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

Lingling Fan1,2, Ting Su2*, Sen Chen1,2, Guozhu Meng3,2
Yang Liu2, Lihua Xu1*, Geguang Pu1,4*, Zhendong Su3

1School of Computer Science and Software Engineering, East China Normal University, China
2School of Computer Science and Engineering, Nanyang Technological University, Singapore
3SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
4Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China

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

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 occurring 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

1 Filtered by the keywords “crash” or “exception” in their issue descriptions.
which architecture layer threw the exception. In particular, our 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., 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. 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>
<thead>
<tr>
<th>Approach</th>
<th>#Projects</th>
<th>#Crashes</th>
<th>#Unique Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hosting Platforms</td>
<td>2174</td>
<td>7764</td>
<td>6588</td>
</tr>
<tr>
<td>(Github/Google Code)</td>
<td>2015/317</td>
<td>7600/104</td>
<td>6494/94</td>
</tr>
<tr>
<td>Testing Tools</td>
<td>2104</td>
<td>13271</td>
<td>9722</td>
</tr>
<tr>
<td>(Monkey/Sapienz/Stoat)</td>
<td>(4360 versions)</td>
<td>(3735/4691/4822)</td>
<td>(3086/4009/5535)</td>
</tr>
</tbody>
</table>

| Total                        | 2386 (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.
## 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)**

<table>
<thead>
<tr>
<th>Exception Category</th>
<th>#Projects</th>
<th>Occurrences</th>
<th>#Types</th>
<th>Issue Duration</th>
<th>Fixing Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>268 (45.8%)</td>
<td>1552 (23.6%)</td>
<td>88 (34%)</td>
<td>2 (0/17)</td>
<td>67%</td>
</tr>
<tr>
<td>Framework</td>
<td>441 (75.3%)</td>
<td>3350 (50.8%)</td>
<td>127 (50%)</td>
<td>4 (1/30)</td>
<td>53%</td>
</tr>
<tr>
<td>Library</td>
<td>253 (43.2%)</td>
<td>1686 (25.6%)</td>
<td>132 (52%)</td>
<td>3 (1/16)</td>
<td>57%</td>
</tr>
</tbody>
</table>

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 *ActivityNotFoundException* (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, CursorIndexOutOfBoundsException*) account for the majority, which reflects the various mistakes of using *SQLite*. In *OS*, *SecurityException, IllegalArgumentException, NullPointerException* 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 *IllegalStateException* exception (account for 47%) in *Widget*; (2) improper use of *Bitmap* causes *OutOfMemoryError* (48%)
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 (NullPointerException, “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 types2 (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 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.

- **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

---

2After 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

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 develop-
ers 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] (Fig-
ure 6) is a popular full-featured client for geocaching. When re-
freshing cachelist (cachelist is associated with a ListView via an
ArrayAdapter), the developers query the database and sub-
stitute this list with new results (via clear() and addAll()) in
doInBackground. However, the app crashes when the list is re-
freshed. The reason is that cachelist is maintained by the UI
thread, which internally checks the equality of item counts be-
 tween 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 db1e46d [16]).

• Framework Constraint Error. Android framework enforces vari-
ous 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 associ-
ated with a single thread and the message queue of this thread8.
Oth-
erwise, 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

8A thread by default is not associated with a message queue; to create it,
Looper#prepare() should be called in the thread [32].

Figure 8: An Example of Database Management Error

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, improper data
migration for version updates is another major reason. Atsashii [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 b31e1ec [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 ex-
 plicitly since Android 5.0; the change of the comparison contract of
Collections#sort() [47] since JDK 7 changes 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 MediaPlayer#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 SharedPreferences 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 li-
 braries) 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
typical example is the CursorIndexOutOfBoundsException 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 discussing on 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.

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

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. (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, Lifecycle, Memory, UI update errors, all of three tools are far from

*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.
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 fix them via adding appropriate conditional checks. For example, Figure 11 (a) checks cursor index to fix CursorIndexOutOfBoundsException, Figure 11 (b) checks the state of the activity attached by a Fragment to fix IllegalStateException, and Figure 11 (c) checks the input of an EditText to fix NumberFormatException. 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 IllegalArgumentException; and committing a FragmentTransaction...
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 lines5 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 committing allowing StateLoss 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.

### 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-Vasquez 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 is the largest database of open-source apps, and covers various 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 patches from developers and Stack Overflow posts to validate our analysis. The whole dataset is also made publicly available.

---

5To 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 CRDCC2017-S04. Lingling Fan is partly supported by ECNU Project of Funded 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


