Understanding and Finding System Setting-Related Defects in Android Apps

Jingling Sun  
East China Normal University  
China  
jingling.sun910@gmail.com

Ting Su  
East China Normal University  
China  
tsu@sei.ecnu.edu.cn

Junxin Li  
East China Normal University  
China  
leejuniorxin@gmail.com

Zhen Dong  
National University of Singapore  
Singapore  
zhen.dong@comp.nus.edu.sg

Geguang Pu*  
East China Normal University  
China  
ggpu@sei.ecnu.edu.cn

Tao Xie  
Peking University  
China  
taoxie@pku.edu.cn

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

ABSTRACT

Android, the most popular mobile system, offers a number of user-configurable system settings (e.g., network, location, and permission) for controlling devices and apps. Even popular, well-tested apps may fail to properly adapt their behaviors to diverse setting changes, thus frustrating their users. However, there exists no effort to systematically investigate such defects. To this end, we conduct the first empirical study to understand the characteristics of these setting-related defects (in short as "setting defects"), which reside in apps and are triggered by system setting changes. We devote substantial manual effort (over three person-months) to analyze 1,074 setting defects from 180 popular apps on GitHub. We investigate their impact, root causes, and consequences. We find that setting defects have a wide, diverse impact on apps' correctness, and the majority of these defects (~70.7%) cause non-crash (logic) failures, and thus could not be automatically detected by existing app testing techniques due to the lack of strong test oracles.

Motivated and guided by our study, we propose setting-wise metamorphic fuzzing, the first automated testing approach to effectively detect setting defects without explicit oracles. Our key insight is that an app’s behavior should, in most cases, remain consistent if a given setting is changed and later properly restored, or exhibit expected differences if not restored. We realize our approach in SetDroid, an automated, end-to-end GUI testing tool, for detecting both crash and non-crash setting defects. SetDroid has been evaluated on 26 popular, open-source apps and detected 42 unique, previously unknown setting defects in 24 apps. To date, 33 have been confirmed and 21 fixed. We also apply SetDroid on five highly popular industrial apps, namely WeChat, QQMail, TikTok, CapCut, and AlipayHK, all of which each have billions of monthly active users. SetDroid successfully detects 17 previously unknown setting defects in these apps' latest releases, and all defects have been confirmed and fixed by the app vendors. The majority of SetDroid-detected defects (49 out of 59) cause non-crash failures, which could not be detected by existing testing tools (as our evaluation confirms). These results demonstrate SetDroid's strong effectiveness and practicality.

CCS CONCEPTS
- Software and its engineering → Software testing and debugging.

KEYWORDS
Empirical study, Testing, Android, Setting

ACM Reference Format:

1 INTRODUCTION

Android supports the running of millions of apps nowadays. Specifically, a number of user-configurable system settings are offered by the (preinstalled) system app Settings on Android for controlling devices and apps. For example, users can change the system language, switch to another network connection, grant or revoke app permissions, or adjust the screen orientation. When these settings change, an app is expected to correctly adapt its behavior, and behave consistently and reliably.
However, achieving the preceding goal is challenging. Even popular, well-tested apps may be unexpectedly affected due to inadequate considerations of diverse setting changes. For example, WordPress [59], a popular website and blog management app (which has 50,000,000 installations on Google Play and 2,400 stars on GitHub), suffered from two defects triggered by switching to the airplane mode (a commonly-used setting during traveling). One defect was triggered when a user turned on the airplane mode when publishing a new blog post; WordPress was stuck at the post uploading status even after the user later turned off the airplane mode and connected to the network [28]. The other defect was triggered when a post draft was created under the airplane mode; WordPress constantly crashed at the next startup [24]. Both defects were labeled as critical but escaped from pre-release developer testing.

Moreover, these setting defects can be frustrating. For example, NextCloud [50] is a popular on-premise file-sharing app (which has 5,000,000 installations on Google Play and 2,200 stars on GitHub). A user reported that he could not use the auto-upload functionality for unknown reasons [11]. After extended discussion, the developers finally found that the auto-upload functionality failed because the power saving mode was turned on. The user complained that he preferred keeping the power saving on all day to save battery. To make sure that the auto-upload functionality would work, he added NextCloud into the whitelist of the power saving mode (which allows NextCloud to use battery without any restrictions), but the functionality still did not work.

Despite these setting defects’ real-world occurrences and impact, there exists no effort to systematically investigate these defects in Android apps. For example, prior work studies only very limited types of system settings (e.g., app permissions [32] and screen orientation [3]). On the other hand, state-of-the-art generic app testing techniques [17, 60] cannot effectively detect these setting defects for two major reasons. First, these techniques usually constrain the testing within the app under test and thus have no or little chance to detect these defects, which require interacting with the system app Settings. Second, these techniques are limited to detecting crash failures [6, 63] due to the lack of strong test oracles [62], while many setting defects are logical ones that lead to app freezing, functionality failures, or GUI display failures.

To fill this gap, we conduct the first systematic study to understand the characteristics of these setting defects. Specifically, we aim to investigate the following research questions:

- **RQ1 (impact):** Do settings defects have a wide impact on the correctness of apps in the wild?
- **RQ2 (root causes):** What are their major root causes?
- **RQ3 (consequences):** What are their common consequences? How do they manifest?

Specifically, Figure 1 shows the overview of this study. We first carefully inspect the Android documentation [39, 40], and systematically summarize the set of system setting categories and options (see Section 2.1). This summarization leads to nine main setting categories and over 50 setting options. Based on the keywords of these settings, we mine 1,074 bug reports of setting defects from the issue repositories of 180 popular Android apps on GitHub (see Section 2.2.2). Finally, we carefully study these defects by reviewing the bug reports and analyzing the root causes, fixes, and consequences (see Section 2.3) to answer RQ1–RQ3 in Section 3.

Our study reveals that setting defects have a wide, diverse impact on the correctness of apps. Specifically, out of the 180 apps, 171 apps (=95%) use at least one setting option in their code, and 162 apps (=90%) have been affected by setting defects. Further, we distill five major root causes. Specifically, **incorrect callback implementations** and **lack of setting checks** are the most common. We also note that only a few setting defects (≈2%) are caused by the mutual influence between two settings. On the other hand, setting defects lead to **diverse** consequences, including crashes, functionality failures, problematic GUI display, and disrespect of setting changes. Specifically, the majority of these defects (≈70.7%) cause **non-crash** failures, highlighting the necessity of new, effective testing techniques.

Guided by our study findings, we design and introduce **setting-wise metamorphic fuzzing**, the first automated testing technique to detect setting defects (causing crash and non-crash failures, respectively) for Android apps. We implement this technique as an end-to-end, automated GUI testing tool, SetDroid, and apply it on 26 popular, open-source Android apps. SetDroid has successfully discovered 42 unique, previously-unknown setting defects. So far, 33 have been confirmed and 21 fixed by the developers. We further apply SetDroid on five highly popular industrial apps that each have billions of monthly active users worldwide, i.e., WeChat [58] and QQMail [53], from Tencent, TikTok [57] and CapCut [43] from ByteDance, and AlipayHK [38] from Alibaba. In these apps’ latest releases, SetDroid successfully finds 17 setting defects, all of which have been confirmed and fixed by the app vendors. The majority of all these setting defects (49 out of 59) cause non-crash failures, which cannot be detected by existing automated testing tools (corroborated by our evaluation in Section 4.3). These results demonstrate SetDroid’s effectiveness and practicality.

In summary, this paper makes the following main contributions:

- **We conduct the first systematic study on setting defects to assess their impact, root causes, consequences, and manifestations.**
- **Informed by this study, we design and introduce setting-wise metamorphic fuzzing, the first automated GUI testing technique to effectively detect setting defects in Android apps.**
- **Our SetDroid tool implemented for our technique has revealed 42 setting defects in 26 open-source apps (33 confirmed, and 21...
fixed) and 17 defects in five industrial apps (all confirmed and fixed). The majority of these defects (49 out of 59) cause non-crash failures and could not be detected by existing testing tools.

2 EMPIRICAL STUDY METHODOLOGY

2.1 Summarizing Setting Categories

To systematically summarize the setting categories, we inspect the Android documentation [39, 40] and the mainstream Android systems (Android 7.1, 8.0, 9.0, and 10.0). We finally identify 9 major setting categories. Further, we summarize (1) the commonly used keywords to denote the settings in these categories; these keywords are used by the bug-report collection in Section 2.2.2; and (2) the specific Android SDK APIs (including classes, methods, or variables) used by or related to these setting categories; these APIs are used by the impact analysis in Section 2.3. Table 1 lists these 9 major setting categories. The column “Setting Categories” lists the category names of these settings as classified by Android, “Keywords” gives the commonly used keywords to denote the settings within these categories, and “Description” summarizes their main functionalities.

2.2 Collecting Bug Reports of Setting Defects

We use three steps to collect valid bug reports of setting defects.

2.2.1 Step 1: App Collection. We choose open-source Android apps on GitHub as our study subjects because we can view their source code, failure/defect descriptions, reproducing steps, fix patches, and discussions. Specifically, we collect the app subjects as follows.

- We use GitHub’s REST API [47] to crawl all the Android projects on GitHub. We focus on the apps that are released on Google Play and F-Droid, the two popular Android app markets. Because these apps can receive feedback from real users and thus are usually well maintained. We attain 1,728 Android projects.
- To focus on those projects that contain enough bug reports for our study, we keep only the projects with more than 200 closed issues/bug reports. We then attain 215 Android projects.
- We manually inspect each project and exclude the ones that are not real apps (e.g., some projects are in fact third-party Android libraries, and release simple demo apps on the markets). Finally, we attain 180 Android apps as our study subjects.

Figure 2 shows the characteristics of the 180 apps in terms of the numbers of stars and issues/bug reports on GitHub, installations on Google Play, and app categories. We can see that these apps are popular and diverse, serving as a solid basis for our analysis.

2.2.2 Step 2: Bug-report Collection. From the 180 apps, we attain 177,769 bug reports in total. To collect bug reports for our study, we use three sets of keywords to filter bug reports. When a bug report contains at least one keyword from each of these three keyword sets, we select the bug report to include in our study.

- **Setting keywords**: A bug report within our study scope should contain at least one of the setting keywords listed in Table 1. For each keyword, we consider the possible forms that users may use (e.g., capitalization, abbreviations, and tenses). For example, users may use “power saving” to represent “power save.”
- **Defect/failure keywords**: We focus on the bug reports that describe real app defects/failures rather than feature requests or documentation issues. Thus, we use the keywords of “crash”, “exception”, “bug”, and “issue” to filter bug reports.
- **Reproducing keywords**: To facilitate bug-report analysis, we focus on the bug reports that contain the reproducing steps. We use the keywords of “repro”, “STR”, and “record” to filter bug reports. These reproducing steps are important, helping us understand and confirm whether a bug report indeed reflects a setting defect.

Finally, we attain 11,656 bug reports within our study scope.

2.2.3 Step 3: Dataset Construction. To answer RQ1, RQ2, and RQ3, we manually inspect the 11,656 bug reports from the previous step, and keep only the valid bug reports by the following rules:

- We retain only the bug reports where the reporters or developers make clear statements that changing system settings is a necessary condition for triggering the failures.
- When we do not have clear clues from bug reports, we reproduce the failures to confirm whether they reflect setting defects. For example, we exclude the bug reports that just mention settings.

Finally, we successfully attain 1,074 valid bug reports as the dataset for our subsequent analysis. Among these bug reports, 482 are closed with explicitly-linked code fixing commits.

2.3 Analysis Methods for Research Questions

This section details the analysis methods used to answer the research questions. Note that, to avoid omissions and misclassifications in answering RQ1, RQ2, and RQ3, four co-authors participate in the process for data collection, classification, manual analysis, and cross-checking.

2.3.1 Analysis Method of RQ1. To answer RQ1, we focus on the 180 apps collected from GitHub and investigate (1) the usage of settings in the apps, i.e., which apps use which setting categories; and (2) the impact of setting defects against the apps, i.e., which apps are ever affected by which setting defects.

To investigate the usage of settings, we use static analysis to analyze whether an app uses specific APIs (classes, methods, or variables) of each setting category (summarized in Section 2.1) in their code. We observe that this method is feasible and reliable because using these specific APIs is the only way for an app to access settings. For example, apps use the class `ConnectivityManager` to query the network connectivity, and get notified when the network connectivity changes. Thus, we use these classes, methods, or variables to determine which setting category is used by an app. We give the complete list of these APIs of each setting category used in this study on the web page of our supplementary materials [33].
Table 1: List of 9 major setting categories summarized by our study including their levels, keywords, and brief descriptions.

<table>
<thead>
<tr>
<th>Setting Categories</th>
<th>Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network and connect</td>
<td>Bluetooth, WLAN, NFC, internet, network, hot-spot, mobile, wifi, airplane</td>
<td>Manage the device’s network mode (WiFi, mobile data or airplane mode), and the connection with other devices (such as Bluetooth).</td>
</tr>
<tr>
<td>Location and security</td>
<td>location, device only, phone only, GPS, high accuracy, screen lock, fingerprint</td>
<td>Manage the device’s security settings (e.g., how to unlock the screen), location setting (turning on/off device location) and three location modes: high accuracy (using the network and GPS), battery saving (using the network), and device only (using GPS).</td>
</tr>
<tr>
<td>Sound</td>
<td>vibrate, ringtone, do not disturb, silent</td>
<td>Manage the device’s sound-related options (e.g., “do not disturb” mode can completely mute the device).</td>
</tr>
<tr>
<td>Battery</td>
<td>power save, battery</td>
<td>Manage the power saving mode and the list of apps that are not restricted by the power saving mode (the power saving whitelist).</td>
</tr>
<tr>
<td>Display</td>
<td>orientation, vertical, horizontal, split screen, Multi-window, screen resolution, brightness, landscape, portrait, rotate</td>
<td>Manage the device’s display settings (e.g., screen brightness and font size) and screen orientation settings (e.g., whether to allow the device to rotate the screen).</td>
</tr>
<tr>
<td>Apps and notifications</td>
<td>permission, disable, notification</td>
<td>Manage the runtime permissions of apps and whether they can push notifications to users.</td>
</tr>
<tr>
<td>Developer</td>
<td>developer option, keep activity</td>
<td>A number of advanced settings to simulate specific running environment (e.g., enable “Force RTL (right to left) layout direction”).</td>
</tr>
<tr>
<td>Accessibility</td>
<td>accessibility, talkback, text-to-speech, color correction, color inversion, high contrast text</td>
<td>Customize the device to be more accessible, e.g., adjusting the contrast of UI interface and opening the screen reader.</td>
</tr>
<tr>
<td>Other Settings</td>
<td>setting, preference, date, time, time zone, hour format, date/time, reading mode, car mode, one-handed mode, dark mode, game mode, night mode, theme, language</td>
<td>Users can change the languages, the way that they input, the system time, the time zone and hour format (24-hour or 12-hour format), and the themes.</td>
</tr>
</tbody>
</table>

Table 2: Statistics of the impact of settings on the apps.

<table>
<thead>
<tr>
<th>Setting Categories</th>
<th>#Apps using settings</th>
<th>#Apps were affected</th>
<th>#Setting Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network and connect</td>
<td>96</td>
<td>68</td>
<td>326</td>
</tr>
<tr>
<td>Location and security</td>
<td>47</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Sound</td>
<td>67</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>Battery</td>
<td>57</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Display</td>
<td>109</td>
<td>78</td>
<td>226</td>
</tr>
<tr>
<td>Apps and notification</td>
<td>134</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>Others</td>
<td>-</td>
<td>-</td>
<td>319</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>171</strong></td>
<td><strong>162</strong></td>
<td><strong>1,074</strong></td>
</tr>
</tbody>
</table>

2.3.2 Analysis Method of RQ2. To answer RQ2, we focus on analyzing the setting defects in the 482 fixed bug reports out of the 1,074 valid bug reports. We study the fixed bug reports, including developer comments and code fixes, to understand the setting defects’ root causes. If necessary, we also refer to Android documentation or Stack Overflow to find more clues.

Specifically, two co-authors first work on a common set of bug reports to identify the root causes based on (1) the causes behind these setting defects and (2) the defect fixing strategies. Then, the two co-authors discuss together with the other co-authors to reach the consensus on the initial categories. After that, the four co-authors work separately on the remaining bug reports to classify the root causes. They discuss and cross-check together when the categories need to be updated (e.g., add, merge, or modify categories).

2.3.3 Analysis Method of RQ3. To answer RQ3, we focus on all the 1,074 valid bug reports. We study these bug reports to determine the consequences. When necessary, we also reproduce the failures to observe the consequences.

3 STUDY RESULTS AND ANALYSIS

3.1 RQ1: Impact of Settings Defects

To understand the impact of setting defects, we investigate two aspects: (1) the usage of settings in the apps; (2) the impact of setting defects against the apps. Table 2 (column “#Apps using settings”) lists the number of apps that use APIs related to each setting category. The result is based on the list of setting-related APIs summarized in Section 2.1. Note that the numbers in Table 2 may overlap because one app may use or be affected by multiple settings. Row “#Total” gives the unique number of apps or setting defects. In addition, we have not counted the usage of some settings (e.g., “Developer”, “Accessibility”, denoted by “-” in “Others” in Table 2) because they do not export explicit APIs. Thus, the current result is in fact a lower bound of setting usage by apps. In Table 2, we can see that 95% (171/180) of the apps use at least one setting-related API. Among all the setting categories, “Apps and notification” is the most commonly used one because most non-trivial apps use dynamic permissions and notifications. The setting category “Network and connect” is also commonly used.

Table 2 (column “#Apps were affected”) counts which apps are ever affected by setting defects according to the 1,074 bug reports in our dataset. We find that most apps have ever been affected by at least one setting defect. Specifically, 162 apps have setting defects, which account for 90% (162/180) of the 180 apps under study. The three categories “Display”, “Network and connect”, and “Apps and notifications” have the widest impact on the app’s correctness.

207
change the system language, specific Activity lifecycle callbacks (e.g., `onCreate()`) are called. If these callbacks are not correctly implemented, setting defects may occur. Thus, these defects are usually fixed in specific callback methods.

For example, in AnkiDroid [41]’s Issue #4951, when a user grants the storage permission from the permission request dialog, the original 3-dot menu icon disappears from the top-right corner of the screen. The reason is that the developers do not properly handle the app logic in the callback. Figure 3 shows the patch. When the user responds to the permission request, the system invokes the callback `onRequestPermissionsResult()` (Line 1). After the user grants the storage permission, the original menu should be redrawn because its content is changed. However, the developers forget to call `invalidateOptionsMenu()` to redraw the menu (Line 5).

### 3.2.2 Lack of Setting Checks

Many apps could be affected when specific settings (e.g., network) change. If developers fail to properly check the status of these settings or do not monitor the status while using related setting APIs, some serious failures may occur. These defects are usually fixed by adding conditional checks.

For example, in NextCloud’s Issue #2889, the user complains that some app functionalities are affected even if she whitelists the app from the power saving. As shown in Figure 4, the developers check only whether the device is in the power saving mode by `isPowerSaveMode()` (Line 3), but do not check whether the app is in the whitelist of the power saving mode by `isIgnoringBatteryOptimizations()`. In the end, the developers fix the defect by adding this check (Lines 5-6).

#### 3.2.3 Fail to Adapt User Interfaces

Some settings, e.g., multi-window display, font size, languages, and dark mode, affect the user interfaces (UIs) of apps. If an app fails to properly adapt its UIs when these settings change, some display defects may exist. We observe that such setting defects are usually fixed by modifying the resource files (e.g., XML layouts) rather than the app code.

For example, because the UI layouts are not properly designed, Status [55]’s Issue #914 leads to the disappearance of some UI elements when the app adapts itself to the multi-window display mode. In Frost [46]’s Issue #1659, when users change the system language from German to Russian, the texts overlap or cannot be displayed completely within the screen, because the translation from German to Russian leads to much longer texts.

### 3.2.4 Mutual Influence Between Settings

Some settings have explicit or implicit mutual influence, which many app developers are unaware of. This factor may lead to some unexpected setting defects. The fixes of such defects usually involve multiple settings.

One typical example of explicit mutual influence is that the positioning in Android can be affected by the settings of both network and location. Because Android supports positioning via either GPS or network or both. In Commons [45]’s Issue #1735, the app

### Table 3: Major root causes of setting defects

<table>
<thead>
<tr>
<th>Root Causes</th>
<th>#Bug Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect callback implementations</td>
<td>164</td>
</tr>
<tr>
<td>Lack of setting checks</td>
<td>143</td>
</tr>
<tr>
<td>Fail to adapt user interfaces</td>
<td>103</td>
</tr>
<tr>
<td>Lack of considering Android versions</td>
<td>27</td>
</tr>
<tr>
<td>Mutual influence between settings</td>
<td>12</td>
</tr>
<tr>
<td>Other minor reasons</td>
<td>33</td>
</tr>
</tbody>
</table>

### 3.3 RQ3: Consequences of Setting Defects

This section summarizes the four major consequences of the defects reflected by the 1,074 bug reports in our dataset. RQ3 aims to understand whether these defects have any common failure
manifestations. We detail the four major consequences w.r.t. their numbers of defects from the most to least. The remaining 59 defects’ consequences are very specific (e.g., sluggish GUIs, delayed updates of app data on GUIs), so we do not discuss them in detail.

### 3.3.1 Crash
315 of the 1,074 setting defects lead to app crash. In most cases, users can recover the app by restoring the setting changes and restarting the app. But in some cases, users cannot restore the settings changes, and the app is totally broken. For example, in OpenFoodFacts [51]’s Issue #1118, when users switch to the Hindi language, the app preference page anymore cannot be opened and just crashes. The users have to reinstall the app.

### 3.3.2 Disrespect of Setting Changes
285 setting defects disrespect the changes of settings, i.e., setting changes do not take effect. The main reason is that developers fail to consider some settings, and thus the app does not adapt itself to these setting changes. For example, in Signal [54]’s Issue #6411, even if users turn on the “Do not disturb” mode, Signal is still making the sound from time to time when notifications come in, annoying the users. Other failure manifestations include untranslated texts or incomplete translations when the system language is changed.

### 3.3.3 Problematic UI Display
218 setting defects lead to problematic UI display. Some settings, e.g., languages and themes, may affect UI display if the corresponding resource files are not correctly implemented. For example, in the email client app K-9 [48], users can see the quoted texts from the last reply when writing an email. But when the app’s theme is changed to the dark mode, the quoted texts from the last reply become invisible. Because the developers forget to adjust the color of the quoted texts (which are in black) according to the current theme.

### 3.3.4 Functionality Failure
197 setting defects lead to functionality failure, i.e., the original app functionality cannot work as expected when some setting changes happen. In most cases, the affected apps do not alert users that the functionality fails due to the setting changes; in some cases, the apps may give a wrong alert and mislead the users. For example, in syncthing [56]’s Issue #727, the background synchronization functionality does not work for unknown reasons. After a long discussion, the developers find that the functionality fails because the power saving mode is enabled. In this case, syncthing does not alert the users that the power saving mode is affecting the synchronization functionality, and thus confuses the users. Other failure manifestations include app stuck, black screen, infinite loading, and unable to refresh.

### Answer to RQ3
Our study reveals that setting defects lead to diverse consequences. The majority (759/1,074 ≈ 70.7%) of them cause non-crash failures and manifest only as GUI defects, which are hard to be automatically detected by existing testing tools.

### 4 DETECTING SETTING DEFECTS

#### 4.1 Setting-wise Metamorphic Fuzzing

**4.1.1 High-level Idea**
Our key insight is that, in most cases, the app behaviors should keep consistent if a given setting is changed and later properly restored, or show expected differences if not restored. Otherwise, a likely setting defect is found. For example, an app’s functionalities should not be affected if the network is closed but immediately opened, or an app should show the texts in a different language if the default language is changed. Thus, based on the preceding observation, we are inspired to leverage the idea of metamorphic testing [5] to tackle the oracle problem.

**4.1.2 Approach.** Our approach, setting-wise metamorphic fuzzing, randomly injects a pair of events \((e_c, e_u)\) into a given seed GUI test \(E\) to obtain a mutant test \(E'\), where \(e_c\) changes a given setting, while \(e_u\) properly restores the setting or does nothing. By comparing the GUI consistency between the seed test \(E\) and the mutant test \(E'\), we can tell whether the app behaviors have been affected.

Formally, let \(E\) be a seed GUI test that is a sequence of events, i.e., \(E = [e_1, e_2, \ldots, e_n]\), where \(e_i\) is a user event (e.g., click, edit, swipe, screen rotation). \(E\) can be executed on an app \(P\) to obtain a sequence of GUI layouts (pages) \(L = [L_1, L_2, \ldots, L_{n+1}]\), where \(L_i\) is a GUI layout (which consists of a number of GUI widgets).

Specifically, if we view the execution of \(e_i\) as a function, then \(L_{i+1} = e_i(L_i), i \geq 1\). By injecting a pair of new events \((e_c, e_u)\) into \(E\), we can obtain a mutant GUI test \(E' = [e'_1, e'_2, \ldots, e'_n, e'_{u+1}, \ldots, e'_{n+1}]\) that can be executed on \(P\) to obtain a sequence of GUI layouts (pages) \(L' = [L'_1, L'_2, \ldots, L'_n, L'_{u+1}, \ldots, L'_{n+1}]\). We compare the GUI consistency between the GUI layouts of \(E\) (i.e., \(L\)) and those of \(E'\) (i.e., \(L'\)), respectively, to find defects. In practice, we check the GUI consistency by comparing the differences of executable GUI widgets between \(L\) and \(L'\). Let \(e.w\) be the GUI widget \(w\) that \(e\) targets.

**Oracle checking rule I.** Rule I is coupled with the following two strategies that inject \((e_c, e_u)\) into \(E\) to obtain \(E'\). Conceptually, in most cases, the app behaviors should keep consistent.

- **Immediate setting mutation.** We inject \(e_c\) followed immediately by \(e_u\). For example, \(e_c\) turns on the power saving mode, and \(e_u\) immediately adds the app into the whitelist of power saving.
- **Lazy setting mutation.** We inject \(e_c\) first and inject \(e_u\) only when it is necessary (e.g., the app prompts an alert dialog or a request message). For example, \(e_c\) revokes app permission, and \(e_u\) grants the permission only when the app requests that permission. Note that our study justifies the rationale of the lazy mutation strategy because prompting proper alerts to users is demanded by Android design guidelines to improve user experience [42].

\[
\text{Seed test trace: } \ell_1 \rightarrow e_c \rightarrow \ell_2 \rightarrow \cdots \rightarrow \ell_i \rightarrow e_u \rightarrow \ell_{u+1} \\
\text{Mutant test trace (w/setting defect): } \ell'_1 \rightarrow e'_c \rightarrow \ell'_{u+1} \rightarrow \cdots \\
\text{Mutant test trace (w/setting defect): } \ell'_1 \rightarrow e'_c \rightarrow \ell'_{u+1} \rightarrow \cdots \\
\text{(e_c, e_u)}
\]

The preceding figure illustrates Rule I: under these two injection strategies, if there exists one GUI event \(e_c \in E'\) and its target widget \(e_c.w\) cannot be located on the corresponding layout \(L'_i\) (\(L'_i\) corresponds to \(L_i\)), then a likely setting defect is found. Because it likely indicates that the app’s behaviors are affected. Formally, \(\exists e_c, e_u, w \in L_i \land e_c.w \notin L'_{i+1}\).
4.2 Design and Implementation of SetDroid

We implement our approach as an automated GUI testing tool named SetDroid. Figure 6 depicts the workflow. It has four main modules: (a) test executor, (b) setting change injector, (c) oracle checker, and (d) bug report reducer. We detail the four modules as follows.

Test executor. The test executor runs the same app under test (AUT) on two identical devices A and B in parallel. During testing, the executor generates a seed test on-the-fly on device A and replays the same seed test injected with setting changes (i.e., the mutant test) on device B at the same time. The executor works in a loop: (1) get the current GUI layout of the AUT on device A, (2) randomly choose an executable widget from the layout and generate an event, (3) execute the event on both devices A and B. This on-the-fly strategy offers the flexibility for injecting setting changes at runtime. We use random seed tests because they are diverse, practical, and scalable to obtain. In practice, we use the UI Automator test framework [61] to execute events and obtain GUI layouts.

Setting change injector. Informed by our study, we adopt two key insights in designing this module. First, we find that many setting defects (211/486≈43.4%) in our study are triggered by changing settings at runtime rather than before starting the apps. Guided by the first insight, SetDroid randomly injects the pair of events \( (e_c, e_u) \) at any position of a seed test rather than only at its beginning. Moreover, if one pair of \( (e_c, e_u) \) is injected and no setting defect is found, the next same pair of \( (e_c, e_u) \) is injected again later. This process continues until the seed test ends. Second, we find that only a few setting defects (10/486≈2%) are caused by explicitly changing two settings (i.e., two settings are changed to non-default values at the same time), and no defects are caused by changing more than two settings. Guided by the second insight, the setting change injector randomly injects one single pair of events \( (e_c, e_u) \) at one time, which does not interleave with others. Only the screen orientation (which is viewed as a normal user event) may be interleaved with other setting changes. Specifically, the injector decides whether to inject \( e_c \) after each GUI event in the seed test by coin-flipping and later injects \( e_u \) according to the mutation strategy. Table 4 lists the supported pairs of events for setting changes and the corresponding injection strategies (defined in Section 4.1). These events are designed based on our empirical study, which can manifest the majority of setting defects and cover the other possible forms of related setting changes. Before testing, the two devices A and B are initialized with the same default setting environment: airplane mode off, Wi-Fi on, mobile data on, location (high accuracy) on, battery saving mode off, multi-window off, screen orientation in the landscape, DND mode off, language is English, and 12-hour format.

Oracle checker. After each event is generated by the test executor, the oracle checker checks whether the layout of device B is consistent with that of device A (i.e., Rule I), or shows expected differences w.r.t. that of device A (i.e., Rule II), while also monitoring app crashes. If a defect is found, the checker generates a bug report that includes the executed events, GUI layouts, and screenshots.

Bug report reducer. The bug report reducer removes any bug report that is duplicated or cannot be faithfully reproduced. Specifically, it replays the recorded GUI tests that trigger setting defects for multiple runs to decide reproducibility. It uses the GUI inconsistencies between the two layouts as the hash key to remove any duplicated bug report. This step does not incur any false negatives.

4.3 Evaluation of SetDroid

We evaluate SetDroid and the usefulness of the insights gained from the study by answering RQ4 and RQ5:

- **RQ4**: Can SetDroid detect unknown setting defects in real-world Android apps (both open-source and industrial apps)?

- **RQ5**: Do the insights gained from the study help SetDroid detect some setting defects that cannot be detected by existing tools?

4.3.1 Evaluation Setup of RQ4. For our evaluation subjects, we consider the 30 apps from prior work [22], because most of these apps are selected from popular open-source apps on GitHub [30]. In the end, our evaluation subjects include 26 apps because the other 4 apps are obsolete. We run SetDroid on a 64-bit Ubuntu 18.04 machine (64 cores, AMD 2990WX CPU, and 64GB RAM), Android emulators (Android 8.0, Pixel XL). SetDroid applies the 13 pairs of setting changing events (in Table 4) separately on each app. For each pair, SetDroid randomly generates 20 seed tests (each seed contains 100 events) for fuzzing, which takes about 1 hour. Thus, the whole evaluation for 26 apps takes 1’13’’26=338 CPU hours (nearly 340 hours).
<table>
<thead>
<tr>
<th>App ID</th>
<th>App name</th>
<th>Installations</th>
<th>#Stars</th>
<th>Issue ID</th>
<th>Issue state</th>
<th>Cause setting</th>
<th>Consequence</th>
<th>#FP</th>
<th>#TP</th>
<th>#FP FP</th>
<th>#TP TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A1PhotoManager</td>
<td>-</td>
<td>162</td>
<td>#175</td>
<td>Confirmed</td>
<td>Permission</td>
<td>Crash</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>AZDP Volume</td>
<td>100K-500K</td>
<td>71</td>
<td>#295</td>
<td>Fixed</td>
<td>Display</td>
<td>Crash</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Always On</td>
<td>10M-50M</td>
<td>121</td>
<td>#2476</td>
<td>Confirmed</td>
<td>Permission</td>
<td>Data lost</td>
<td>2</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>AntikDroid</td>
<td>5M-10M</td>
<td>3.2K</td>
<td>#5407</td>
<td>Fixed</td>
<td>Permission</td>
<td>Stuck</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>AntennaPod</td>
<td>500K-1M</td>
<td>3.3K</td>
<td>#4227</td>
<td>Fixed</td>
<td>Network</td>
<td>Crash</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Commons</td>
<td>50K-100K</td>
<td>649</td>
<td>#3906</td>
<td>Discussion</td>
<td>Location</td>
<td>Data lost</td>
<td>0</td>
<td>9</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>ConnectBot</td>
<td>1M-5M</td>
<td>1.6K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>FillUp</td>
<td>100K-500K</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Forecastie</td>
<td>10K-50K</td>
<td>609</td>
<td>#505</td>
<td>Fixed</td>
<td>Permission</td>
<td>Lack of prompt</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Good Weather</td>
<td>5K-10K</td>
<td>196</td>
<td>#62</td>
<td>Waiting</td>
<td>Network</td>
<td>Data lost</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>51</td>
</tr>
<tr>
<td>11</td>
<td>Notepad</td>
<td>100K-500K</td>
<td>156</td>
<td>#55</td>
<td>Waiting</td>
<td>Location</td>
<td>Crash</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Omni Notes</td>
<td>100K-500K</td>
<td>2.2K</td>
<td>#776</td>
<td>Fixed</td>
<td>Permission</td>
<td>Lack of prompt</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Opensaduku</td>
<td>100K-50K</td>
<td>209</td>
<td>#93</td>
<td>Confirmed</td>
<td>Language</td>
<td>Incomplete translation(7)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>RedReader</td>
<td>50K-100K</td>
<td>1.1K</td>
<td>#783</td>
<td>Discussion</td>
<td>Language</td>
<td>Incomplete translation(23)</td>
<td>0</td>
<td>4</td>
<td>291</td>
<td>41</td>
</tr>
<tr>
<td>15</td>
<td>Suntimes</td>
<td>50K-100K</td>
<td>6.4K</td>
<td>#459</td>
<td>Confirmed</td>
<td>Permission</td>
<td>Crash</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>Vanilla Music</td>
<td>500K-1M</td>
<td>777</td>
<td>#1048</td>
<td>Waiting</td>
<td>Display</td>
<td>Crash</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>Wikipedia</td>
<td>50M-100M</td>
<td>1.3K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>OpenFileShare</td>
<td>1K-5K</td>
<td>58</td>
<td>#55</td>
<td>Confirmed</td>
<td>Display</td>
<td>Functionality failure</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>RadioBeacon</td>
<td>-</td>
<td>130</td>
<td>#2420</td>
<td>Fixed</td>
<td>Location</td>
<td>Incomplete translation(23)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>RunnerUp</td>
<td>10K-50K</td>
<td>511</td>
<td>#1965</td>
<td>Fixed</td>
<td>Display &amp; Permission</td>
<td>Black screen</td>
<td>4</td>
<td>21</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>RunnerUp</td>
<td>10K-50K</td>
<td>511</td>
<td>#1965</td>
<td>Fixed</td>
<td>Display &amp; Permission</td>
<td>Black screen</td>
<td>4</td>
<td>21</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>Amane</td>
<td>1M-5M</td>
<td>3K</td>
<td>#1925</td>
<td>Fixed</td>
<td>Permission</td>
<td>Crash</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>Amane</td>
<td>1M-5M</td>
<td>3.6K</td>
<td>#620</td>
<td>Fixed</td>
<td>Display</td>
<td>Data lost</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>Materialistic</td>
<td>100K-500K</td>
<td>2.1K</td>
<td>#1429</td>
<td>Waiting</td>
<td>Network</td>
<td>Crash</td>
<td>1</td>
<td>8</td>
<td>144</td>
<td>1</td>
</tr>
</tbody>
</table>

14 CPU days. For each bug report, SetDroid provides the failure-triggersing event trace and the screenshots. With this information, we manually inspect all bug reports and count the true positives (TP for short) and false positives (FP for short). We validate each bug report in the issue repositories. For each bug report, we provide the developers with the failure-reproducing steps and videos to ease failure diagnosis. The bug report is not marked as a duplicate by the developers, we regard it as a unique defect.

We also evaluate SetDroid on five industrial apps from Tecent, ByteDance and Alibaba, i.e., WeChat [38], QQMail [33], TikTok [57], CapCut [43], and AlipayHK [38], all of which each have billions of monthly active users. We allocate 2-day testing time for each app and run on two real devices (Galaxy A6s, Android 8.1.0). Then, we inspect any found defects and report them to the developers.

### Evaluation Setup of RQ5

Existing automated testing tools for Android can be divided into two categories. The first category includes generic testing tools [4, 7, 12, 23, 25, 27, 29, 37, 49]. These tools focus on the app under test and do not interact with the system app Settings to change settings. The second category includes tools for detecting specific failures [15, 22, 31, 32]. Specifically, Prefest [22] and PATDroid [32] are relevant to SetDroid. Prefest does app preference-wise testing but also considers some system settings (i.e., WiFi, Bluetooth, mobile data, GPS locating, and network locating), while PATDroid considers permissions. Note that in principle all these existing tools cannot detect non-crash failures that SetDroid targets. But we still do the comparison. Specifically, we build two baselines for comparison:

- **Baseline A** (random testing): This baseline mimics one typical generic testing tool, Monkey [49], which randomly explores the app under test without explicitly changing settings. This baseline follows the same testing strategy of Monkey but generates tests based on widgets (and thus the tests are easier to reproduce and much more understandable). In practice, Baseline A just runs Module a in Figure 6.

- **Baseline B** (random testing+setting changes): This baseline mimics the testing strategies of Prefest and PATDroid, which change settings before starting an app and then randomly explore the app. Baseline B considers all the setting changes in Table 4, including all the settings in Prefest and PATDroid. In practice, Baseline B just runs Modules a and b in Figure 6.

We also run Prefest and PATDroid for direct comparison.
We allocate 13 hours (the same time for SetDroid) for the two baselines, Prefest, and PATDroid to test each of the 26 open-source apps on one emulator, and check the generated bug reports to confirm whether they could find setting defects.

4.3.3 Results for RQ4. Effectiveness of SetDroid. Table 5 shows the evaluation results of SetDroid. Columns 2-4 give the app name, the number of installations on Google Play (* indicates that the app is not released on Google Play), and the number of stars on GitHub; Columns 5-8 give the issue ID, issue state (fixed, confirmed, under discussion with developers, or waiting for the reply), cause setting, and consequence. Out of the 26 apps, SetDroid finds 42 unique and previously-unknown setting defects from 24 apps. So far, 33 have been confirmed and 21 have been fixed. The result demonstrates SetDroid’s effectiveness. Further, we receive positive feedback from developers. For example, one developer of Forecastie comments that “Yep, good spot. Cheers for posting this bug”; one developer of Omni Notes responds “Thanks for pointing my attention to that”; “Well spotted. Cheers for the bug report.”. These comments show that SetDroid can find setting defects cared by developers.

Usability of SetDroid. During testing, SetDroid reports 811 defects. Among them, 149 defects are reported by oracle checking rule I, 124 of which are TPs (124/149≈83.2%); the remaining 662 defects are reported by oracle checking rule II, 127 of which are TPs (127/662≈19.2%). In Table 5, Columns 9-12 give the detailed numbers of true positives (TPs) and false positives (FPs) of oracle checking rules I and II, respectively, for each app. We analyze the FPs of these two rules and identify some major reasons.

- **FP I** of oracle checking rule I. Rule I in fact has very low false-positive rate (16.8%). We find that all these FPs are caused by specific app features triggered by setting changes. For example, when the screen orientation setting is changed, the Always on app pops up animation on top of the screen, leading to some GUI inconsistencies between the two devices.

- **FP II** of oracle checking rule II. From Table 5, we can see materialistic and RedReader incur the majority of false positives (435/477≈91.2%) of Rule II. These two apps are news readers in English. When SetDroid changes the default language, the texts of these news readers do not get translated. But SetDroid assumes that these behaviors violate Rule II, i.e., disrespect of setting changes. SetDroid reports one defect when a news article is detected, thus finally leading to a large number of false positives. For other apps, FPs are caused by some reserved keywords that do not get translated when the language is changed.

Although the false-positive rate of Rule II is high, the efforts of checking these cases are still affordable. In practice, SetDroid highlights the untranslated texts on the screenshots, and thus tool users can quickly locate and check the GUI inconsistencies. For example, SetDroid reports 332 defects in RedReader, and we are able to identify all 41 true positives within 20 minutes. On the other hand, we believe that this problem of high false-positive rate can be easily resolved by specifying that some GUI elements should be ignored during oracle checking.

**Diversity of defects found by SetDroid.** From Table 5, we can see that the setting defects found by SetDroid are diverse: the apps are affected by different settings with different consequences.

<table>
<thead>
<tr>
<th>ID</th>
<th>App</th>
<th>Setting</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>QQMail</td>
<td>Permission</td>
<td>Functional failure</td>
</tr>
<tr>
<td>2</td>
<td>QQMail</td>
<td>Permission</td>
<td>Crash</td>
</tr>
<tr>
<td>3</td>
<td>WeChat</td>
<td>Permission</td>
<td>Functional failure</td>
</tr>
<tr>
<td>4</td>
<td>WeChat</td>
<td>Permission</td>
<td>Functional failure</td>
</tr>
<tr>
<td>5</td>
<td>WeChat</td>
<td>Language</td>
<td>Problematic UI display</td>
</tr>
<tr>
<td>6</td>
<td>WeChat</td>
<td>Language</td>
<td>Incomplete translation</td>
</tr>
<tr>
<td>7</td>
<td>WeChat</td>
<td>Network</td>
<td>Stuck</td>
</tr>
<tr>
<td>8</td>
<td>WeChat</td>
<td>Network</td>
<td>Functional failure</td>
</tr>
<tr>
<td>9</td>
<td>CapCut</td>
<td>Network</td>
<td>Infinite loading</td>
</tr>
<tr>
<td>10</td>
<td>CapCut</td>
<td>Permission</td>
<td>Functional failure</td>
</tr>
<tr>
<td>11</td>
<td>CapCut</td>
<td>Display&amp;Permission</td>
<td>Problematic UI display</td>
</tr>
<tr>
<td>12</td>
<td>CapCut</td>
<td>Network</td>
<td>Functional failure</td>
</tr>
<tr>
<td>13</td>
<td>TikTok</td>
<td>Network</td>
<td>Functional failure</td>
</tr>
<tr>
<td>14</td>
<td>TikTok</td>
<td>Permission</td>
<td>Functional failure</td>
</tr>
<tr>
<td>15</td>
<td>TikTok</td>
<td>Location</td>
<td>Functional failure</td>
</tr>
<tr>
<td>16</td>
<td>AlipayHK</td>
<td>Language</td>
<td>Functional failure</td>
</tr>
<tr>
<td>17</td>
<td>AlipayHK</td>
<td>Location</td>
<td>Functional failure</td>
</tr>
</tbody>
</table>

In terms of root causes, we inspect these 21 fixed defects and find that most of them are due to the lack of setting checks. Some defects are due to incorrect callback implementations (e.g., AnkiDroid has one defect that fails to properly handle permission callbacks), while some defects are due to mutual influence between settings (e.g., Suntimes has one defect that fails to properly handle network connection and location positioning). On the other hand, most of the language defects are due to incomplete translation.

These setting defects also lead to different consequences. For example, based on Rule I, SetDroid detects one defect in Omni Notes. When the device-only mode is turned on, the app cannot insert the current location into the notes and prompts the user with a wrong, confusing message “location not found”. Based on Rule II, SetDroid detects a defect in Always On. When we change the default system language to another language, Always On indeed adjusts to the new language. But when Always On is closed and reopened, the language setting gets lost and Always On returns back to the default language.

**Practicality of SetDroid on industrial apps.** Our tool SetDroid detects 17 unique setting defects, all of which have been confirmed and fixed by Tencent, ByteDance, and Alibaba. Table 6 shows the details of these defects. According to our observation, these defects affect different modules and lead to different consequences. Some defects are severe and quickly fixed by the vendors.

**Answer to RQ4:** SetDroid can effectively detect setting defects in popular, real-world Android apps. These defects are diverse in terms of root causes and consequences, and are of developers’ concern. SetDroid also shows reasonable usability. Oracle checking rule I incurs very few false positives.

4.3.4 Results for RQ5. Table 7 shows the comparison results. We can see that SetDroid can detect more crash and non-crash setting defects than the other approaches. Baseline A does not detect any defect because it does not explicitly change settings like existing automated app testing tools, while Baseline B detects only 3 crashes (which are also detected by SetDroid) because it only changes settings before running tests. Because Prefest and PATDroid cover only limited types of settings, we compare the number of defects detected by Prefest/PATDroid and a restricted SetDroid (focusing on only those types of settings covered by Prefest/PATDroid), respectively (shown in the last eight columns of Table 7). Prefest does not detect any setting defect while PATDroid detects two crashes related to permissions. Note that the crashes detected by
Table 7: Comparison with existing tools and the baselines. C and NC represent crash and non-crash consequences, respectively.

<table>
<thead>
<tr>
<th>Setting</th>
<th>All settings</th>
<th>Baseline A</th>
<th>Baseline B</th>
<th>SETDROID</th>
<th>Bluetooth, network and location</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consequence</td>
<td>#Defects</td>
<td>C</td>
<td>NC</td>
<td>C</td>
<td>NC</td>
<td>C</td>
</tr>
<tr>
<td>Crash</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Non-crash</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

PATDROID and SETDROID do not overlap, likely caused by the randomness in test generation.

In summary, (1) SETDROID detects 33 non-crash setting defects, none of which can be detected by other approaches under comparison. (2) SETDROID is designed to change settings at random events, indeed exposing more (crash) setting defects, compared to Baseline B. (3) PREVEST and PATDROID focus on the combinations of setting changes, but do not detect any defects caused by multiple settings in our subjects, conforming to our findings that most of the setting defects can be manifested by one single setting. These results indicate the superiority of SETDROID over existing tools and usefulness of our study insights in designing SETDROID.

**Answer to RQ5:** Inspired by the insights of our study, SETDROID is designed to be able to detect non-crash setting defects that cannot be detected by existing automated testing tools. Moreover, by changing and restoring settings randomly, SETDROID can detect more crashes caused by setting changes.

5 THREATS TO VALIDITY

A main threat to validity is likely insufficient representativeness of app subjects used in our study. To alleviate this threat, for our systematic study, we collect 180 apps from 1,728 Android apps on GitHub. As shown in Section 2.2.2, these 180 apps are popular and cover diverse app categories. For the evaluation of SETDROID, besides highly popular industrial apps, we use all the non-obsolete app subjects from recent prior work [22].

Another main threat is likely incompleteness of setting keywords, causing incomprehensiveness of the setting defects collected by us. To alleviate this threat, we study the official Android documentation and collect as many keywords as possible for each setting, and we also consider different possible forms that users may use in bug reports. The final main threat is likely incorrectness of manual inspection. Our manual analysis may introduce errors. To alleviate this threat, the four co-authors cross-check each other’s analysis results to ensure correctness.

6 RELATED WORK

*Configuration testing for traditional software.* Prior work investigates misconfiguration defects for traditional software. Yin et al. [65] conduct a study on a commercial storage system (COMP-A) and four widely used open-source systems (CentOS, MySQL, Apache, and OpenLDAP) to study the main reasons of configuration defects. Multiple studies [14, 26, 34] focus on effective configuration combination strategies for testing and show that simple algorithms such as *most-enabled-disabled* are the most effective. Efforts [21, 64, 66] also exist to automatically detect configuration defects in traditional software. In contrast, our work is the first to systematically study setting defects in Android apps.

*Empirical studies for Android app defects.* A number of empirical studies investigate different types of Android app defects [2, 9, 15]. For example, Hu et al. [15] study the WebView defects, while Fan et al. [8, 9] and Su et al. [36] study the framework-specific crash defects. But they do not cover setting defects addressed by our work. Some studies investigate Android configurations [10, 16, 18], but these configurations denote different Android SDK versions, device screen sizes, or configuration files (e.g., *AndroidManifest.xml*) of Android apps. These configurations are different from the system settings considered in our work.

**Automated Android app testing.** A number of automated GUI testing techniques have been proposed [4, 7, 12, 13, 20, 23, 25, 27, 29, 35, 37, 49]. However, these testing techniques are limited to crash defects due to lack of strong test oracles. In contrast, our testing technique, informed by our study, leverages the idea of metamorphic testing to detect both crash and non-crash setting defects. Adamsen et al. [1] also use specific metamorphic relations to enhance existing test suites for Android, but they do not target setting defects. Some previous work explores limited types of setting defects. Sadeghi et al. [32] propose PATDROID, which uses combinatorial testing to automatically detect permission defects. Similarly, Lu et al. [22] propose PREVEST, which uses symbolic execution and combinatorial testing to detect crashes induced by changing app-specific preferences and some system settings. However, our work has two significant differences from theirs. First, we systematically explore different system settings (typically provided by the system app *Settings*), while they explore only limited types of settings. Second, SETDROID can detect non-crash setting defects, while PATDROID and PREVEST can detect only crash ones. Our evaluation in Section 4.3.4 also shows these differences. Riganelli et al. [31] use screen rotations to detect data loss defects. However, they can detect only the setting defects induced by screen rotations, while SETDROID can detect many different setting defects.

7 CONCLUSION

We have conducted the first empirical analysis of setting defects in Android apps and found that most apps are affected by setting defects. We have identified five major root causes and four types of consequences of these defects. The study results guide our design of setting-wise metamorphic fuzzing for finding setting defects, realized in the SETDROID tool. SETDROID finds 59 previously-unknown setting defects from 26 open-source and 5 industrial apps. These defects have diverse root causes and consequences; the majority (49 out of 59) cause non-crash failures and could not be detected by existing tools. We have open-sourced both SETDROID and our dataset to facilitate replication and future research at [https://github.com/setting-defect-fuzzing/home](https://github.com/setting-defect-fuzzing/home).

ACKNOWLEDGMENTS

This work is partially supported by the National Key R&D Program of China No.2020AAA0107800, NSFC Project No. 62072178, NSFC Project No. 61632005, and the project of STC of Shanghai No. 1951103602. Ting Su and Zhendong Su are partially supported by a Google Faculty Research Award. Tao Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China.
REFERENCES


