An Empirical Study of Functional Bugs in Android Apps

Yiheng Xiong
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
xyh@stu.ecnu.edu.cn

Mengqian Xu
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
xmq@stu.ecnu.edu.cn

Ting Su*
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
tsu@sei.ecnu.edu.cn

Jingling Sun
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
jingling.sun910@gmail.com

Jue Wang
State Key Lab for Novel Software Tech. and Dept. of Computer Sci. and Tech., Nanjing University
China
juewang591@gmail.com

He Wen
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
paigewen07@gmail.com

Geguang Pu*
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
ggpu@sei.ecnu.edu.cn

Jifeng He
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
jifeng@sei.ecnu.edu.cn

Zhendong Su
ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Android apps are ubiquitous and serve many aspects of our daily lives. Ensuring their functional correctness is crucial for their success. To date, we still lack a general and in-depth understanding of functional bugs, which hinders the development of practices and techniques to tackle functional bugs. To fill this gap, we conduct the *first* systematic study on 399 functional bugs from 8 popular open-source and representative Android apps to investigate the root causes, bug symptoms, test oracles, and the capabilities and limitations of existing testing techniques. This study took us substantial effort. It reveals several new interesting findings and implications which help shed light on future research on tackling functional bugs. Furthermore, findings from our study guided the design of a proof-of-concept differential testing tool, RegDroid, to automatically find functional bugs in Android apps. We applied RegDroid on 5 real-world popular apps, and successfully discovered 14 functional bugs, 10 of which were previously unknown and affected the latest released versions—all these 10 bugs have been confirmed and fixed by the app developers. Specifically, 10 out of these 14 found bugs cannot be found by existing testing techniques. We have made all the artifacts (including the dataset of 399 functional bugs and RegDroid) in our work publicly available at *https://github.com/Android-Functional-bugs-study/home*.

CCS CONCEPTS

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

KEYWORDS

Empirical study, Testing, Android, Non-crashing functional bugs

ACM Reference Format:
Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598138

1 INTRODUCTION

Android apps are GUI-based interactive applications. They are ubiquitous and serve many different aspects of our daily lives [53]. Recent reports show that app users highly value the user experience — only 16% of the users will try a function-failing app more than twice [10, 37]. Indeed, those function-failing apps could severely affect the users in real life [22, 41, 44, 51]. Thus, ensuring the functional correctness of an app is crucial for its success.

However, effectively finding non-crashing functional failures[1] (*functional bugs* for short) in Android apps remains a significant challenge [48, 64]. One *important* gap is that we still lack a general and in-depth understanding of functional bugs. For example, we are unclear how functional bugs are induced (*i.e.*, root causes), how these bugs affect the apps (*i.e.*, bug symptoms), and how these bugs

*Ting Su and Geguang Pu are the corresponding authors.

[1]In our context, the non-crashing functional failures denote those errors which fail the expected functionalities of an app but do not manifest themselves as app crashes.

Table 1: Summary of the prior relevant studies involving functional bugs ("✓","✗","?" denote that the study *does, does not* or *only partially does* the investigation, respectively). #Bugs denotes the number of functional bugs in the dataset.

Relevant Work	#Bugs	Dataset Available?	Root causes	Symptoms	Oracles	Assessing Tools	Lessons
Hu *et al.* [23]	147	No	✗	✗	✗	✗	✗
Zaeem *et al.* [69]	91	No	✗	✗	?	✗	✗
Johnson *et al.* [28]	130	Yes	✗	?	✗	✗	✗
Our work	399	Yes	✓	✓	✓	✓	✓

can be found (*i.e.*, test oracles). In contrast, crashing bugs in the apps have been extensively investigated [8, 14, 26, 54–56, 66].

To clearly show the gap, Table 1 summarizes the prior relevant bug studies involving functional bugs in Android apps. Hu *et al.* [23] are the first to classify the common bug types in Android apps, but they did not analyze the functional bugs in their dataset. Zaeem *et al.* [69] *only* analyzed some (limited) app agnostic oracles from a small set of bugs (15 bugs) in their dataset, and did not give detailed analysis on other oracle types. Johnson *et al.* [28] focused on investigating the reproducibility of bug reports, and only partially inspected the symptoms of the functional bugs in their dataset. None of these studies gives a systematic examination of functional bugs. As a result, it is difficult to fairly assess existing testing techniques and distill lessons on tackling functional bugs.

To fill the gap, this paper aims to complement the prior work by conducting a systematic empirical study to analyze functional bugs in Android apps. We studied a broad range of 399 functional bugs from 8 representative open-source Android apps. These apps are popular, actively-maintained and have different features. Specifically, we investigate the following research questions:

- **RQ1 (Root causes)**: What are the common root causes of these functional bugs? How these bugs are induced?
- **RQ2 (Bug symptoms)**: What are the symptoms of these functional bugs? How these bugs affect the apps?
- **RQ3 (Test oracles)**: What kinds of test oracles are needed to detect these functional bugs? How these bugs can be found?
- **RQ4 (Tool Evaluation)**: How is the status of existing testing techniques in finding these functional bugs? Are they effective?

Answering these questions is practically beneficial to both developers and researchers. **RQ1** characterizes the common faults which can help developers avoid functional bugs at the early stage of app development. **RQ2**~**RQ3** investigates the bug symptoms and oracles which can provide guidance for researchers to design effective bug finding techniques. **RQ4** examines the capabilities and limitations of existing testing techniques which can help better understand their effectiveness. Moreover, by analyzing the correlations between the results of **RQ1**~**RQ4**, we can find more insights. Specifically, *to validate the representativeness of our study* (**RQ1**~**RQ3**), we also analyzed a third-party bug dataset, AndroR2 [28, 68], which contains 130 functional bugs from 42 different apps. It shows that our analysis results based on our own bug dataset are *representative* and *consistent* with those based on AndroR2 (see Section 7).

Through our study, we have obtained several new interesting findings and implications which help shed light on tackling functional bugs (see Section 3, 4, 5, 6, 8). For example, developers should pay more attention to avoiding *missing cases* and *Android resource related error*, the two most common faults of functional bugs; researchers should choose appropriate types of oracles in finding different functional bugs, in addition to generating effective inputs.

Furthermore, findings from our study guided the design of a proof-of-concept differential testing tool, RegDroid, to automatically find functional bugs in Android apps. We applied RegDroid on 5 popular open-source apps, and successfully discovered 14 functional bugs, 10 of which are previously unknown bugs affecting the latest released versions—all these 10 bugs have been confirmed and fixed by the app developers. Specifically, among these 14 bugs, 10 bugs cannot be found by existing testing tools.

In summary, this paper has made the following contributions:

- To our knowledge, we conduct the *first* systematic study to investigate functional bugs in Android apps. We construct the largest dataset of 399 functional bugs, which serve as the basis of our study as well as future research in this direction.
- We study the functional bugs from different perspectives (root causes, bug symptoms, test oracles and the capabilities and limitations of existing testing techniques), and distill several findings.
- We enumerate the implications of our findings which help shed light on tackling functional bugs, and also demonstrate the usefulness of our implications via a proof-of-concept testing tool.

2 EMPIRICAL STUDY METHODOLOGY

This section presents our methodology for collecting and analyzing functional bugs in Android apps. Figure 1 shows the overview of our study including (a) bug collection and (b) bug analysis.

Table 2: Eight popular open-source and representative Android apps used in our study (K=1,000, M=1,000,000)

App Name	App Feature	First Release	#Installations	#Stars	#LOC	#Bugs Selected
Simplenote	Notebook	Nov. 2013	1M~5M	1.5K	37,649	35
AnkiDroid	Flashcard Learning	Jun. 2009	10M~50M	5.4K	218,558	82
Amaze	File Manager	Nov. 2014	1M~5M	4.0K	94,768	30
K-9 Mail	Email Client	Jan. 2014	5M~10M	7.1K	173,753	36
NewPipe	Video Player	Sep. 2015	5M~10M	20.7K	94,245	65
AntennaPod	Podcast Manager	Feb. 2014	500k~1M	4.4K	90,427	41
WordPress	Blog Manager	Dec. 2015	10M~50M	2.7K	457,891	67
Firefox	Web Browser	Jun. 2017	10M~50M	2.1K	68,249	43

2.1 Collection of Functional Bugs

Step 1: selecting app subjects. *This step aims to select representative app subjects for our study.* Specifically, we focus on open-source Android apps from GitHub because we can access their public issues. Specifically, we first obtained a set of candidate apps by using these two rules: (1) the app should be released on Google Play [20] to represent real-world apps, and (2) the app should contain enough closed issues for bug analysis (we require at least 200 closed issues). In this way, we got 182 candidate apps from GitHub. To select representative apps, we used the two metrics, *i.e.*, the app popularity *and* the app features, to narrow the scope. Specifically, (1) we first ranked the candidate apps according to their popularity from the most to least in terms of both the installations on Google Play and the stars on GitHub, and then (2) we manually examined them from the top to bottom to annotate the app features. Note that we decided the app feature based on the suggested app category on Google Play and the feature description on GitHub. Finally, we selected 8 most popular apps with different features. These apps

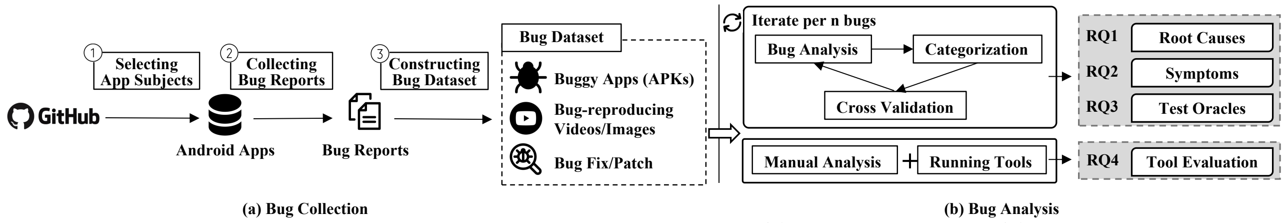

Figure 1: Overview of our study including two major steps: (a) bug collection and (b) bug analysis.

Table 3: The taxonomy of root causes.

Category	Subcategory	Description	Example	#Bugs	Ratio
General programming error	Missing features	The app functionality is supposed to be implemented but was not implemented.	Figure 2(a)	29	7.4%
	Missing cases	Some specific usage cases of the app functionality were not implemented.	Figure 2(b)	62	15.8%
	Wrong control flow	Some conditions in the control flow were incorrectly implemented.	Figure 2(c)	26	6.6%
	Lack of data synchronization	The content on the UI page was not properly synchronized with the changed app data (or the app data itself was not synchronized).	Figure 2(d)	34	8.7%
	Improper exception handling	The app failed to properly handle exceptional cases, thus breaking the normal execution of app functionality.	Figure 2(e)	10	2.5%
	Third-party library usage issue	The third-party libraries were incorrectly used.	-	10	2.5%
	Incorrect variable assignment	Some wrong values were assigned to specific program variables.	-	19	4.8%
	Other wrong functionality implementations	Any other bug which did not correctly implement the design requirement of the app functionality.	-	33	8.4%
	Subtotal	-	-	**223**	**56.7%**
Android related error	Android mechanism related error	Improper handling of Android development framework mechanisms (e.g., lifecycle management, event callbacks).	Figure 2(f) Figure 2(g)	45	11.5%
	Android resource related error	Errors in defining or manipulating Android app resources (e.g., layout files, icons, resource ids of UI elements).	Figure 2(h)	62	15.8%
	Android framework API misuse	The APIs of Android development framework were incorrectly used (e.g., violating API constraints or selecting inappropriate APIs).	Figure 2(i)	14	3.6%
	Android compatiblity issue	The app fails to be compatible with different SDK versions or device configurations.	-	27	6.8%
	Android framework bugs/limitations	The errors were caused by the bugs or limitations in the Android framework itself.	-	16	4.1%
	Subtotal	-	-	**164**	**41.7%**
Third-party library bugs/limitations	Third-party library bugs/limitations	The errors were caused by the bugs or limitations in the third-party libraries used by the apps.	-	6	1.5%

cover the most popular app categories on Google Play [4], *e.g.*, *Education*, *Business*, *Tools*, *Music&Audio*, *Entertainment*. Table 2 gives the details of these 8 apps, where *App Feature* denotes the major app feature, *First Release* gives the dates of their first releases on GitHub, *#Installations* and *#Stars* characterize the app popularity in terms of their installations on Google Play and stars on GitHub respectively, *#LOC* indicates the app complexity in terms of code lines (in Java, Kotlin and XML). We can see that these apps are diverse, popular, long-maintained and non-trivial.

Step 2: collecting bug reports. *This step aims to collect a broad range of representative functional bugs from the selected app subjects.* Specifically, we crawled *all* the issues which were reported within *the recent three years* (from August 2018 to July 2021) at the time of our study. Then, we filtered the issues which were explicitly labeled as *bugs* (*e.g.*, "*Bug*", "*Issue*") and were already *closed* by app developers. We focus on the closed issues because they have been resolved by developers and likely contain adequate information for bug analysis. In this way, we got 3,186 bug reports. Based on these 3,186 bug reports, we manually examined each of them to exclude invalid ones (*e.g.*, *duplicated*, *mislabeled*, *feature requests*), we got 2,482 valid bug reports. Among these 2,482 bug reports, we identified that 1,623 bug reports (65.4% ≈1,623/2,482) lead to functional bugs, 767 bug reports (30.9%) lead to crashing bugs and 91 bug reports (3.7%) lead to non-functional bugs (*e.g.*, performance or energy issues). We can see that functional bugs are indeed prevalent.

Step 3: constructing the bug dataset. *This step aims to rigorously construct a bug dataset of valid functional bugs.* Specifically, among the 1,623 functional bug reports, we only retained a bug report if it satisfies *all* the following three criteria. First, the corresponding functional bug is reproducible on the buggy app version at the time of our study, *or* the bug report contains clear bug-reproducing videos/images even if we failed to reproduce the bug by ourselves. Second, the corresponding bug is explicitly linked with the patch. Because the patch facilitates analyzing the bug's root cause. Third, the corresponding bug happens on common Android phone devices. We excluded those bugs which depend on specific devices (*e.g.*, tablets) or other platforms (*e.g.*, web clients) as they could be device or platform specific issues. In this way, we obtained 399 valid functional bugs, 265 of which are reproducible at the time of our study, and the remaining 134 bugs have clear bug-reproducing videos/images (although they are not reproducible at the time of our study). For each valid functional bug, we prepared (1) the buggy app (the APK file), (2) the bug-reproducing videos/images, and (3) the associated bug patch. Note that the other bugs were excluded because the reproducing information is unclear (*e.g.*, missing clear reproducing steps, faulty app versions, or bug-reproducing videos/images). This process took us considerable manual effort (nearly four person months) as reproducing bugs for mobile apps is notoriously difficult [17, 56, 68, 70]. In Table 2, *#Bugs Selected* gives the numbers of valid functional bugs of these apps, respectively.

2.2 Analysis Methods of Functional Bugs

To answer ***RQ1***, ***RQ2*** *and* ***RQ3***, we focused on the 399 valid functional bugs. We analyzed (1) the root causes by inspecting the bug patches (and reproducing the bugs if necessary), and (2) the symptoms and test oracles by reproducing the corresponding bug (or reviewing the bug-reproducing videos/images in the bug report). To build the taxonomies, we adopted the open card sorting approach [52]. Specifically, the preceding analysis was done by an iterative process. In each iteration, 40 bugs were randomly selected, and the two co-authors independently studied each of them. According to their own understanding, they independently labelled each bug with the categories of root causes, symptoms and oracles. Afterwards, these two co-authors cross-validated and discussed the labels until they reached consensus on the categorized results. When they could not reach consensus, the other two co-authors participated in the discussion to help make the final decision. Such an iteration was repeated ten times until all the 399 bugs were analyzed. We observed that this iterative process converged on the categories after the 2~3 rounds. This manual analysis process requires considerable domain-specific knowledge of both Android and the app logic, which took us around *six* person months.

*To answer **RQ4***, we studied whether existing automated testing tools can find the 399 functional bugs in our dataset. First, we manually analyzed how many bugs are within a tool's capability scope based on its testing technique. Second, we ran the tool to analyze its actual capability in finding the bugs within scope.

Representativeness of our study. *To validate the representativeness of the analysis results in **RQ1**, **RQ2** and **RQ3***, we also investigated the functional bugs contained in a third-party dataset, AndroR2 [28, 68]. We aim to validate whether the analysis results on our own bug dataset are representative and consistent with those based on AndroR2. Note that AndroR2 does not investigate functional bugs. Section 7 will discuss our analysis on AndroR2.

3 RQ1: ROOT CAUSES

In this section, we focus on the 399 functional bugs in our dataset to study the root causes. We successfully identified the root causes of 393 functional bugs (we failed to analyze 6 bugs due to limited debugging information and high complexity), and classified them into three major categories: (1) *General Programming Error* (Section 3.1), (2) *Android Related Error* (Section 3.2), and (3) *Third-party library Bugs/Limitations* (Section 3.3). These three categories include 15 subcategories. Table 3 lists these root causes: *Category* gives the three major categories, *Subcategory* gives the subcategories and *Description* describes the root cause. To be concise, we put the descriptions of root causes in table 3 and explain them as follows.

3.1 General Programming Error

General programming error denotes the errors that frequently occur in classic software development. We find that 223 out of 393 (56.7%) functional bugs were induced by *general programming error*, which we classified into seven subcategories of root causes (see Table 3). Specifically, we followed the operational bug classification strategy [32, 60] to achieve disjoint classification: a bug that is classified into the higher subcategory will not be considered in the lower one, and the higher subcategory indicates the wider scope (*e.g.*, feature-level errors) and the lower one indicates the narrower scope (*e.g.*, code- *or* variable-level errors). In the following, we illustrate these subcategories described in Table 3 one by one.

Missing Features (Example: *Amaze*'s Issue #2518). In this bug, clicking the `Exit` button on the app's main page does not exit. The reason is that this feature was not implemented. Figure 2(a) shows the patch, in which the function code for the `Exit` button (denoted by `R.id.exit`) is added (Lines 3-5).

Missing Cases (Example: *K-9 Mail*'s Issue #4984). In this bug, if a user sends an email with an empty text body, this mail cannot be found in the directory of the sent emails. The expected feature is that all the sent emails should be in that directory. The reason is that the usage case of any email with an empty text body was not implemented, and thus the app failed to show such emails. Figure 2(b) shows the patch, in which an additional checking is added to implement this usage case (Line 2).

Finding 1: *Missing cases* affects 15.8% of the 393 functional bugs, which is one of the most common root causes.

Wrong Control Flow (Example: *K-9 Mail*'s Issue #4374). In this bug, a user reports that the subject of a sent email is unexpectedly encrypted. The root cause is that the condition of deciding whether an email's subject should be encrypted was incorrect. Figure 2(c) shows the patch, in which a flag variable `shouldEncrypt` is added to decide whether the subject should be encrypted (Line 4).

Lack of Data Synchronization (Example: *NewPipe*'s Issue #6419). In this bug, the app failed to update the text showing the playback speed of videos on the screen after the speed value was changed by a user. Figure 2(d) shows the patch, in which the text on the UI page is synchronized with the playback speed data (`onPlaybackChanged()`) after the speed value (stored in `playbackTempo`) is changed (Line 5).

Improper Exception Handling (Example: *AnkiDroid*'s Issue #7023). In this bug, when a user exports a deck whose images were deleted, *AnkiDroid* displays an error message "`Error exporting apkg file`". Figure 2(e) shows the patch, in which a condition is added to handle the exception of file inexistence (*i.e.*, checking the existence of images) before exporting (Lines 1-4).

Third-party Library Usage Issue (Example: *NewPipe*'s Issue #5895). In this bug, users cannot load videos on the trending page. The reason is that *NewPipe* used a third-party library API from Youtube to access the title of the trending page. But the API was upgraded by storing the title in a raw String rather than in a `Json` object. As a result, *NewPipe* failed to load the video information by using the outdated API. The bug was fixed by updating the API usage.

Incorrect Variable Assignment (Example: *Amaze*'s Issue #1872). In this bug, the app failed to open a directory because it mistakenly treated the directory as a file. The root cause is that a boolean variable `isDirectory` of the directory was erroneously assigned as `false`, which caused the directory to be recognized as a file.

Other Wrong Functionality Implementations (Example: *Firefox*'s Issue #3563). In this bug, *Firefox* implements a fingerprint lock feature to hide the private tabs created by a user from being accessed by others. However, when a user uses this feature to hide the created private tabs, other users can still view these private tabs if they create their own new tabs. However, the app design requirement is that the locked private tabs should not be accessed unless the user himself/herself unlocks the private tabs. The root cause is that the implementation did not meet the design requirement.

3.2 Android Related Error

Android related error denotes the errors which are related to Android's specific features and induced by inadequate understanding and/or incorrect enforcement. We find that 164 out of 393 (41.7%)

```
1 +  public boolean onOptionsItemSelected() {
2 +     ...
3 +     case R.id.exit:
4 +        ((MainActivity) requireActivity()).goToMain();
5 +        return true;
```

(a) Missing features: Amaze's Issue #2518

```
1    private fun extract(text: String) {
2 +    if (text.isEmpty())  return ""
3      ... // extract the text
4    }
```

(b) Missing cases: K-9 Mail's Issue #4984

```
1    private void startOrContinueBuildMessage() {
2       ...
3 -    if (isEncryptSubject()) {
4 +    if (isEncryptSubject() && shouldEncrypt) {
5          moveSubjectIntoEncrypted();
```

(c) Wrong control flow: K-9 Mail's Issue #4374

```
1    public void onPlaybackChanged(final float playbackTempo,
2                  final float pitch, final boolean skipSilence) {
3       if (player != null) {
4          player.setPlayback(playbackTempo, pitch, skipSilence);
5 +       onPlaybackChanged(player.getPlayback());
```

(d) Lack of data synchronization: NewPipe's Issue #6419

```
1 +   if (!file.exists()) {
2 +      Timber.d("Skipping missing file %s", file);
3 +         continue;
4 +   }
```

(e) Improper exception handling: AnkiDroid's Issue #7023

```
1 +  override fun onSaveInstanceState(outState: Bundle) {
2 +     super.onSaveInstanceState(outState)
3 +     outState.putString(ACCOUNT, accountUuid)
4 +     outState.putLongIfPresent(FOLDER, folderId)
5 +     outState.putString(NAME, folderDisplayName)
6 +  }
7 +
8 +  private fun restoreInstanceState(savedInstanceState: Bundle) {
9 +     val accountUuid = savedInstanceState.getString(ACCOUNT)
10 +    if (accountUuid != null) {
11 +       val folderId = savedInstanceState.getLongOrNull(FOLDER)
12 +       val folderDisplayName= savedInstanceState.getString(NAME)
```

(f) Android mechanism related error: K-9 Mail's Issue #4936

```
1    private onTouchEvent(MotionEvent event) {
2       ...
3       HiTestResult hr = getHitTestResult();
4 -     return  onImageClick();
5 +     if (isValidInstagramImageClick(hr)){
6 +        return super.onTouchEvent(event);
7 +     else {
8 +        return onImageClick();
9 +     }
```

(g) Android mechanism related error: WordPress's Issue #12767

```
1    public boolean onOptionsItemSelected() {
2       ...
3 -     case R.id.clear_history:
4 +     case R.id.clear_logs_item:
5          DBWriter.clearDownloadLog()
6          return true;
7       ...
```

(h) Android resource related error: AntennaPod's Issue #4656

```
1    public void rename() {
2     ...
3       getMainActivity().mainActivityHelper
4          .rename(
5 -          Uri.parse(PATH).buildUpon().
6 -              appendPath(name1).build().toString(),
7 +           Uri.parse(PATH).buildUpon().
8 +               appendEncodedPath(name1).build().toString());
```

(i) Android framework API Misuse: Amaze's Issue #2113

Figure 2: Illustrative examples for explaining root causes (the code snippets are simplified).

functional bugs belong to *Android related error*, which we classified into five disjoint subcategories of root causes according to the involved Android features.

Android Mechanism Related Error Specifically, we observe that the functional bugs in this subcategory share some common root causes: mishandling lifecycle, input events or back stack.

Mishandling lifecycle **(Example: *K-9 Mail* Issue #4936).** Mishandling the lifecycle of Android components (*e.g.*, `Activity`, `Fragment`) when specific events happen (*e.g.*, rotating screens, putting the app into background and returning back, the app is killed by the system) could lead to functional bugs. In this bug, the user account got lost if the screen is rotated. The reason is that *K-9 Mail* did not save and restore the important variables (`accountUuid`, `folderId`, `folderDisplayName`) which record the account information. Figure 2(f) shows the patch, in which these variables are saved in callback `onSaveInstanceState()` (Lines 3-5) and restored in callback `restoreInstanceState()` (Lines 9-12) to handle the change of lifecycle states due to screen rotation.

Mishandling input events **(Example: *WordPress*'s Issue #12767).** In Android, handling user or system events is achieved by overriding event callbacks [3]. However, improper handling may lead to functional bugs. In this bug, when a user clicks a thumbnail of an `Instagram` image, the app fails to preview the image and prompts an error message "`Unable to view image`". The reason is that the image cannot be previewed by the default photo viewer (Line 4). Figure 2(g) shows the patch, in which the user event (click) is routed to `super.onTouchEvent()`, which later will be handled by `WebView`, if the thumbnail denotes an Instagram image (Lines 5-6).

Finding 2: *11.5% of the 393 functional bugs are induced by Android mechanism related error due to the lack of good understanding of Android mechanisms.*

Android Resource Related Error (Example: *AntennaPod*'s Issue #4656). In this bug, clicking the `Trash` button on the `Log` page cannot clear the download logs. The reason is that the resource id of the `Trash` button was incorrect. Figure 2(h) shows the patch, in which the resource id of the `Trash` button is corrected (Lines 3-4).

Finding 3: *Android resource related error* is also one of the most common root causes affecting 15.8% of the 393 bugs.

Android Framework API Misuse (Example: *Amaze*'s Issue #2113). In this case, the file renaming function is broken if the new file name contains space characters. The root cause is that the used API `appendPath()` converts any space character to "%20" in ASCII encoding [61]. Figure 2(i) shows the patch, in which `appendEncodePath()` is used instead, which converts a space character into plain text.

Android Compatibility Issue (Example: *AntennaPod*'s Issue #3986). In this bug, some texts overlap the statistics chart on the UI page when they use the app on a small-screen phone under some specific language (*e.g.*, German). The root cause is that the app fails to adapt its UIs to different languages with longer texts.

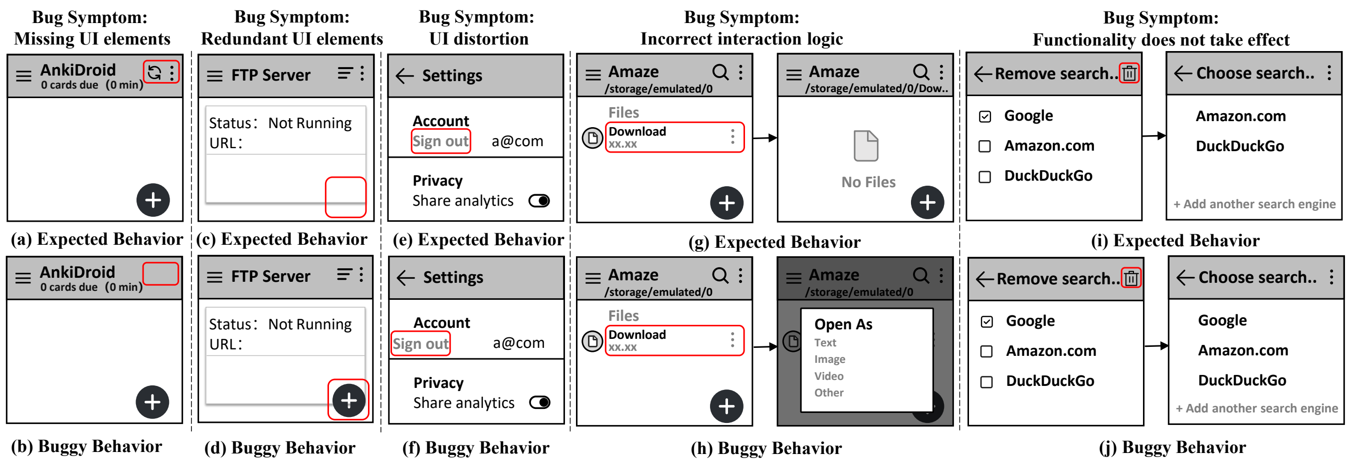

Figure 3: Illustrative examples of bug symptoms. The red boxes indicate the clues of bug symptoms.

Android Framework Bugs/Limitations (**Example:** ***Simplenote***'**s Issue 702**). In this bug, when a user sets the dark theme, the app's toolbar did not follow the dark theme. A bug in Android library `AppCompat` [19] makes `WebView` fail under the dark theme.

3.3 Third-Party Library Bugs/Limitations

It is the least common type of root cause which is responsible for only 6 out of the studied 393 functional bugs.

Finding 4: Functional bugs in Android apps have diverse root causes, which include 15 subcategories from the 3 major categories, *i.e.*, *general programming error*, *Android related error* and *third-party library bugs/limitations.*

4 RQ2: BUG SYMPTOMS

In this section, we focus on the 399 functional bugs in our dataset to study their symptoms. Considering Android apps are UI-based, interactive software, we group the symptoms of functional bugs into two major types: (1) *UI Display Issue* (Section 4.1), and (2) *UI Interaction Issue* (Section 4.2), in addition to other specific symptoms (Section 4.3). Figure 4 gives the taxonomy of bug symptoms.

4.1 UI Display Issue

Since a UI page of an app is composed of two major components, *i.e.*, the UI structure (*i.e.*, the UI layout in the form of a tree) and the content (*e.g.*, texts), *UI display issue* contain two groups of issues, *i.e.*, *UI Structure related Issue* and *Content related Issue*.

UI structure related issue. We find that 161 out of the 399 bugs (accounting for 40.4%) belong to *UI structure related issue*. Specifically, this symptom includes three different subgroups: (1) *Missing UI Elements*, (2) *Redundant UI Elements* and (3) *UI Distortion*.

- **Missing UI elements** (18.0%≈72/399 bugs). The impact of a bug with this symptom is that some UI element (*e.g.*, widgets) unexpectedly disappears on a UI page. For example, in *AnkiDroid*'s Issue #4951 (see Figure 3(a) and (b)), the "*sync*" and "*menu*" buttons disappear (but should be there).
- **Redundant UI elements** (8.5%≈34/399 bugs). The impact of a bug with this symptom is that some UI element unexpectedly appears on a UI page. For example, in *Amaze*'s Issue #1933 (see Figure 3(c) and (d)), when a user rotates the device screen from portrait to landscape and rotates back, a floating action button appears (but should not be there).
- **UI distortion** (13.8%≈55/399 bugs). *UI distortion* denotes those minor displaying issues of UI elements, *e.g.*, misaligned UI elements or asymmetric shapes [18], which affect the usability of app functionalities. For example, in *Simplenote*'s Issue #728 (see Figure 3(e) and (f)), the option "Sign out" is not aligned with the other UI elements.

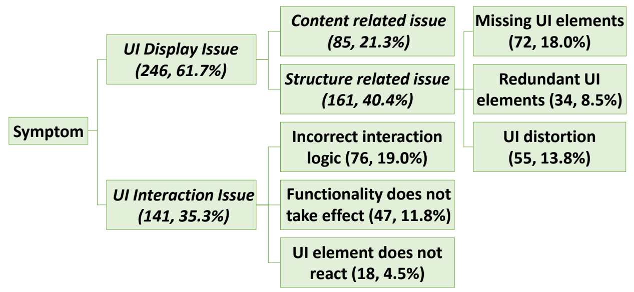
Symptom
UI Display Issue (246, 61.7%)
UI Interaction Issue (141, 35.3%)
Content related issue (85, 21.3%)
Structure related issue (161, 40.4%)
Incorrect interaction logic (76, 19.0%)
Functionality does not take effect (47, 11.8%)
UI element does not react (18, 4.5%)
Missing UI elements (72, 18.0%)
Redundant UI elements (34, 8.5%)
UI distortion (55, 13.8%)

Figure 4: The taxonomy of bug symptoms. The numbers (X, Y) underneath the category name give #functional bugs in this category and the percentage among all the 399 bugs.

Content related issue. 85 out of the 399 bugs (accounting for 21.3%) belong to this symptom. The impact of a bug with this symptom is that the content is not displayed as expected. For example, in *K-9 Mail*'s Issue #4872, when a user writes a URL "`http://example.com`" in a draft, closes and reopens the draft, the content becomes "`http://example.com <http://example.com>`". The expected behavior is that the draft's content should not be changed.

4.2 UI Interaction Issue

UI interaction issue is another major bug symptom, in addition to *UI display issue*. Typically, *UI display issue* can be identified on a single UI page, while identifying UI Interaction issue requires examining some consecutive UI pages. As shown in Figure 4, *UI interaction issue* includes the following three subgroups: (1) *Incorrect Interaction Logic*, (2) *Functionality Does Not Take Effect*, and (3) *UI Element Does Not React*.

Table 4: Distributions of 399 functional bugs in terms of three oracle types.

App	#Bugs	Oracle Types		
		#AFA	#AFR	
			#AIF	#ASF
Simplenote	35	16	11	8
AnkiDroid	82	22	20	40
Amaze	30	14	8	8
K-9 Mail	36	14	8	14
NewPipe	65	22	14	29
AntennaPod	41	15	8	18
WordPress	67	19	14	34
Firefox	43	11	17	15
Total	399	133	100	166

Incorrect interaction logic (19.0%≈76/399 bugs). The impact of a bug with this symptom is that the app performs incorrect UI page transitions or state reactions, which violates the original functional logic. For example, in *Amaze*'s Issue #1872 (see Figure 3(g) and (h)), when a user clicks the directory `Download`, the app erroneously shows a file opening dialog (intended for a normal file). The expected behavior is that the app should directly open `Download`.

Functionality does not take effect (11.8%≈47/399 bugs). The impact of a bug with this symptom is that the app disrespects the user intended functionality. For example, in *K-9 Mail*'s Issue #3866, after a user turns off the background synchronization option in the app preference, the app disrespects this configuration and still does background synchronization. Another example is in *Firefox*'s Issue #4739 (see Figure 3(i) and (j)), when a user selects the item `Google` and tries to delete it by clicking the trash button, the app fails to delete `Google` (which is still there).

UI element does not react (4.5%≈18/399 bugs). The impact of a bug with this symptom is that the app does not react when some executable UI element is exercised. For example, in *Amaze*'s Issue #2518, when a user clicks the `Exit` button on the app's main page, the `Exit` button does not react.

4.3 Other Symptoms

The remaining 12 functional bugs show other specific symptoms, *e.g.*, infinite loading, UI element flashing and failed animations.

Finding 5: Functional bugs have 7 major symptoms of two major types, *i.e.*, *UI display issue* and *UI interaction issue*. Specifically, *Content related issue* (21.3%) and *Incorrect interaction logic* (19%) are the two most common symptoms.

5 RQ3: TEST ORACLES

In this section, we investigate what kinds of test oracles are needed to find these functional bugs. Because test oracles are crucial for bug manifestation, in addition to test inputs. Specifically, we examined the bug-reproducing videos/images (including the executed events and UI pages) of all the 399 functional bugs to identify the oracles. We classified the oracles into two major groups *from the view of the required knowledge on app features*: (1) *app feature agnostic oracle* (*AFA* for short) and (2) *app feature related oracle* (*AFR* for short), which contains two subgroups, *i.e.*, *app independent feature oracle* (*AIF* for short) and *app specific feature oracle* (*ASF* for short). Table 4 gives the disjoint classifications of all the 399 functional bugs based on these three oracle types. We explain these oracles as follows.

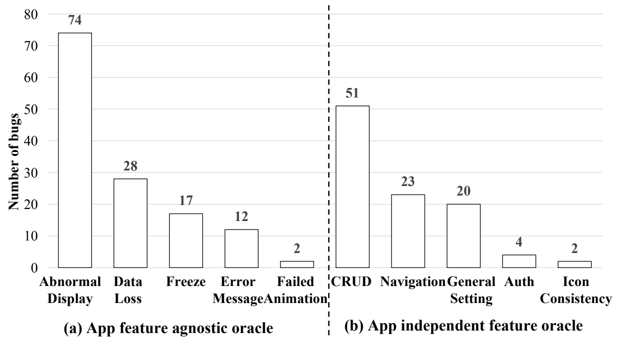

Figure 5: Distributions of the oracles in AFA and AIF.

App feature agnostic oracle (AFA). This type of oracles is defined based on *the implicit knowledge of correct app behaviors without knowing app features*. Thus, this type of oracle is most general (and could be automatically generated) for finding functional bugs. Figure 5 (a) shows the distributions of different oracles in this oracle group. *Abnormal Display* checks common abnormal display issues (*e.g.*, overlapped texts, misaligned UI elements), which is the dominant case (74 out of 133 bugs) in this group. *Data Loss* checks whether the user data is properly preserved after specific user actions (*e.g.*, rotating screens, putting the app to the background and returning back). *Freeze* checks any executable UI element should react to user actions (*e.g.*, a `clickable` button should always react to a `click` action). *Error Message* checks typical error messages (*e.g.*, "Network error") which indicate something went wrong. *Failed Animation* checks the failures of some animation effects.

App feature related oracle (AFR). This type of oracles requires some knowledge of app features. Specifically, this oracle type contains two subgroups:

- ***App independent feature oracle (AIF)***. This type of oracles is related to app independent features, which are implemented similarly across different apps and thus we have the common knowledge of the expected behaviors. The typical examples of this oracle type are those common UI design patterns [2, 39, 43, 62]. Figure 5(b) shows the distributions of different oracles in the oracle group. *CRUD* [38] checks the functionality of *create, read, update* and *delete* (*e.g.*, the number of items should increase or decrease after a creation or deletion operation is executed). *Navigation* checks the functionality of specific navigation (*e.g.*, the app should be navigated to the previous UI page after pressing `Back`). *General Setting* checks the functionality of specific setting (*e.g.*, the texts of an app should respect the selected language setting). *Auth* checks the functionality of user authentication (*e.g.*, the login operation should succeed when the credential is correct). *Icon Consistency* checks the consistency between the icon and the corresponding functionality (*e.g.*, a deletion icon should do the deletion operation).
- ***App specific feature oracle (ASF)***. This type of oracles is related to the app specific logic and requires the knowledge on an app's own functionalities. It is hard to be automatically generated. For example, in *AnkiDroid*, if a user suspends a learning card, the color behind the card should be changed.

In Table 4, we find that 33.3% (133/399) of functional bugs can be observed by *app feature agnostic oracle* (see *#AFA*), while 66.7%

Table 5: Results of existing testing tools on finding the 399 functional bugs. "-" denotes the tool is not available.

Tool	Testing Technique	Target Bugs	#Bugs in Scope	#Tested Bugs	#Found Bugs
DiffDroid	Differential testing	Compatibility	27	-	-
OwlEye	Deep learning	UI display	14	-	-
iFixDataloss	Implicit knowledge	Data loss	8	6	0
ITDroid	Differential testing	I18n-related	1	1	1
SetDroid	Metamorphic testing	Setting-related	18	13	1
Genie	Metamorphic testing	Functional	2	2	0
ODIN	Implicit knowledge	Functional	20	14	0
#Total			84	33	2

(266/399) of the bugs require *app feature related oracle* (see *#AFR*). Specifically, 25.1% (100/399) of the bugs can be captured by *app independent feature oracle* (see *#AIF*), while 41.6% (166/399) of the bugs require *app specific feature oracle* (see *#ASF*).

Limitations of prior studies. Zaeem *et al.* [69] (listed in Table 1) only identified the oracle of *Data Loss* in *app feature agnostic oracle* and missed some major oracles (*e.g.*, *Abnormal Display*, *Freeze*, *Error Message*). They also missed some major oracles (*e.g.*, *CRUD*, *Navigation*) in *app independent feature oracle.*

Finding 6: Among all the studied 399 functional bugs, 33.3% (133/399) of the bugs can be observed by *app feature agnostic oracle*, while 66.7% (266/399) require *app feature related oracle.*

6 RQ4: STATUS OF EXISTING TECHNIQUES

This section examines the status of existing automated testing techniques in finding functional bugs. We analyze how many of the 399 functional bugs in our dataset could be found by existing tools.

Analysis Method. We surveyed the relevant work which targets functional bugs in Android apps in the literature. Specifically, we focus on those testing tools which can *automatically* generate inputs and oracles. Table 5 lists the selected representative testing tools: *"Tool"* gives the tool name, *"Testing Technique"* gives the main testing technique used by the tool, *"Target Bugs"* gives the type of functional bugs the tool targets. DiffDroid [16] compares the GUI pages of an app on two different devices to find compatibility issues. OwlEye [36] uses deep learning to detect UI display issues, especially UI distortions (*e.g.*, overlapping texts, component occlusion). iFixdataloss [21] is the most recent work of finding data loss issues among the others [45, 46, 69]. ITDroid [13] compares the GUI pages of an app under two different language settings to find internationalization (i18n) issues. SetDroid [58, 59] uses metamorphic testing to find system setting-related functional bugs. Genie uses the metamorphic relation of independent view property [57] to find bugs, while Odin [65] uses the implicit knowledge of "bugs as deviant behavior" [12] to find bugs. Genie [57] and Odin [65] are the only two tools that target generic functional bugs, while the others target specific types of functional bugs.

To investigate these tools, we conducted a *two-step* analysis method. In the first step, we *manually* analyzed which functional bugs in our dataset are within the tool's capability scope. To do this analysis, we (1) read the corresponding paper and examine the tool (if available) to understand its technique; and (2) check whether the oracle generated by the tool can capture a bug according to its symptom and/or root cause (assuming the tool can always generate the necessary inputs reaching the bug). Thus, the analysis results can be interpreted as the *optimistic upper bound* of the tool's capability (without considering the effect of other factors, *e.g.*, allocated testing time and specific device types, which may affect the tool's practical effectiveness). It also allows us to analyze those unavailable tools (DiffDroid and OwlEye). Before the analysis, we talked with Genie's and Odin's authors to validate our understanding on their techniques which are more sophisticated than the other tools.

In the second step, we ran each tool on the bugs within its scope to validate its actual capability. We did not run DiffDroid and OwlEye because they are not publicly available. We followed the default setup (*e.g.*, allocated testing time, running parameters) of each tool in our experiment. If an app requires some initial setup (*e.g.*, user login), we provided the tool with a script to complete the setup. After the testing, we inspected the running results of each tool to examine whether the tool found the bugs.

Analysis Results. In Table 5, *"#Bugs in scope"* gives the numbers of bugs within the scope of the tool based on our manual analysis, *"#Tested Bugs"* gives the number of bugs tested by the tool (recall that our bug dataset contains 134 bugs which are not reproducible but have clear reproducing videos/images for bug analysis in Section 2; such bugs are excluded in this step) and *"#Found bugs"* gives the number of bugs found by the tool. We can see that only 21% (≈84/399) of the 399 functional bugs are within the capability scope of all these tools. It indicates that the oracles of these tools are limited in finding functional bugs. Moreover, only 2 out of 33 tested bugs are found by these tools. By analyzing the testing results of these tools, we find that the major reason is that their generated inputs are of low quality and hard to reach the buggy scenarios. In practice, Genie and Odin report many false positives.

Finding 7: Most existing tools only target specific types of functional bugs. Only 21% of the 399 functional bugs are within the capability scope of existing automated testing tools due to the limited oracles. Moreover, these tools can only find 2 bugs due to the low quality of the generated inputs.

7 REPRESENTATIVENESS OF OUR STUDY

In this section, we validate the representativeness of our study by analyzing another bug dataset of Android apps, AndroR2 [28, 68]. This dataset was *not* constructed by the authors of this paper. It contains 180 manually-reproduced bug reports (corresponding to 50 crashing and 130 non-crashing functional bugs). Prior to our work, it was the largest publicly available bug dataset for Android apps. It was systematically constructed based on the issues from GitHub. Specifically, to avoid overfitting to a specific app, AndroR2 only includes *at most* 10 randomly-selected, reproducible bugs per app. For example, the 130 functional bugs in AndroR2 come from 42 different apps (on average 3 bugs per app). Thus, AndroR2 complements our dataset because we focus on selecting a broad range of bugs (spanning three years) from eight representative apps. *Note that AndroR2 focuses on studying the reproducibility of bug reports rather than analyzing functional bugs.*

We inspected the 130 functional bugs in AndroR2 by following the similar analysis methods of **RQ1~RQ3**. Among these 130 bugs, 109 bugs have fixing patches and only 6 bugs overlap with our dataset. Our goal is to validate (1) whether the root causes, symptoms and oracles of these 130 bugs are covered by our dataset, and

(2) whether the analysis results are consistent with ours. Specifically, we manually reproduced the 130 bugs to analyze their symptoms, oracles, and analyzed the root causes of 109 bugs by inspecting the patches. We find that (1) these bugs' root causes, symptoms and oracles are all covered by our results, and (2) the analysis results are consistent with ours. For example, for the root causes, *missing cases* and *Android resource related error* are the two most common faults, affecting 14.7% (16/109) and 15.6% (17/109) of these bugs. For the bug symptoms, *content related issue* (18.5%≈24/130) and *incorrect interaction logic* (24.6%≈32/130 bugs) are the two most common symptoms. For the test oracles, 30% (39/130) of these bugs can be observed by *app feature agnostic oracle*, while 70% (91/130) of these bugs require *app feature related oracle*. To sum up, our analysis results based on our own dataset of 399 bugs are representative.

8 IMPLICATIONS AND DISCUSSIONS

In this section, we discuss the implications obtained from our study (**RQ1**~**RQ4**) and shed light on what developers and researchers could do to tackle functional bugs in Android apps.

8.1 Avoiding and Diagnosing Functional Bugs

Missing cases is one of the most common root causes affecting many (15.8%) functional bugs (Finding 1, **RQ1**). It indicates that, in addition to the *main* usage cases, developers should pay attention to the *alternative* usage cases from the view of app users when developing or testing some app functionality. For example, for a notebook app (like *Simplenote*), developers should consider that users may input special characters which may lead to garbled text. In fact, our inspection on all the developer-written tests of the eight apps in our study shows that many (73.7%) of these tests missed the functional bugs because of *missing cases*.

Android resource related error is another one of the most common root causes (Finding 3, **RQ1**) affecting many (15.8%) functional bugs. It mainly leads to *UI distortion*, *missing UI elements* and *content related issue*. In some cases, this error occurs because developers mistake the resource ids between some UI elements (see Figure 2(h)). To counter this, developers could (1) avoid ambiguity when defining resource ids and (2) check whether there are some resource ids *defined* in the resource files but have never been *used* in the app code. In other cases, some hard-coded strings left in the app code lead to displaying incorrect texts or icons. Developers can use Lint [35] to avoid such bad programming practices.

By analyzing the correlation between bug symptoms (**RQ2**) and root causes (**RQ1**), we find that (1) most (78%) of *UI distortion* are induced by *Android resource related error* (42%), *Android compatibility issue* (20%) and *missing cases* (16%), and (2) many (61%) of *UI element does not react* are induced by resource related error (39%) and missing feature (22%). Thus, developers could use such information of the correlation to aid diagnosing functional bugs.

8.2 Finding Functional Bugs

Choosing appropriate oracles to find functional bugs. Our study reveals that 33.3% of the functional bugs can be found by *app feature agnostic oracle*, while 66.7% of the functional bugs require *app feature related oracle* (Finding 6, **RQ3**). Moreover, by analyzing the correlation between bug symptoms (**RQ2**) and test oracles (**RQ3**), we find that (1) *app feature agnostic oracle* can capture many *UI distortion* (73%) and *UI element does not react* (78%), respectively, while (2) *functionality does not take effect* (80%) and *Redundant UI elements* (71%) can only be captured by *app feature related oracle*. These results indicate that choosing appropriate oracles is important in effectively finding functional bugs (*e.g.*, balancing the implementation effort and the effectiveness of the oracles).

For example, to find the bugs of *UI element does not react*, we can implement an app agnostic oracle, *i.e.*, exercising any executable (*e.g.*, `clickable`) UI element should always lead to some UI effects. On the other hand, to find app feature related bugs, we have to use the derived oracles or specified oracles [5]. In the case of derived oracles, we could use metamorphic testing [7] or differential testing [40] to derive more oracles and overcome the oracle limitations in existing automated testing tools (Finding 7, **RQ4**). Section 9 will demonstrate the feasibility of adapting differential testing to find functional bugs. In the case of specified oracles, property-based testing [9] could be an ideal approach. Because we can specify the knowledge of app functionalities as some properties [38, 42] and then automatically validate functional correctness by generating different inputs. To our knowledge, little effort adapts property-based testing to find functional bugs in Android apps [30].

Effective input generation is still crucial. Although existing automated testing tools have designed automated oracles to find functional bugs, their generated inputs are hard to cover meaningful app functionalities (Finding 7, **RQ4**). Thus, researchers should design more effective input generation techniques to generate context-aware inputs that meaningfully interact with the app's functionalities, and thus increase the chance of finding functional bugs.

9 FINDING FUNCTIONAL BUGS VIA DIFFERENTIAL TESTING

According to **RQ3**, 66.7% of functional bugs can only be found by *app feature related oracle*. This finding inspires us to adapt the idea of *differential testing* to overcome the oracle problem. Because differential testing can compare the behaviors of the tested app with those of a reference app (with similar functionalities) to identify likely functional bugs without specifying app features. Moreover, inspired by the results of **RQ2**, we find that a prior version of the tested app can be used as the reference app (which likely has similar functionalities). For example, we can compare the two app versions, *e.g.*, corresponding to Figure 4(a) *v.s.* (b) and Figure 4(g) *v.s.* (h), to find those functional bugs of *missing UI elements* or *incorrect interaction logic*, respectively. Specifically, we identify the functional inconsistencies between the two app versions by checking whether some event trace from the prior app version is *executable* on the tested app version. In this way, we can find both *UI display issue* and *UI interaction issue*. To our knowledge, we are the *first* to apply differential testing on two different versions of the same app to find functional (regression) bugs (discussed in Section 11).

9.1 Approach Design and Implementation

We implemented the preceding idea as an automated differential testing tool named RegDroid to find functional bugs. Figure 6 depicts the workflow of RegDroid. At the high-level, RegDroid generates random GUI tests (in the form of UI event sequences),

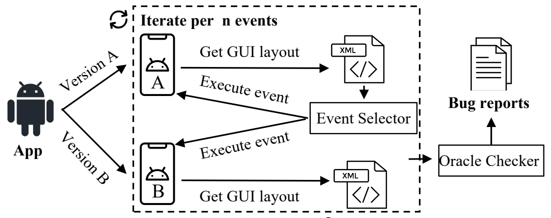

Figure 6: Overview of RegDroid.

Table 6: Statistics of the 14 functional bugs found by RegDroid from five popular open-source apps.

App Name	#Stars	ID	Bug State	New Bug?	Bug Symptom
AnkiDroid	5.4K	1	Fixed	Yes (#12053)	Incorrect interaction logic
		2	Fixed	Yes (#11220)	UI element does not react
		3	Fixed	No	Incorrect interaction logic
		4	Fixed	Yes (#11363)	UI element does not react
Amaze	4.0K	5	Fixed	Yes (#3378)	Missing UI elements
		6	Fixed	Yes (#3394)	Redundant UI elements
		7	Fixed	No	Missing UI elements
		8	Fixed	No	Redundant UI elements
AntennaPod	4.4K	9	Fixed	Yes (#5977)	Missing UI elements
		10	Fixed	Yes (#5863)	UI Element does not react
Markor	2.4K	11	Fixed	Yes (#1800)	Functionality does not take effect
Omni-Notes	2.5K	12	Fixed	Yes (#865)	UI Element does not react
		13	Fixed	Yes (#867)	Functionality does not take effect
		14	Fixed	No	Missing UI elements

runs these tests separately on the two app versions (*e.g.*, version A and B), and checks whether the GUI pages of version A and B (along the GUI tests) contain the similar UI widgets. Specifically, RegDroid runs two identical Android devices in parallel to improve testing efficiency. It includes two key modules: (a) *event selector* and (b) *oracle checker*, which we explain as follows.

Event selector. This module is responsible for generating random GUI tests. It *randomly* selects one executable GUI widget w from the current GUI page ℓ_A of app version A and generates the corresponding event e (*e.g.*, `click`, `edit`, `scroll`) according to w's widget property (*e.g.*, `clickable`, `editable`, `scrollable`). Later, this event e will be executed on version A and B, respectively. Note that the oracle checker module will be called before the event execution (which we explain later). This module will be iteratively called until generating n events, or some inconsistency is found (in this case, RegDroid will restart the app from fresh and test it again).

Oracle checker. This module is responsible for oracle checking. It is called before the actual execution of the selected event e. Specifically, since the UI widget w exists on the current GUI page ℓ_A of app version A, this module checks whether w also exists on the corresponding GUI page ℓ_B of app version B. Specifically, we check whether the GUI page ℓ_B contains the same UI widget whose `resource id` and `class name` are identical to those of w. If w does not exist on ℓ_B, RegDroid finds the inconsistency and reports a likely bug. Otherwise, e will be executed on version A and B, respectively. After the execution, *event selector* is called.

RegDroid is implemented in `Python`, and uses uiautomator2 [63] to parse UI widgets from the UI pages and send UI events.

9.2 Evaluation Setup and Results

Evaluation Setup. We applied RegDroid on five popular, long-maintained open-source apps (see Table 6). Among these five apps, three apps are selected from our studied subjects, *i.e.*, *AnkiDroid*, *Amaze*, *AntennaPod*, which are convenient to setup on the machine for testing. The other two apps are *not* within our studied subjects, *i.e.*, *Markor* and *Omni-Notes*. The experiments were conducted on a 64-bit Ubuntu 20.04 machine and Android emulators (Android 8.0, Pixel XL). Specifically, we selected 56 versions of *AnkiDroid*, 19 versions of *Amaze*, 16 versions of *AntennaPod*, 19 versions of *Markor* and 16 versions of *Omni-Notes* for testing. Note that all these versions are continuously released on GitHub. RegDroid is configured to test any two continuous release versions of an app by generating 50 random GUI tests (each test contains 100 events), which took about 12 hours. Since we tested 5 apps and their 121 two-continuous release versions in total, the whole experiment took around 8 days = 121*12/8/24 (because at most 16 Android devices can run in parallel on one machine, we can test at most 8 two-continuous release versions in parallel on one machine). Afterwards, we inspected all the bugs reported by RegDroid. If a bug is a true positive, we tried to reproduce it on the latest app version to verify whether this is a new (unknown) bug or a fixed one. If it is a new bug, we filed a bug report to the app developers. Otherwise, we further checked whether this bug was reported by others or *silently* fixed by the developers.

Results. RegDroid successfully found 14 unique functional bugs. Among these 14 bugs, 10 are new bugs (which affect the latest released versions), 2 were silently fixed by the developers (which have not been reported by others), and 2 were reported by others and fixed. We submitted the bug reports for those 10 news bugs, all of which have been confirmed and fixed by the app developers. Table 6 shows these 14 functional bugs, including the app name, the number of stars on GitHub, the bug id, the bug state, whether it is a new bug affecting the latest released version (with the issue id on GitHub), and the bug symptom. We can see that the found bugs cover five different symptoms (*e.g.*, *missing UI elements*, *incorrect interaction logic*, *functionality does not take effect*). Specifically, among these 14 bugs, only 4 bugs (ID: 5, 6, 7, 14) can be detected by existing testing tools in Section 6. RegDroid complements existing testing tools in finding the remaining 10 functional bugs. As a by-product of the functional testing, RegDroid also found 12 crashing bugs.

We find that 6 out of the new 10 bugs (60%) have hidden in the apps for over six months *and* affected more than eight versions. For example, RegDroid found a functional bug in *Markor* (Bug ID: 11) in which selecting any option (*e.g.*, `search`, `select all`) in a menu list does not take effect in the landscape mode. This bug has been hidden in the app for three years. The developer quickly fixed this bug in a few hours after we reported it. We also received positive feedback from the developers, *e.g.*, *"Thanks for reporting!", "Thanks for the feedback!"* These results show that RegDroid is useful.

Discussion. In the experiment, RegDroid reported 205 bugs. Among these bugs, 73 are true positives (36%) and 132 are false positives (64%). Many false positives are duplicated (only 50 out of 132 are unique ones). We note that these false positives are caused by two major reasons: (1) some app feature of the current version was updated (removed or added) in the newer version (accounting for 93% cases); and (2) some bug in the current version was fixed in the newer version (accounting for 7% cases). In practice, RegDroid highlights the inconsistencies on the UI pages to ease bug inspection. It took us less than one hour to inspect all the 205 bugs.

Despite the false positive rate, our *simple* implementation (without any optimization) of RegDroid already shows its promise in finding hard-to-detect functional bugs, which complements existing tools. In fact, its positive rate (64%) is comparable to existing sophisticated, state-of-the-art functional testing tools Genie [57] and Odin [65], which respectively have 59% and 68% false positive rates. Moreover, it is feasible to avoid many false positives by choosing two "closer" app versions with fewer feature (UI) changes. It can be achieved by analyzing the app code to identify which code commits changed UIs. We leave such optimizations as future work.

10 THREATS TO VALIDITY

Our study may suffer from some threats to validity. First, our study collected 399 functional bugs from 8 apps. Our findings may not be general to all the apps. To mitigate this threat, these apps are selected carefully to ensure that their representativeness. These 8 apps are popular, actively-maintained and have different features. Moreover, we complement our study by studying a third-party dataset, AndroR2, which contains 130 bugs from 43 different apps. The results from these two datasets are consistent (Section 7). Second, our study involves manual analysis, which may bring some biases. To this end, in Section 3, 4, 5, two co-authors independently inspected the bugs and then cross-checked the results between themselves and the other co-authors to achieve consensus.

11 RELATED WORK

Empirical studies related to functional bugs in Android apps. Different from the prior work on studying crashing bugs [14, 26, 54], our work studies non-crashing functional bugs. To our knowledge, the most relevant work is from Hu *et al.* [23], Zaeem *et al.* [69] and Johnson *et al.* [28]. But all these work does not aim to conduct a systematic study on functional bugs (see Table 1). The other studies only investigate specific types (and a small portion) of functional bugs, *e.g.*, WebView bugs [25], compatibility issues [67], system setting-related defects [58], i18n bugs [13] and UI display issues [36]. In contrast, our work covers different functional bugs.

Finding functional bugs in Android apps. In addition to the automated testing tools discussed in Section 6, other tools require human inputs to find functional bugs. For example, some tools rely on human-written tests to find functional bugs. Thor [1] and ChimpCheck [30] could inject neutral events (*e.g.*, pause-stop-restart) into the human tests to check whether the assertions in the tests still hold. AppFlow [24], ACAT [47], AppTestMigrator [6] and CraftDroid [33] migrate the human tests written for one app to another similar app to validate functional correctness. Some tools require manually defining app functionalities in specification languages for bug finding, *e.g.*, Augusto [38] and FARLEAD-Android [29]. Static analysis is also used to find specific functional bugs with distinct faulty code patterns. For example, KREfinder [49] and LiveDroid [15] scan incorrect handling of specific data variables to find data loss issues. But these tools suffer from high false positives. Android Lint [35] is a popular static analysis tool for finding different issues in Android apps based on many predefined rules. However, it focuses on general code issues in Android apps and falls short in detecting app specific functional bugs.

Differential testing on Android apps. Our demo tool RegDroid adapts differential testing on two different versions of the same app to find functional bugs. Thus, RegDroid can be viewed as a *differential regression testing* technique. To our knowledge, we are the *first* to adopt this technique to test Android apps. For example, DiffDroid [16] and SPAG-C [34] also use differential testing, but they run the same app version on two different devices to find compatibility issues. In the direction of regression testing for Android apps, no prior work compares the UI pages between two app versions to find functional bugs. For example, Redroid [11] and ReTestDroid [27] analyze the updated app code to identify which prior tests can be reused to test the app updates. QADoird [50] analyzes which new activities or UI events should be targeted for regression testing. ATOM [31] updates the test scripts for the evolving apps. None of these work targets finding functional bugs.

12 CONCLUSION

In this paper, we conduct the first systematic study on 399 functional bugs to study their symptoms, root causes, test oracles, and the capabilities and limitations of existing testing techniques. From this study, we obtain new interesting findings and implications that help shed light on tackling functional bugs. Based on some study findings, we design a differential testing tool RegDroid to detect functional bugs in Android apps. RegDroid successfully demonstrates its usefulness and complements existing testing tools.

DATA AVAILABILITY

We have made all the artifacts (including the bug dataset and the source code of RegDroid) publicly available at *https://github.com/Android-Functional-bugs-study/home* for replication.

ACKNOWLEDGMENTS

We thank the anonymous ISSTA reviewers for their valuable feedback and the insightful comments from Zhao Zhang and Chao Peng from ByteDance on the bug symptoms of functional bugs. This work was supported in part by NSFC Grant 62072178, National Key Research and Development Program (Grant 2022YFB3104002), "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, National Key Research and Development Program (Grant 2020AAA0107800), and the Shanghai Collaborative Innovation Center of Trusted Industry Internet Software.

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)*. 83–93. https://doi.org/10.1145/2771783.2771786

[2] Anders Toxboe . 2022. *Design patterns.* Retrieved 2022-9 from https://ui-patterns.com/patterns

[3] Android. 2022. *Input events overview.* Retrieved 2022-9 from https://developer.android.com/develop/ui/views/touch-and-input/input-events

[4] AppBrain Team. 2023. *Most popular Google Play categories.* Retrieved 2023-2 from https://www.appbrain.com/stats/android-market-app-categories

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[6] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 54–65. https://doi.org/10.1109/ASE.2019.00016

[7] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 2020. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01, Hong Kong University of Science and Technology.

[8] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 429–440. https://doi.org/10.1109/ASE.2015.89

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP).* 268–279. https://doi.org/10.1145/351240.351266

[10] Compuware. 2013. *Users Have Low Tolerance For Buggy Apps - Only 16% Will Try A Failing App More Than Twice.* Retrieved 2020-5 from https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/

[11] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Redroid: A Regression Test Selection Approach for Android Applications. In *The 28th International Conference on Software Engineering and Knowledge Engineering (SEKE).* 486–491. https://doi.org/10.18293/SEKE2016-223

[12] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001).* 57–72. https://doi.org/10.1145/502034.502041

[13] Camilo Escobar-Velásquez, Michael Osorio-Riaño, Juan Dominguez-Osorio, Maria Arevalo, and Mario Linares-Vásquez. 2020. An Empirical Study of i18n Collateral Changes and Bugs in GUIs of Android apps. In *IEEE International Conference on Software Maintenance and Evolution (ICSME).* 581–592. https://doi.org/10.1109/ICSME46990.2020.00061

[14] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).* 408–419. https://doi.org/10.1145/3180155.3180222

[15] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. Livedroid: Identifying and preserving mobile app state in volatile runtime environments. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428228

[16] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 308–318. https://doi.org/10.1109/ASE.2017.8115644

[17] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 141–152. https://doi.org/10.1145/3213846.3213869

[18] Gil Bouhnick. 2019. *Visually distorted - when symmetrical UI looks all wrong.* Retrieved 2022-10 from https://www.mobilespoon.net/2019/08/visually-distorted-when-ui-looks-all.html

[19] Google. 2022. *Google Issue Tracker.* Retrieved 2022-9 from https://issuetracker.google.com/issues/37124582?pli=1

[20] Google. 2022. *Google Play Store.* Retrieved 2022-9 from https://play.google.com/store/apps

[21] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* 605–616. https://doi.org/10.1145/3533767.3534402

[22] Hacker News. 2013. *Tell Facebook: There's a severe bug when changing profile pics on the iOS app.* Retrieved 2022-8 from https://news.ycombinator.com/item?id=6456285.

[23] 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).* 77–83. https://doi.org/10.1145/1982595.1982612

[24] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT (FSE).* 269–282. https://doi.org/10.1145/3236024.3236055

[25] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A tale of two cities: how WebView induces bugs to Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE).* 702–713. https://doi.org/10.1145/3238147.3238180

[26] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2019. Characterizing Android-specific crash bugs. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft).* 111–122. https://doi.org/10.1109/MOBILESoft.2019.00024

[27] Bo Jiang, Yu Wu, Yongfei Zhang, Zhenyu Zhang, and W. K. Chan. 2018. ReTestDroid: Towards Safer Regression Test Selection for Android Application. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC).* 235–244. https://doi.org/10.1109/COMPSAC.2018.00037

[28] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* 321–322. https://doi.org/10.1109/SANER53432.2022.00048

[29] Yavuz Köroglu and Alper Sen. 2021. Functional test generation from UI test scenarios using reinforcement learning for android applications. *Softw. Test. Verification Reliab.* 31, 3 (2021). https://doi.org/10.1002/stvr.1752

[30] Edmund S. L. Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. ChimpCheck: property-based randomized test generation for interactive apps. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!).* 58–77. https://doi.org/10.1145/3133850.3133853

[31] Xiao Li, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li. 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).* 161–171. https://doi.org/10.1109/ICST.2017.22

[32] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID).* 25–33. https://doi.org/10.1145/1181309.1181314

[33] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 42–53. https://doi.org/10.1109/ASE.2019.00015

[34] Ying-Dar Lin, José F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. 2014. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *IEEE Trans. Software Eng.* 40, 10 (2014), 957–970. https://doi.org/10.1109/TSE.2014.2331982

[35] Lint Team. 2022. *Android Lint.* Retrieved 2022-10 from http://tools.android.com/lint/overview

[36] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 398–409. https://doi.org/10.1145/3324884.3416547

[37] Localytics. 2019. *25% of Users Abandon Apps After One Use.* Retrieved 2020-5 from http://info.localytics.com/blog/25-of-users-abandon-apps-after-one-use

[38] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: exploiting popular functionalities for the generation of semantic GUI tests with Oracles. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).* 280–290. https://doi.org/10.1145/3180155.3180162

[39] Martijn van Welie. 2008. *Pattern library.* Retrieved 2022-9 from http://www.welie.com/patterns/index.php

[40] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[41] Motherboard. 2020. *Here's the Shadow Inc. App That Failed in Iowa Last Night.* Retrieved 2020-5 from https://www.vice.com/en_us/article/y3m33x/heres-the-shadow-inc-app-that-failed-in-iowa-last-night

[42] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).* 1025–1038. https://doi.org/10.1145/3519939.3523728

[43] Dominik Pacholczyk. 2014. *Mobile UI Design Patterns A Deeper Look At The Hottest Apps Today.* www.uxpin.com.

[44] Reuters. 2021. *Japan's COVID-19 app failed to pass on some contact warnings.* Retrieved 2022-10 from https://www.reuters.com/article/us-health-coronavirus-japan-app-idUSKBN2A31BA

[45] Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino. 2018. Is this the lifecycle we really want?: an automated black-box testing approach for Android activities. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA).* 68–77. https://doi.org/10.1145/3236454.3236490

[46] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152. https://doi.org/10.1145/3395363.3397379

[47] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2018. Automation of android applications functional testing using machine learning activities classification. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 122–132. https://doi.org/10.1145/3197231.3197241

[48] Konstantin Rubinov and Luciano Baresi. 2018. What Are We Missing When Testing Our Android Apps? *Computer* 51, 4 (2018), 60–68.

[49] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding resume and restart errors in android applications. *ACM SIGPLAN Notices* 51, 10 (2016), 864–880. https://doi.org/10.1145/3022671.2984011

[50] Aman Sharma and Rupesh Nasre. 2019. QADroid: regression event selection for Android applications. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 66–77. https://doi.org/10.1145/3293882.3330550

[51] Sixth Tone. 2019. *E-Commerce App Loses 'Tens of Millions' From Coupon Glitches*. Retrieved 2020-5 from https://www.sixthtone.com/news/1003483/e-commerce-app-loses-tens-of-millions-from-coupon-glitches

[52] Donna Spencer and T Warfel. 2004. Card Sorting. *Boxes and arrows* 7 (2004).

[53] statcounter. 2022. *Mobile Operating System Market Share Worldwide*. Retrieved 2022-8 from https://gs.statcounter.com/os-market-share/mobile/worldwide

[54] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2022. Why My App Crashes? Understanding and Benchmarking Framework-Specific Exceptions of Android Apps. *IEEE Trans. Software Eng.* 48, 4 (2022), 1115–1137. https://doi.org/10.1109/TSE.2020.3013438

[55] 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 *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 245–256. https://doi.org/10.1145/3106237.3106298

[56] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 119–130. https://doi.org/10.1145/3468264.3468620

[57] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–31. https://doi.org/10.1145/3485533

[58] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 204–215. https://doi.org/10.1145/3460319.3464806

[59] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and Finding System Setting-Related Defects in Android Apps. *IEEE Transactions on Software Engineering* (2023). https://doi.org/10.1109/TSE.2023.3236449

[60] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705. https://doi.org/10.1007/s10664-013-9258-8

[61] ASCII Team. 2022. *ASCII Codes Table*. Retrieved 2022-10 from https://ascii.cl/

[62] Jenifer Tidwell. 2010. *Designing interfaces: Patterns for effective interaction design*. " O'Reilly Media, Inc.".

[63] uiautomator2 Team. 2021. *uiautomator2*. Retrieved 2022-10 from https://github.com/openatx/uiautomator2

[64] Mario Linares Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 399–410. https://doi.org/10.1109/ICSME.2017.27

[65] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 434–446. https://doi.org/10.1145/3540250.3549170

[66] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 738–748. https://doi.org/10.1145/3238147.3240465

[67] 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)*. 226–237. https://doi.org/10.1145/2970276.2970312

[68] Tyler Wendland, Jingyang Sun, Junayed Mahmud, S. M. Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A Dataset of Manually-Reproduced Bug Reports for Android apps. In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR0)*. 600–604. https://doi.org/10.1109/MSR52588.2021.00082

[69] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Seventh IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 183–192. https://doi.org/10.1109/ICST.2014.31

[70] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 128–139. https://doi.org/10.1109/ICSE.2019.00030

Received 2023-02-16; accepted 2023-05-03