. We apply each tool on apps to collect potential bugs, warnings or code smells for in-depth analysis.

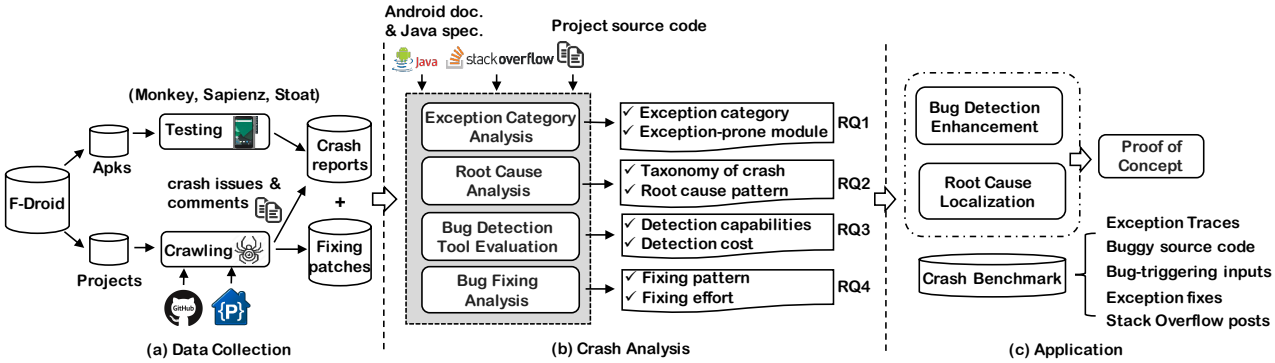


**Figure 2: Overview of our study and its applications**

# 4 EMPIRICAL STUDY

## 4.1 RQ1: Exception Categories

**Exception Categories**. To investigate app crashes, we group their exception traces into three different categories according to their exception signalers. In detail, we refer to Android-18 API documentation [26] and use the rules below (adopted by prior work [18]) to categorize exceptions: (1) *Application Exception*: the signaler is from the app itself (identified by the app's package name). (2) *Framework Exception*: the signaler is from the Android framework, *i.e.*, from these packages: "`android.*`", "`com.android.*`", "`java.*`", and "`javax.*`". (3) *Library Exception*: the signaler is from the core libraries reused by Android (*e.g.*, "`org.apache.*`", "`org.json.*`", "`org.w3c.*`" and *etc*) or third-party libraries used by the app.

**Table 2: Statistics of the exceptions from Github and Google Code grouped by their signalers (M: Median)**

| Exception Category | #Projects | Occurences | #Types | Issue Duration M (Q1/Q3) | Fixing Rate |
|---|---|---|---|---|---|
| App | 268 (45.8%) | 1552 (23.6%) | 88 (34%) | 2 (0/17) | 67% |
| Framework | 441 (75.3%) | 3350 (50.8%) | 127 (50%) | 4 (1/30) | 53% |
| Library | 253 (43.2%) | 1686 (25.6%) | 132 (52%) | 3 (1/16) | 57% |

Table 2 classifies the exceptions from Github and Google Code according to the above rules, and shows the number of their affected projects, occurrences, number of exception types, issue durations (the days during the issue was opened and closed), and the fixed issue rate (the percentage of closed issues). From the table, we observe two important facts: (1) *Framework exceptions are more pervasive and recurring*. It affects 75.3% projects, and occupies 50.8% exceptions. (2) *Framework exceptions require more fixing effort*. On average, it takes 2 more times effort (see column 5) to fix a framework exception than an application exception

These facts are reasonable. First, most apps heavily use APIs provided by *Android Development Framework* (ADF) to achieve their functionalities. ADF enforces various constraints to ensure the correctness of apps, however, if violated, apps may crash and throw exceptions. Second, fixing application exceptions is relatively easy since developers are familiar with the code logic. However, when fixing framework exceptions, they have to understand and locate the constraints they violate, which usually takes longer.

**Locations of Framework Exception Manifestation**. To further understand framework exceptions, we group them by the class names of their signalers. In this way, we get more than 110 groups. To distill our findings, we further group these classes into 17 modules. A *module* is used to achieve either one general purpose or stand-alone functionality from the perspective of developers. We group 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 kind of activity), *WindowsManager* (manage window display) and *etc*.

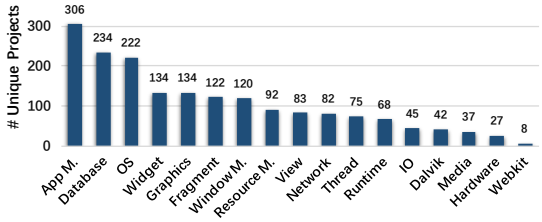


**Figure 3: Exception-proneness of Android modules for framework exceptions (*M.* refers to Management)**

Figure 3 shows the exception-proneness of these modules across all apps. We find *App Management*, *Database* and *OS* are the top 3 exception-prone modules. In *App Management*, the most common exceptions are `ActivityNotFound` (caused by no activity is found to handle the given intent) and `IllegalArgument` (caused by improper registering/unregistering `Broadcast Receiver` in the activity's callbacks) exceptions. Although `Activity`, `Broadcast Receiver` and `Service` are the basic building blocks of apps, surprisingly, developers make the most number of mistakes on them.

As for *Database*, the exceptions of *SQLite* (*e.g.*, `SQLiteException`, `SQLiteDatabaseLocked`, `CursorIndexOutOfBounds`) account for the majority, which reflects the various mistakes of using *SQLite*. In *OS*, `SecurityException`, `IllegalArgument`, `NullPointer` are the most common ones. As for the other modules, there are also interesting findings: (1) improper use of `ListView` with `Adapter` throws a large number of `IllegalState` exception (account for 47%) in *Widget*; (2) improper use of *Bitmap* causes `OutOfMemoryError` (48%)

in *Graphics*; (3) improper handling callbacks of *Fragment* brings `IllegalState` (85%) in *Fragment*; improper showing or dismissing dialogs triggers `BadTokens` (25%) in *WindowManager*.

**Answer to RQ1:** *Framework exceptions are pervasive, among which App Management, Database and OS are the three most exception-prone modules for developers.*

## 4.2 RQ2: Taxonomy of Framework Exceptions

This section investigates framework exceptions. We classify them into different categories by their root causes. *Root cause*, from the view of developers, is the initial cause of the exception.

***Exception Buckets.*** Following the common industrial practice, we group framework exceptions into different buckets. Each bucket contains the exceptions that share the similar root cause. To achieve this, we use the exception type, message and signaler to approximate the root cause. For example, the exception in Figure 1 is labeled as (`NumberFormatException`, "*invalid double*", `invalidReal`). Finally, we get 2,016 buckets, and find the exceptions from the top 200 buckets have occupied over 80% of all exceptions. The remaining 20% buckets have only 5 exceptions or fewer in each of them. Therefore, we focus on the exceptions of the top 200 buckets.

***Analysis Methods.*** We randomly select a number of exceptions from each bucket, and use 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 also manually checked the remaining issues to include valid ones that are missed by the keyword rules. We finally got 194 valid issues. We investigate each exception trace and its patch to understand the root causes. (2) *Exception Instances Repository*. From the 9,722 exceptions detected by testing tools, we filtered out the framework exceptions, and mapped each of them with its exception trace, source code version, 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 refer 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 3.2 when needing more information from developers and cross-validate our understanding.

***Taxonomy.*** We analyzed 86 exception types[2] (covering 84.6% of all framework exceptions), and finally distilled 11 common faults that developers are most likely to make. Table 3 lists them by the order of closing rate from highest to lowest. We explain them as follows.

• ***Component Lifecycle Error.*** Android apps are comprised of different components. Each component is required to follow a prescribed lifecycle paradigm, which defines how the component is created, used and destroyed [39]. For example, `Activity` provides a core set of six callbacks to allow developers to know 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 interrupt). *Bankdroid* [12] (Figure 4) is an app for providing service of Swedish banks. The app uses a background thread `DataRetrieverTask` to perform data retrieval, and pops up a dialog when the task is finished. However, if the user edits and updates a bank from `BankEditActivity` (which starts `DataRetrieverTask`), during which he presses the back button, the app will crash when the updates finish. The reason is that the developers fail to check `BankEditActivity`'s state (in this case, *destroyed*) after the task is finished. The bug triggers a `BadTokenException` and was fixed in revision 8b31cd3 [13]. Besides, `Fragment` [28], a reusable class implementing a portion of `Activity`, has much more complex lifecycle. It provides a core set of 12 callbacks to manage its state transition, which makes lifecycle management more challenging, *e.g.*, state loss of `Fragments`, attachment loss from its activity.

```
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();
+   }
  }}
```

**Figure 4: An Example of Lifecycle Error**

```
... // Once a sync is request, a new AsyncTask is fired-off
private class NoteSyncTask extends AsyncTask<Void,Void,...>{
    protected LoginStatus doInBackground(Void... voids) {
      ... // check local note status
      dbHelper.updateNote(note.getId(), remoteNote, note);
 }}
 ... // the method of dbHelper
 int updateNote(long id, CloudNote remoteNote, ...) {
  SQLiteDatabase db = this.getWritableDatabase();
  ... //backup local notes, take a while
  db.update(table_notes, values, ...);
- db.close();
}
```

**Figure 5: An Example of Concurrency Error**

```
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);
-   }
  }}
```

**Figure 6: An Example of UI Update Error**

• ***Concurrency Error.*** Android system provides such concurrent programming constructs as `AsyncTask` and `Thread` to execute intensive tasks. However, improper handling concurrent tasks may bring data race [14] or resource leak [54], and even cause app crashes. Nextcloud Notes [71] (Figure 5), a cloud-based notes-taking app that automatically synchronizes local and remote notes, when the app attempts to re-open an already-closed database [72]. The exception can be reproduced by executing these two steps repeatedly: (1) open any note from the list view; (2) close the note as quickly as possible by pressing back-button. The app creates a new

[2] After the investigation on a number of `NullPointerExceptions`, we find most of them are triggered by null object references. So we did not continue to analyze them.

```
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();
}
```

**Figure 7: An Example Violating Framework Constraints**

```
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);
     ...
 }}
```

**Figure 8: An Example of Database Management Error**

`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 by leaving the database unclosed (since `SQLiteDatabase` already implemented thread-safe database access mechanism) in revision aa1a972 [73].

• ***UI Update Error***. Each Android app owns a UI thread, which is in charge of dispatching events and rendering user interface. To ensure good responsiveness, apps should offload intensive tasks to background threads. However, many developers fail to keep in mind that *Android UI toolkit is not thread-safe and one should not manipulate UI from a background thread* [37]. *cgeo* [15] (Figure 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 the list is refreshed. The reason is that `cacheList` is maintained by the UI thread, which internally checks the equality of item counts between `ListView` and `cacheList`. But when a background thread touches `cacheList`, the checking will fail and an exception will be thrown. The developer realized this, and fixed it by moving the refreshing operations into `onPostExecute`, which instead runs in the UI thread (in revision d6b4e4d [16]).

• ***Framework Constraint Error***. Android framework enforces various constraints for app development. For example, `Handler` is part of Android framework for managing threads, which allows to send and process messages or runnable objects associated with a thread's message queue [29]. *Each* `Handler` *instance must be associated with a single thread and the message queue of this thread*[3]. Otherwise, a runtime exception will be thrown. *Local-GSM-Backend* [57] (Figure 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 [36], and thus requires setting up a message loop in `worker`. They later fixed it by calling `Looper#prepare()` (in revision 07e4a759 [58]). Other constraints include performance consideration (avoid performing network operations in the main UI thread [35], permission consideration (require run-time permission grant for dangerous permissions [38] since Android 6.0, otherwise `SecurityException`) and *etc.*

• ***Database Management Error***. Improper manipulating database columns/tables causes many exceptions. Besides this, improper data migration for version updates is another major reason. *Atarashii* [8] (Figure 8) is a popular app for managing the reading and watching of anime. When the user upgrades from v1.2 to v1.3, the app crashes once started. The reason is that the callback `onCreate()` is only called if no old version database file exists, so the new database table *friends* is not successfully created when upgrading. Instead, `onUpgrade()` is called, it crashes the app because the table *friends* does not exist (fixed in revision b311ec3 [9]).

• ***API Updates and Compatibility***. Android system is evolving fast. API updates and implementation (*e.g.*, SDKs, core libraries) changes can affect the robustness of apps. Device fragmentation [89] aggravates this problem. For example, `Service` should be started explicitly since Android 5.0; the change of the comparison contract of `Collections#sort()` [47] since JDK 7 crashes several apps since the developers are unaware of this.

• ***Memory/Hardware Error***. Android devices have limited resources (*e.g.*, memory). Improper using of resources may bring exceptions. For example, `OutOfMemoryError` occurs if loading too large Bitmaps; `RuntimeException` appears when `MediaRecorder#stop()` is called but no valid audio/video data is 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`.

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

• ***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 ignore checking their availability before use.

• ***Indexing Error***. Indexing error happens when developers access data, *e.g.*, *database*, *string*, and *array*, with a wrong index value. One

[3] A thread by default is not associated with a message queue; to create it, `Looper#prepare()` should be called in the thread [32].

**Table 3: Statistics of 11 common fault categories, and the evaluation results of static analysis tools on them, sorted by *closing rate* in descending order.**

| Category (Name for short) | Occurrence | #S.O. posts | #Instance | Static Tools | | | | Closing Rate |
|---|---|---|---|---|---|---|---|---|
| | | | | Lint | FindBugs | PMD | SonarQube | |
| API Updates and Compatibility (**API**) | 68 | 60 | 7 | - | - | - | - | 93.3% |
| XML Layout Error (**XML**) | 122 | 246 | 4 | 1 | - | - | - | 93.2% |
| API Parameter Error (**Parameter**) | 820 | 819 | 6 | - | - | - | - | 88.5% |
| Framework Constraint Error (**Constraint**) | 383 | 1726 | 12 | 3 | - | - | - | 87.7% |
| Others (Java-specific errors) | 249 | 4826 | 10 | - | - | - | - | 86.1% |
| Index Error (**Index**) | 950 | 218 | 4 | - | - | - | - | 84.1% |
| Database Management Error (**Database**) | 128 | 61 | 3 | - | - | - | - | 76.8% |
| Resource-Not-Found Error (**Resource**) | 1303 | 7178 | 5 | - | - | - | - | 75.3% |
| UI Update Error (**UI**) | 327 | 666 | 3 | - | - | - | - | 75.0% |
| Concurrency Error (**Concurrency**) | 372 | 263 | 7 | - | - | - | - | 73.5% |
| Component Lifecycle Error (**Lifecycle**) | 608 | 1065 | 11 | - | - | - | - | 58.8% |
| Memory/Hardware Error (**Memory**) | 414 | 792 | 3 | - | - | - | - | 51.6% |

typical example is the `CursorIndexOutOfBounds` exception caused by accessing database with incorrect cursor index.

In Table 3, column 2 and 3, respectively, counts the occurrences of each category and the number of Stack Overflow posts on discussing these faults; column 4 shows the number of distinct exception types of each category (total 75 instances). We find that (1) Besides the "trivial" errors such as Resource-Not-Found Error, Index Error and API Parameter Error, app developers are more likely to make Android specific errors, *e.g.*, Lifecycle Error, Memory/Hardware Error, Android Framework Constraint Error. (2) developers also discuss more on Android Framework Constraint Error, Lifecycle Error and API Parameter Error. Additionally, we find existing mutation operators [19, 52] designed for detecting app bugs can cover only a few of these 75 instances. Deng *et al.*'s 11 operators [19] can only detect 2 instances (the remaining ones detect UI and event handling failures instead of fatal crashes); MDroid+ [52] proposes 38 operators, but can only cover 8 instances.

**Answer to RQ2:** *We distilled 11 fault categories that explain why framework exceptions are recurring. Among them, developers make more mistakes on Lifecycle Error, Memory/Hardware Error and Android Framework Constraint Error. Existing mutation operators are inadequate for detecting these errors.*

## 4.3 RQ3: Auditing Bug Detection Tools

Dynamic testing and static analysis are the two main avenues to help detect software bugs. This section investigates the detection abilities of these two techniques on framework exceptions (categorized in Section 4.2). In particular, we select three state-of-the-art testing tools, *i.e.*, Monkey, Sapienz, and Stoat; and four static analysis tools widely used by android developers [53], *i.e.*, Lint, FindBugs, PMD, and SonarQube. Lint, developed by Google, detects code structural problems, and scans for android-specific bugs [27]. PMD uses defect patterns to detect bad coding practices. FindBugs, provided as a plugin in Android Studio, also enforces various checking rules, and adopts control- and data-flow analysis to scan potential bugs (*e.g.*, null-pointer dereferences). SonarQube is a continuous code quality platform that provides bug reports for suspicious code.

***Static Analysis Tools.*** We randomly select 75 distinct exception instances (corresponding to column 4 in Table 3) from Github that cover all manifestations of root faults, and checkout the corresponding buggy code version to investigate how many of them can be detected by static analysis tools. Our investigation finds static tools specialize in detecting bad practices, code smells, and potential bugs that may lead to severe errors, but with a mass of false alarms.

As shown in Table 3, FindBugs, PMD, and SonarQube fail to report any warnings on these bugs. Lint only identifies 4 out of 75 bugs, which include one XML error (the resource file "string.xml" contains an illegal character "$") and three framework constraint errors (duplicate resource ids within a layout file; Fragment cannot be instantiated; using the wrong AppCompat method). In addition, although these tools claim to support android projects, we have not found any android-specific rules in FindBugs and SonarQube, and only three android rules [74] in PMD. Lint defines 281 android rules [27] but detects only a few bugs. Therefore, the current static analysis tools focus more on basic Java defects, and much less effective in detecting framework exceptions of Android apps.

***Dynamic Testing Tool.*** We apply testing tools on each app (total 2,104) with the same configuration in Section 3.2. As we observed, they can detect many framework exceptions. To understand their abilities, we use two metrics[4]. (1) *detection time* (the time to detect an exception). Since one exception may be found multiple times, we use the time of its first occurrence. (2) *Occurrences* (how many times an exception is detected during a specified duration). Figure 9 and Figure 10, respectively, show the detection time and occurrences of exceptions by each tool grouped by the fault categories.

Figure 9 shows concurrency errors are hard to detect for all three tools (requiring longer time). But for other fault categories, the time varies on different tools. For example, Sapienz is better at database errors (since 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 are better at lifecycle errors (since both of them emit events very quickly without waiting the previous ones to take effect, *e.g.*, open and quickly close an activity without waiting the activity finishes its task). Figure 10 shows it is easy for three tools to detect API compatibility, Resource-Not-Found and XML errors since the occurrences of these errors are much more than those of the others. But for other categories, *e.g.*, Concurrency, Lifecyle, Memory, UI update errors, all of three tools are far from

[4]We have not presented 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.

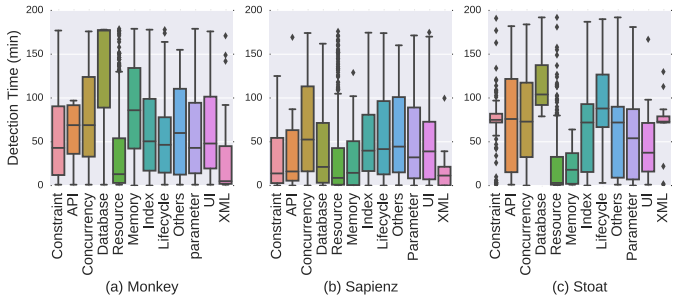


**Figure 9: Detection time of exceptions by each tool**

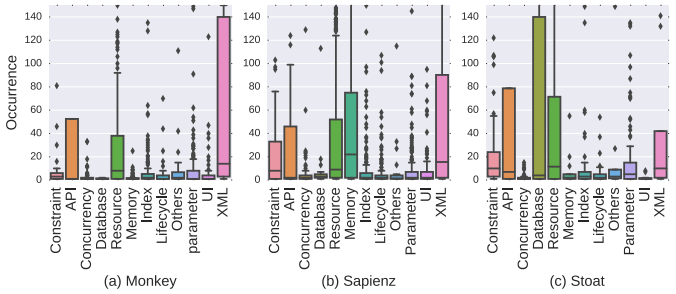


**Figure 10: Occurrences of exceptions by each tool**

effective regardless of their testing strategies. The main reason is that these errors contain non-determinism (interact 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 errors, some exceptions require special environment interplay. For example, `InstantiationException` of Fragment can only be triggered when a Fragment (without an empty constructor) 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 are hard to trigger since they usually need right timings of events.

**Answer to RQ3:** *Existing static analysis tools are ineffective in detecting framework exceptions. Dynamic testing tools are still far from effective in detecting database, framework constraint and concurrency errors.*

## 4.4 RQ4: Fixing Patterns and Efforts

This section uses the exception-fix repository constructed in RQ2 (194 instances) to investigate the common practices of developers to fix framework exceptions. We categorize 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 4 common fix patterns, which can resolve over 80% of the issues.

• ***Refine Conditional Checks***. Missing checks on API parameters, activity states, index values, database/SDK versions, external resources can introduce unexpected exceptions. Developers usually

```
(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
 }}

(c) MTA-Fare-Buster revision dba01df
   String input = amountOnCard.getText().toString();
+  if (input.equals("")) {
+    amountOnCard.setText(...); //set default value
+  }
   float amountOnCardValue=Float.valueOf(input.toString());
   ...
```

**Figure 11: Example fixes by adding conditional checks**

```
// 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();
+       }
+     });
  }}}
```

**Figure 12: Example fixes by moving code into correct thread**

fix them via adding appropriate conditional checks. For example, Figure 11 (a) checks cursor index to fix `CursorIndexOutOfBound`, Figure 11 (b) checks the state of the activity attached by a Fragment to fix `IllegalState`, and Figure 11 (c) checks the input of an `EditText` to fix `NumberFormat`. We find most of exceptions from *Parameter Error*, *Indexing Error*, *Resource Error*, *Lifecycle Error*, and *API Error* can be 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. Figure 12 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 [24, 67]. Additionally, moving extensive tasks (*e.g.*, network access, database query) into background thread can resolve such performance exceptions as `NetworkOnMainThread` and "Application Not Responding" (ANR) [30].

• ***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`

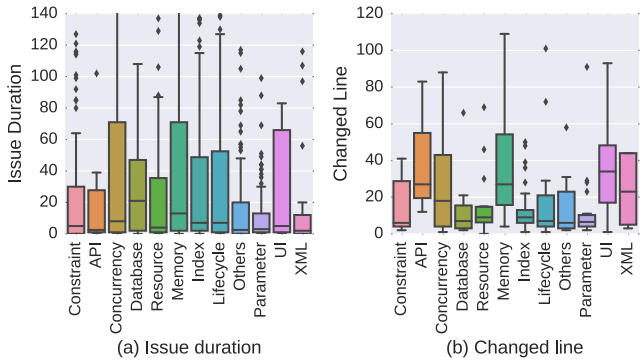


**Figure 13: Fixing Effort**

before the activity's state has been saved (*i.e.*, before the callback `onSaveInstanceState()`) can avoid state loss exception [59, 80].

• ***Adjust Implementation Choices***. To resolve other exceptions, developers have to adjust the implementation or do code refactoring. For example, to fix `OutOfMemory` caused by loading Bitmap, the common practice is to optimize memory usage by resizing the original bitmap [33]; 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 [70].

To further understand the characteristics of developer fixes, we group these issues by their root causes, and compute three metrics: (1) *Issue Duration*, which indicates how long the developers took to fix the issue (Figure 13(a)); (2) *Number of Changed Code Lines*, *i.e.*, the number of code lines[5] the developers changed to fix this issue (Figure 13(b)); and (3) *Issue Closing Rate*, *i.e.*, how many issues have been closed (the last column in Table 3). We can see that the fixes for *Parameter Error*, *Indexing Error*, *Resource Error*, and *Database Error* require fewer code changes (most patches are less than 20 lines). Because most of them can be fixed by refining conditional checks. We also note *UI Error*, *Concurrency Error*, and *Memory/Hardware Error* require larger code patches.

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 main difficulty is how to reproduce exceptions and validate the correctness of fixes [68]. Most users do not report complete reproducing steps/inputs and other necessary information (*e.g.*, exception trace, device model, code version) to developers. Even if the exception trace is provided, reproducing such exceptions as non-deterministic ones (*e.g.*, concurrency errors) is rather difficult. In such cases, after fixing the issue, they choose to leave it for the app users to validate before closing the issue. As shown in Figure 13 and Table 3, concurrency errors have longer issue durations and lower fixing rate.

• ***Inexperience with Android System***. A good understanding of Android system is essential to correctly fix exceptions. As the closing rates in Table 3 indicate, developers are more confused by Memory/Hardware Error, Lifecycle Error, Concurrency Error, and UI 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 [64]. 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 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 RQ4:** *Refining conditional checks, using correct thread types, working in the right callbacks, adjusting implementation choices are the 4 common fix practices. Memory/Hardware, Lifecycle, Concurrency, and UI update Error are more difficult to fix.*

## 4.5 Discussion

Through this study, we find: (1) Besides the trivial errors, developers are most likely to introduce Lifecycle, Memory/Hardware, Concurrency, and UI update errors, which requires more fixing efforts. (2) Bug detection tools need more enhancement. Static analysis tools could integrate new rules especially for UI update, Lifecycle, Framework Constraint errors. Testing tools could integrate specific testing strategies to detect these errors. (3) To counter framework exceptions, developers should gain more understanding on Android system; different supporting tools should be developed to reproduce exceptions for debugging, locate their root causes for fixing, and check API compatibility across different SDKs.

Linares-Vásquez *et al.* [52] also investigated android app bugs very recently, but our study significantly differs from theirs. We focuses on framework exceptions and give a comprehensive, deep analysis, including exception manifestations, root causes, abilities of existing bug analysis tools, and fixing practices. While they focus on designing mutation operators from existing bugs, and their 38 operators only cover 8 out of 75 instances distilled by our study. We believe our results can further improve existing mutation operators.

The validity of our study may be subject to some threats. (1) *Representativeness of the subjects.* To counter this, we collected all the subjects (total 2486 apps at the time of our study) from F-Droid, which the largest database of open-source apps, and covers diverse app categories. We believe these subjects are the representatives of real-world apps. (2) *Comprehensiveness of app exceptions.* To collect a comprehensive set of exception traces, we mine from Github and Google Code; and apply testing tools, which leads to total 16,245 exceptions. To our knowledge, this is the largest study for analyzing Android app exceptions. (3) *Completeness/Correctness of exception analysis.* For completeness, (i) we investigated 8,243 framework exceptions, and carefully inspected all common exception types. (ii) We surveyed previous work [2, 3, 14, 18, 22, 40, 45, 46, 60, 61, 63, 85, 92] that reported exceptions, and observed all 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 to validate our analysis. The whole dataset is also made publicly available.

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

# 5 APPLICATIONS OF OUR STUDY

This section discusses the follow-up research motivated by our findings, and also demonstrates usefulness by two prototype tools.

## 5.1 Benchmarking Android App Exceptions

Our study outputs a large and comprehensive dataset of Android app exceptions (especially for framework exceptions), which includes total 16,245 unique app exceptions and 8,243 unique framework exceptions. Each exception is accompanied with buggy code version, exception trace, error category, and possible fixes. We believe this dataset can (1) provide an effective and reliable basis for comparing dynamic/static analysis tools; (2) enable the research on investigating fault localization techniques and give a large set of exceptions as benchmarks; and (3) enable patch generation by comparing the exceptions and their fixes.

## 5.2 Improving Exception Detection

Dynamic testing and static analysis are the two avenues to detect faults. However, more improvements should be made on both sides.

***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 Figure 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 [49, 86] are desired.

***Static Analysis***. Incorporating new checking rules into static analysis tools to enhance their abilities is highly valuable. Through our study, we find it is feasible to check some framework exceptions, especially for framework constraint, lifecycle and UI update errors. For example, to warn the potential crash in Figure 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 Figure 4, static analysis can check whether there is appropriate checking on activity state before showing a dialog from a background thread. In fact, there is already some initial work [40] that implements lifecycle checking on Lint.

***Demonstration of Usefulness***. We enhanced Stoat [85] with two strategies: (1) randomly generate inputs with 5 specific formats (*e.g.*, empty string, lengthy string, null) or characters (*e.g.*, "", "%") to fill in `EditTexts` or `Intent`'s fields; (2) randomly inject 3 types of special actions mentioned above into normal UI actions. We applied Stoat on dozens of most popular apps (*e.g.*, Facebook, Gmail, Google+, WeChat) from Google Play, and 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. However, these bugs have not been found by Monkey and Sapienz, while other testing tools, *e.g.*, CrashScope [69] and AppDoctor [46] only consider 2 and 3 of these 8 enhancement cases, respectively.

## 5.3 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 have built a framework exception localization tool, ExLocator, based on Soot [87], 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 (As Figure 13 shows, these errors are more difficult to fix). 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, and then utilizes data-/control-flow analysis to locate the root cause. 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. We applied our tool on total 27 randomly selected cases from Github, and correctly locates 25 exceptions out of 27 (92% 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 [48, 66, 81, 90] can only handle general exception types.

# 6 CONCLUSION

This paper conducts a large-scale analysis of framework exceptions in Android apps. We constructed a comprehensive dataset that contains 16,245 unique exception traces. After investigating 8,243 framework exceptions, we identified their characteristics, evaluated their manifestation via popular bug detection tools, and reviewed their fixes. Our findings enables several follow-up research. We demonstrated the usefulness of our findings by two prototype tools.

# 7 ACKNOWLEDGEMENTS

We appreciate the anonymous reviewers for their valuable feedback. This work is partially supported by NSFC Grant 61502170, NTU Research Grant NGF-2017-03-033 and NRF Grant CRDCG2017-S04. Lingling Fan is partly supported by ECNU Project of Funding Overseas Short-term Studies, Ting Su partially supported by NSFC Grant 61572197 and 61632005, and Geguang Pu partially supported by MOST NKTSP Project 2015BAG19B02 and STCSM Project 16DZ1100600. Zhendong Su is partially supported by United States NSF Grants 1528133 and 1618158, and by a Google Faculty Research Award.

## REFERENCES


[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 83–93.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. 258–261.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59. https://doi.org/10.1109/MS.2014.55

[4] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test., Verif. Reliab.* 28, 1 (2018).

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. 59.

[6] AppBrain. 2017. Number of Android applications. (2017). Retrieved 2017-7 from http://www.appbrain.com/stats/number-of-android-apps

[7] Appium. 2017. Appium: Mobile App Automation Made Awesome. (2017). Retrieved 2017-7 from http://appium.io/

[8] Atarashii. 2017. Atarashii. (2017). Retrieved 2017-7 from https://github.com/AnimeNeko/Atarashii

[9] Atarashii. 2017. Atarashii revision b311ec3. (2017). Retrieved 2017-7 from https://github.com/AnimeNeko/Atarashii/commit/b311ec327413aa4ef4aaabb8a8496c61d342cfe9

[10] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 641–660.

[11] Abhijeet Banerjee, Hai-Feng Guo, and Abhik Roychoudhury. 2016. Debugging Energy-efficiency Related Field Failures in Mobile Apps. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft '16)*. ACM, New York, NY, USA, 127–138.

[12] Bankdroid. 2017. Bankdroid. (2017). Retrieved 2017-7 from https://github.com/liato/android-bankdroid

[13] Bankdroid. 2017. Bankdroid revision 8b31cd3. (2017). Retrieved 2017-7 from https://github.com/liato/android-bankdroid/commit/8b31cd36fab5ff746ed5a2096369f9990de7b064

[14] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 332–348.

[15] c:geo. 2017. c:geo. (2017). Retrieved 2017-7 from https://github.com/cgeo/cgeo

[16] c:geo. 2017. c:geo revision d6b4e4d. (2017). Retrieved 2017-7 from https://github.com/cgeo/cgeo/commit/d6b4e4d958568ea04669f511a85f24ac08f524b6

[17] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 623–640.

[18] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 134–145.

[19] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2017. Mutation operators for testing Android apps. *Information & Software Technology* 81 (2017), 154–168.

[20] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, USA, 21–21.

[21] Lingling Fan et al. 2017. Dataset of Android App Crashes. (2017). Retrieved 2017-7 from https://crashanalysis.github.io/

[22] Lingling Fan, Sen Chen, Lihua Xu, Zongyuan Yang, and Huibiao Zhu. 2016. Model-Based Continuous Verification. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE, 81–88.

[23] FindBugs. 2017. FindBugs. (2017). Retrieved 2017-7 from http://findbugs.sourceforge.net/

[24] WordPress for Android. 2017. revision 663ce5c. (2017). Retrieved 2017-7 from https://github.com/wordpress-mobile/WordPress-Android/commit/663ce5c1bbd739f29f6c23d9ecacbd666e4f806f

[25] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 307–318.

[26] Google. 2017. Android Developers Documentation. (2017). Retrieved 2017-7 from https://developer.android.com/reference/packages.html

[27] Google. 2017. Android Lint Checks. (2017). Retrieved 2017-8 from http://tools.android.com/tips/lint-checks

[28] Google. 2017. Fragments. (2017). Retrieved 2017-7 from https://developer.android.com/guide/components/fragments.html

[29] Google. 2017. Handler. (2017). Retrieved 2017-7 from https://developer.android.com/reference/android/os/Handler.html

[30] Google. 2017. Keeping Your App Responsive. (2017). Retrieved 2017-7 from https://developer.android.com/training/articles/perf-anr.html

[31] Google. 2017. Lint. (2017). Retrieved 2017-7 from https://developer.android.com/studio/write/lint.html

[32] Google. 2017. Looper. (2017). Retrieved 2017-7 from https://developer.android.com/reference/android/os/Looper.html

[33] Google. 2017. Managing Bitmap Memory. (2017). Retrieved 2017-7 from https://developer.android.com/topic/performance/graphics/manage-memory.html

[34] Google. 2017. Monkey. (2017). Retrieved 2017-7 from http://developer.android.com/tools/help/monkey.html

[35] Google. 2017. NetworkOnMainThreadException. (2017). Retrieved 2017-7 from https://developer.android.com/reference/android/os/NetworkOnMainThreadException.html

[36] Google. 2017. PhoneStateListener. (2017). Retrieved 2017-7 from http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.3.1_r1/android/telephony/PhoneStateListener.java#PhoneStateListener.0mHandler

[37] Google. 2017. Processes and Threads. (2017). Retrieved 2017-7 from https://developer.android.com/guide/components/processes-and-threads.html

[38] Google. 2017. Requesting Permissions at Run Time. (2017). Retrieved 2017-7 from https://developer.android.com/training/permissions/requesting.html

[39] Google. 2017. The Activity Lifecycle. (2017). Retrieved 2017-7 from https://developer.android.com/guide/components/activities/activity-lifecycle.html

[40] Simone GRAZIUSSI. 2016. *Lifecycle and Event-Based Testing for Android Applications*. Master's thesis. School Of Industrial Engineering and Information, Politecnico.

[41] F-droid Group. 2017. F-Droid. (2017). Retrieved 2017-2-18 from https://f-droid.org/

[42] JaCoCo Group. 2017. JaCoCo. (2017). Retrieved 2017-7 from http://www.eclemma.org/jacoco/

[43] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lu. 2017. AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. 103–114.

[44] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390

[45] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 77–83.

[46] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. 18:1–18:15.

[47] JDK. 2017. JDK 7 Compatibility Issues. (2017). Retrieved 2017-7 from http://kb.yworks.com/article/550/

[48] Shujuan Jiang, Hongchang Zhang, Qingtan Wang, and Yanmei Zhang. 2010. A Debugging Approach for Java Runtime Exceptions Based on Program Slicing and Stack Traces. In *Proceedings of the 2010 10th International Conference on Quality Software (QSIC '10)*. IEEE Computer Society, Washington, DC, USA, 393–398.

[49] Nataniel P. Borges Jr. 2017. Data flow oriented UI testing: exploiting data flows and UI elements to test Android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 432–435.

[50] Junit. 2017. Junit. (2017). Retrieved 2017-7 from http://junit.org/junit4/

[51] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the Test Automation Culture of App Developers. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 1–10.

[52] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 233–244.

[53] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How Developers Detect and Fix Performance Bottlenecks in Android Apps. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 352–361.

[54] Yepang Liu, Lili Wei, Chang Xu, and Shing-Chi Cheung. 2016. DroidLeaks: Benchmarking Resource Leak Bugs for Android Applications. *CoRR* abs/1611.08079 (2016).

[55] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 1013–1024. http://doi.acm.org/10.1145/2568225.2568229

[56] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and Detecting Wake Lock Misuses for Android Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 396–409.

[57] Local-GSM-Backend. 2017. Local-GSM-Backend. (2017). Retrieved 2017-7 from https://github.com/n76/Local-GSM-Backend

[58] Local-GSM-Backend. 2017. Local-GSM-Backend revision 07e4a759. (2017). Retrieved 2017-7 from https://github.com/n76/Local-GSM-Backend/commit/07e4a759392c6f2c0b28890f96a177cb211ffc2d

[59] Alex Lockwood. 2017. Fragment Transactions and Activity State Loss. (2017). Retrieved 2017-7 from http://www.androiddesignpatterns.com/2013/08/fragment-transaction-commit-state-loss.html

[60] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 224–234.

[61] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 599–609.

[62] Amiya Kumar Maji, Kangli Hao, Salmin Sultana, and Saurabh Bagchi. 2010. Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*. IEEE Computer Society, Washington, DC, USA, 249–258.

[63] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 94–105.

[64] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, Washington, DC, USA, 70–79.

[65] Guozhu Meng, Yinxing Xue, Jing Kai Siow, Ting Su, Annamalai Narayanan, and Yang Liu. 2017. AndroVault: Constructing Knowledge Graph from Millions of Android Apps for Automated Analysis. *arXiv preprint arXiv:1711.07451* (2017).

[66] Hamed Mirzaei and Abbas Heydarnoori. 2015. Exception Fault Localization in Android Applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '15)*. IEEE Press, Piscataway, NJ, USA, 156–157.

[67] NextGIS Mobile. 2017. revision 2ef12a7. (2017). Retrieved 2017-7 from https://github.com/nextgis/android_gisapp/commit/2ef12a75eda6ed1c39a51e2ba18039cc571e5b0e

[68] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. 33–44.

[69] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: a practical tool for automated testing of Android applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 15–18.

[70] MPDroid. 2017. MPDroid. (2017). Retrieved 2017-7 from https://github.com/abarisain/dmix/issues/286

[71] Nextcloud Notes. 2017. Nextcloud Notes. (2017). Retrieved 2017-7 from https://github.com/stefan-niedermann/nextcloud-notes

[72] Nextcloud Notes. 2017. Nextcloud Notes. (2017). Retrieved 2017-7 from https://github.com/stefan-niedermann/nextcloud-notes/issues/199

[73] Nextcloud Notes. 2017. Nextcloud Notes. (2017). Retrieved 2017-7 from https://github.com/stefan-niedermann/nextcloud-notes/pull/212/commits/aa1a97292b5f7511473282cc40f23e786f019d7f

[74] PMD. 2017. PMD Android rules. (2017). Retrieved 2017-8 from https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/android.html

[75] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 190–203.

[76] Robolectric. 2017. Robolectric: test-drive your Android code. (2017). Retrieved 2017-7 from http://robolectric.org/

[77] Vlad Roubtsov. 2017. EMMA. (2017). Retrieved 2017-2-18 from http://emma.sourceforge.net/

[78] Gayathri Santhanakrishnan, Chris Cargile, and Aspen Olmsted. 2016. Memory leak detection in android applications based on code patterns. In *Information Society (i-Society), 2016 International Conference on*. IEEE, 133–134.

[79] Hossain Shahriar, Sarah North, and Edward Mawangi. 2014. Testing of Memory Leak in Android Applications. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*. 176–183.

[80] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 864–880.

[81] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault Localization and Repair for Java Runtime Exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 153–164.

[82] Sonar. 2017. Sonar. (2017). Retrieved 2017-7 from https://www.sonarqube.org/

[83] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 27–37.

[84] Ting Su. 2016. FSMdroid: Guided GUI Testing of Android Apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 689–691.

[85] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.

[86] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1, Article 5 (March 2017), 35 pages.

[87] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.

[88] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5.

[89] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 226–237.

[90] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 204–214.

[91] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 250–265.

[92] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, Washington, DC, USA, 183–192.