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:
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 *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464806

*Corresponding author.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464806

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.

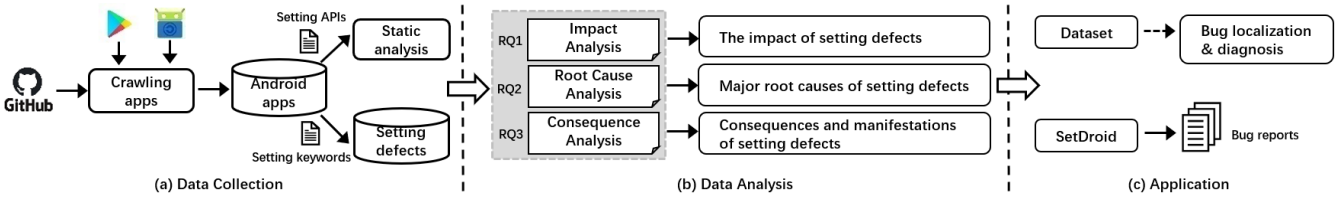


Figure 1: Overview of our study, including three steps: (a) data collection, (b) data analysis, and (c) application.

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 already 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

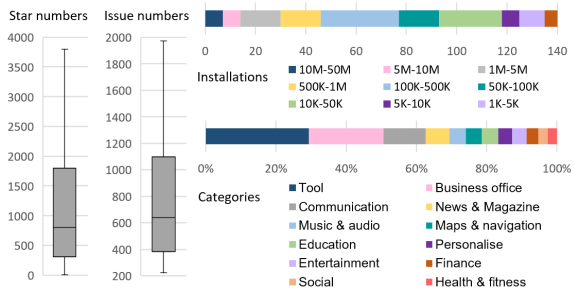


Figure 2: Characteristics of the 180 apps under study.

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.

Setting Categories	Keywords	Description
Network and connect	Bluetooth, WLAN, NFC, internet, network, hot-spot, mobile, wifi, airplane	Manage the device's network mode (WiFi, mobile data or airplane mode), and the connection with other devices (such as Bluetooth).
Location and security	location, device only, phone only, GPS, high accuracy, screen lock, fingerprint	Manage the device's security settings (*e.g.*, how to unlock 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).
Sound	vibrate, ringtone, do not disturb, slient	Manage the device's sound-related options (*e.g.*, the "do not disturb" mode can completely mute the device).
Battery	power save, battery	Manage the power saving mode and the list of apps that are not restricted by the power saving mode (the power saving whitelist).
Display	orientation, vertical, horizontal, split screen, Multi-window, screen resulotion, brightness, landscape, portrait, rotate	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).
Apps and notifications	permission, disable, notification	Manage the runtime permissions of apps and whether they can push notifications to users.
Developer	developer option, keep activity	A number of advanced settings to simulate specific running environment (*e.g.*, enable "Force RTL (right to left) layout direction").
Accessibility	accessibility, talkback, text-to-speech, color correction, color inversion, high contrast text	Customize the device to be more accessible, *e.g.*, adjusting the contrast of UI interface and opening the screen reader.
Other Settings	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	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.

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

Setting Categories	#Apps using settings	#Apps were affected	#Setting Defects
Network and connect	86	68	326
Location and security	47	10	14
Sound	67	16	50
Battery	57	10	18
Display	109	78	226
Apps and notification	134	49	121
Others	-	-	319
#Total	171	162	1,074

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.

Table 2 (column "#Setting Defects") classifies the defects reflected by the 1,074 bug reports in our dataset according to the defects' setting categories. Similar to the observation from column "#Apps were affected", we can see that the three categories "Display", "Network and connect", and "Apps and notifications" lead to the majority (701/1,074 ≈65.2%) of setting defects. This result indicates that the settings in these categories are more likely to cause setting defects than the other ones.

Answer to RQ1: Our study reveals that 95% (171/180) of apps in our dataset use system settings according to the setting-related APIs used in the app code. 90% (162/180) of apps are ever affected by setting defects. Thus, setting defects indeed have a wide impact on the app correctness.

3.2 RQ2: Root Causes of Setting Defects

To analyze the root causes, we focus on investigating 482 *fixed* bug reports with explicitly-linked code fixing commits. Table 3 summarizes these root causes ordered by their corresponding numbers of bug reports from the most to least. We next explain and illustrate these root causes.

3.2.1 Incorrect Callback Implementations. To properly handle settings, developers are required to properly implement the callback methods, which are called by the Android system when some settings change. For example, when users grant or deny permissions, the callback `onRequestPermissionsResult()` is called; when users

Table 3: Major root causes of setting defects

Root Causes	#Bug Reports
Incorrect callback implementations	164
Lack of setting checks	143
Fail to adapt user interfaces	103
Lack of considering Android versions	27
Mutual influence between settings	12
Other minor reasons	33

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 `PowerManager#isPowerSaveMode()` (Line 3), but do not check whether the app is in the whitelist of the power saving mode by `PowerManager# 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

```
1   public void onRequestPermissionsResult (int requestCode, String[] p, int[] grantResults) {
2     if (requestCode == REQUEST_STORAGE_PERMISSION) {
3       if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
4         showStartupScreensAndDialogs(anki.getSharedPrefs(this), 0);
5 +       invalidateOptionsMenu();}
6   }}
```

Figure 3: Patch for AnkiDroid's Issue #4951

```
1   public static boolean isPowerSave(Context context){
2     PowerManager pm = context.getSystemService(Context.POWER_SERVICE);
3     isSave = pm.isPowerSaveMode()
4 -   return pm != null && isSave();
5 +   boolean isIgnore = pm.isIgnoringBatteryOptimizations();
6 +   return pm != null && isSave && !isIgnore;
7   }
```

Figure 4: Patch for NextCloud's Issue #2889

```
1 + if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
2 +   NotificationManager nm = context.getSystemService(Context.NOTIFICATION_SERVICE);
3 +   if (!nm.isNotificationPolicyAccessGranted()) {
4 +     Intent i = new Intent(Settings.ACTION_NOTIFICATION_POLICY_ACCESS_SETTINGS);
5 +     context.startActivity(i);}
6 + }
7   audioManager.setStreamVolume(streamType, index, flags);
```

Figure 5: Patch for Openlauncher's Issue #67

crashes if it is opened offline. The root cause is that the app calls `locationManager#getlastknownlocation` to get the current geographic location via network. When the network is closed, this call returns a `NULL` value, which is later used by `getLatitude()`. As a result, the app crashes by a `NullPointerException`.

One typical example of implicit mutual influence is that when the power saving mode is enabled, some settings such as *location*, *network*, and *animation* are affected. This factor may make the failure diagnosis quite difficult. For example, in *Clover* [44]'s Issue #360, the app is always stuck for unknown reasons and then forced closed. The developers finally locate the culprit: the animation is *automatically* disabled when the power saving mode is on. The app is stuck because the startup animation cannot be played.

3.2.5 Lack of Considering Android Versions. The Android system evolves fast, and some setting mechanisms may change. This factor may lead to some device-specific setting defects. For example, in *Openlauncher* [52]'s Issue #67: when a user changes the volume while the "do not disturb" (DND) mode is enabled (in the notification setting category), the app crashes. The root cause is that since Android's Nougat version, if an app is in the DND mode, the app needs to get the `ACCESS_NOTIFICATION_POLICY`) permission before it can use `AudioManager` to change the volume. As shown in Figure 5, the developers fix this defect by checking whether the the system version is above Android 7.0 (Line 1) before calling the `AudioManager#setStreamVolume()` (Line 7).

Answer to RQ2: Our study distills 5 major root causes of setting defects. Among these causes, *incorrect callback implementations*, *lack of setting checks*, and *fail to adapt user interfaces* are responsible for the majority (410/482 ≈85.1%) of setting defects. *Mutual influence between settings* and *lack of considering Android versions* could lead to setting defects, despite only a few (39/482 ≈8%).

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., slaggy 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

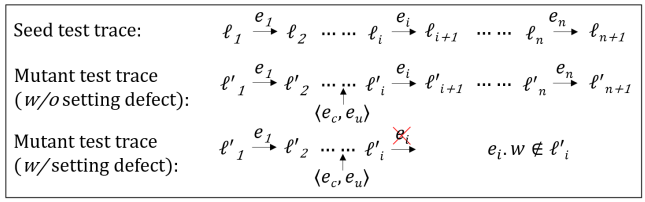
4.1.1 High-level Idea. Our key insight is that, in most cases, the app behaviors should keep *consistent* if a given setting is changed and later *properly* restored, *or* show expected differences if not restored. Otherwise, a likely setting defect is found. For example, an app's functionalities should 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 $\langle e_c, e_u \rangle$ 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.*, $E = [e_1, \ldots, e_i, \ldots, e_n]$, 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) $L = [\ell_1, \ldots, \ell_i, \ldots, \ell_{n+1}]$, 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 $\ell_{i+1} = e_i(\ell_i), i \geq 1$. By injecting a pair of new events $\langle e_c, e_u \rangle$ into E, we can obtain a mutant GUI test $E' = [e'_1, \ldots, e_c, \ldots, e_u, \ldots, e'_n]$ that can be executed on P to obtain a sequence of GUI layouts (pages) $L' = [\ell'_1, \ldots, \ell_c, \ldots, \ell_u, \ldots, \ell'_{n+1}]$. 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 $\langle e_c, e_u \rangle$ 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].



The preceding figure illustrates Rule I: under these two injection strategies, if there exists one GUI event $e_i \in E'$ and its target widget $e_i.w$ cannot be located on the corresponding layout $\ell'_i \in L'$ (ℓ'_i corresponds to $\ell_i \in L$), then a likely setting defect is found. Because it likely indicates that the app's behaviors are affected. Formally,

$$\exists e_i . e_i.w \in \ell_i \wedge e_i.w \notin \ell'_i \tag{1}$$

Oracle checking rule II. Under Rule II, we inject only e_c into E (e_u is ignored). This rule aims to confirm that changing a given setting, *e.g.*, *languages*, *hour format* (12-hour or 24-hour format), indeed leads to some GUI changes. For example, when the default language

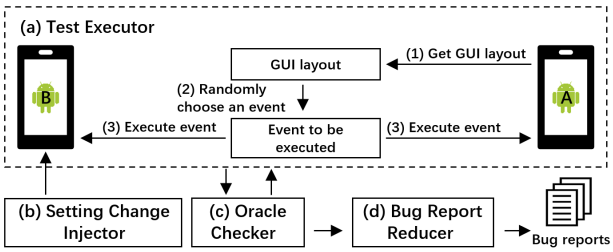


Figure 6: Workflow of SetDroid to find setting defects.

is changed, we check whether the texts in ℓ_i and ℓ_i' are indeed in the expected different languages while no other inconsistencies appear. In practice, we use the language identification tool named langid [19] to determine the language of each text.

Thus, Rule I does *equal checking* on GUI consistency and applies to three common consequences of setting defects, *i.e.*, *crash* (a special case of GUI inconsistency), *functionality failure*, and *problematic UI display*. Rule II does *inequality checking* on GUI consistency and applies to the checking of *disrespect of setting changes*.

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 $\langle e_c, e_u \rangle$ at any position of a seed test rather than only at its beginning. Moreover, if one pair of $\langle e_c, e_u \rangle$ is injected and no setting defect is found, the next same pair of $\langle e_c, e_u \rangle$ 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 $\langle e_c, e_u \rangle$ 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

Table 4: List of pairs of events for setting changes

Setting	Oracle Rule	Injection Strategy	Pair of events for setting changes
Network	I	Immediate	⟨turn on airplane, turn off airplane⟩
Network	I	Lazy	⟨turn on airplane, turn off airplane⟩
Network	I	Lazy	⟨switch to mobile data, switch to Wi-Fi⟩
Location	I	Lazy	⟨turn off location, turn on location⟩
Location	I	Lazy	⟨switch to "device only", switch to "high accurancy"⟩
Sound	I	Lazy	⟨turn on "do not disturb", turn off "do not disturb"⟩
Battery	I	Immediate	⟨turn on the power saving mode, add the app into the whitelist⟩
Battery	I	Lazy	⟨turn on the power saving mode, turn off the power saving mode⟩
Display	I	Immediate	⟨switch to landscape, switch to portrait⟩
Display	I	Immediate	⟨turn on multi-window, turn off multi-window⟩
Permission	I	Lazy	⟨turn off permission, turn on permission⟩
Language	II	-	⟨change system language, -⟩
Time	II	-	⟨change hour format, -⟩

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

Table 5: List of the 42 setting defects found by SetDroid and the detailed statistics of true positives and false positives.

App ID	App name	#Installations	#Stars	Issue ID	Issue state	Cause setting	Consequence	#FP^{I}	#TP^{I}	#FP^{II}	#TP^{II}
1	APhotoManager	-	162	#175	Confirmed	Permission	Crash	0	3	0	0
2	A2DP Volume	100K-500K	71	#295	Fixed	Display	Crash	0	10	0	0
				#294	Fixed	Display	Data lost				
				#291	Fixed	Display	Crash				
				#290	Fixed	Display & Permission	Data lost				
				#289	Confirmed	Developer	Crash				
3	Always On	10M-50M	121	#2476	Confirmed	Language	Disrespect of Settings	3	0	3	6
				#2475	Confirmed	Language	Incomplete translation(5)				
4	AnkiDroid	5M-10M	3.2K	#5407	Fixed	Permission	Stuck	0	4	7	0
5	AntennaPod	500K-1M	3.3K	#4227	Fixed	Network	Lack of refresh	1	2	7	1
6	Commons	50K-100K	649	#3906	Discussion	Location	Infinite loading	0	9	30	0
				#3134	Confirmed	Permission	Crash				
7	ConnectBot	1M-5M	1.6K					1	8	8	0
8	FillUp	100K-500K	29					3	0	0	0
9	Forecastie	10K-50K	609	#505	Fixed	Permission	Lack of prompt	1	3	1	5
				#504	Fixed	Language	Incomplete translation(5)				
				#358	Confirmed	Display	Data lost				
10	Good Weather	5K-10K	196	#62	Waiting	Network	Infinite loading	0	6	4	51
				#61	Waiting	Location	Lack of prompt				
				#55	Waiting	Language	Language confusion				
11	Notepad	100K-500K	156					0	3	4	1
12	Omni Notes	100K-500K	2.2K	#776	Fixed	Permission	Lack of prompt	0	9	3	3
				#775	Fixed	Location	Functionality failure				
				#764	Fixed	Language	Disrespect of Settings				
				#695	Fixed	Language	Incomplete translation(2)				
13	Opensuduku	10K-50K	209	#93	Confirmed	Language	Incomplete translation(7)	1	2	1	7
14	RedReader	50K-100K	1.1K	#783	Discussion	Network	Infinite loading	0	4	291	41
				#749	Confirmed	Language	Incomplete translation(23)				
15	Timber	100K-500K	6.4K	#459	Confirmed	Display	Data lost	0	4	0	9
				#458	Waiting	Permission	Crash				
				#454	Waiting	Permission	Incomplete translation(9)				
16	Vanilla Music	500K-1M	777	#1048	Waiting	Display	Crash	1	7	0	0
17	Wikipedia	50M-100M	1.3K					0	6	4	0
18	OpenBikeSharing	1K-5K	58	#55	Confirmed	Display	Functionality failure	3	8	0	0
19	Suntimes	-	134	#420	Fixed	Location	Infinite loading	1	2	0	0
20	RadioBeacon	-	43	#249	Confirmed	Network	Stuck	3	2	10	1
				#234	Confirmed	Permission	Crash				
21	RunnerUp	10K-50K	511	#923	Fixed	Permission	Lack of prompt	0	1	0	0
22	Amaze	1M-5M	3K	#1965	Fixed	Display & Permission	Black screen	4	21	18	0
				#1964	Fixed	Display & Permission	Data lost				
				#1920	Fixed	Network	Lack of prompt				
				#1919	Fixed	Display & Permission	Crash				
				#1885	Fixed	Permission	Crash				
23	Habits	1M-5M	3.6K	#620	Fixed	Display	Data lost	2	2	0	1
				#599	Fixed	Language	Incomplete translation(2)				
24	Materialistic	100K-500K	2.1K	#1429	Waiting	Network	Lack of refresh	1	8	144	1

14 CPU days). For each bug report, SetDroid provides the failure-triggering 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 TP on real Android devices before reporting these TPs. When the triggering trace and consequences of a TP are different from those of each of all bug reports submitted by us so far, we submit a new 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. If the bug report is not marked as a duplicate one 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 [58], QQMail [53], 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.

4.3.2 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 only 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 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 6: Setting defects found in the five industrial apps.

ID	App	Setting	Consequence
1	QQMail	Permission	Functionality failure
2	QQMail	Permission	Crash
3	Wechat	Permission	Functionality failure
4	Wechat	Permission	Functionality failure
5	Wechat	Language	Problematic UI display
6	Wechat	Language	Incomplete translation
7	Wechat	Network	Stuck
8	Wechat	Network	Functionality failure
9	CapCut	Network	Infinite loading
10	CapCut	Permission	Functionality failure
11	CapCut	Display&Permission	Problematic UI display
12	CapCut	Network	Functionality failure
13	TikTok	Network	Functionality failure
14	TikTok	Permission	Functionality failure
15	TikTok	Locaion	Functionality failure
16	AlipayHK	Language	Functionality failure
17	AlipayHK	Location	Functionality failure

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 RQ.4: 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.

Setting	All settings						Bluetooth, network and location				Permission			
Tool	Baseline A		Baseline B		SetDroid		Prefest		SetDroid		PATDroid		SetDroid	
Consequence	C	NC	C	NC	C	NC	C	NC	C	NC	C	NC	C	NC
#Defects	0	0	3	0	9	33	0	0	0	10	2	0	6	8

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) Prefest 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 RQ.5: 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 Prefest, 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 Prefest 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*.

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. 19511103602. 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

[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] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1323–1334. https://doi.org/10.1145/3377811.3380392

[3] Domenico Amalfitano, Vincenzo Riccio, Ana CR Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. In *Software Testing, Verification and Reliability (STVR)*. e1654. https://doi.org/10.1002/stvr.1654

[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 641–660. https://doi.org/10.1145/2509136.2509549

[5] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01, Hong Kong University of Science and Technology. https://arxiv.org/abs/2002.12543

[6] 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

[7] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 1–12. https://doi.org/10.1145/3377811.3380402

[8] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 486–497. https://doi.org/10.1145/3238147.3238170

[9] 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 *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 408–419. https://doi.org/10.1145/3180155.3180222

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

[11] fpernice518. 2018. NextCloud issue #2979. Retrieved 2021-1 from https://github.com/nextcloud/android/issues/2979.

[12] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 269–280. https://doi.org/10.1109/ICSE.2019.00042

[13] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. 2020. Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 557–568. https://doi.org/10.1109/ICSME46990.2020.00059

[14] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. In *Empirical Software Engineering (EMSE)*. 674–717. https://doi.org/10.1145/3382025.3414985

[15] 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

[16] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2017. Developer mistakes in writing Android manifests: an empirical study of configuration error. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 25–36. https://doi.org/10.1109/MSR.2017.41

[17] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: a systematic literature review. In *IEEE Transactions on Reliability*. 45–66. https://doi.org/10.1109/TR.2018.2865733

[18] Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. 2018. Configurations in Android testing: they matter. *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile)* (2018), 1–6. https://doi.org/10.1145/3243218.3243219

[19] langid Team. 2021. langid. Retrieved 2021-1 from https://github.com/saffsd/langid.py.

[20] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[21] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 445–456. https://doi.org/10.1145/2642937.2643001

[22] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. 2019. Preference-wise testing for Android applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 268–278. https://doi.org/10.1145/3338906.3338980

[23] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 599–609. https://doi.org/10.1145/2635868.2635896

[24] malinajirka. 2019. WordPress issue #10096. Retrieved 2021-1 from https://github.com/wordpress-mobile/WordPress-Android/issues/10096.

[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 94–105. https://doi.org/10.1145/2931037.2931054

[26] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 643–654. https://doi.org/10.1145/2884781.2884793

[27] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. Sig-droid: automated system input generation for android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 461–471. https://doi.org/10.1145/2509136.2509549

[28] mzorz. 2017. WordPress issue #6026. Retrieved 2021-1 from https://github.com/wordpress-mobile/WordPress-Android/issues/6026.

[29] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications (ISSTA). In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164. https://doi.org/10.1145/3395363.3397354

[30] pcqpcq. 2021. opensource-android-apps. Retrieved 2021-1 from https://github.com/pcqpcq/open-source-android-apps/.

[31] 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 *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 141–152. https://doi.org/10.1145/3395363.3397379

[32] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: permission-aware gui testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*. 220–232. https://doi.org/10.1145/3106237.3106250

[33] setting-defect fuzzing. 2021. Dataset. Retrieved 2021-1 from https://github.com/setting-defect-fuzzing/home.

[34] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. 632–642. https://doi.org/10.1109/ICSE.2017.64

[35] Ting Su. 2016. FSMdroid: guided GUI testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 689–691. https://doi.org/10.1145/2889160.2891043

[36] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why my app crashes? Understanding and benchmarking framework-specific exceptions of Android apps. *IEEE Transactions on Software Engineering (TSE)*. https://doi.org/10.1109/TSE.2020.3013438

[37] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*. 245–256. https://doi.org/10.1145/3106237.3106298

[38] AlipayHK Team. 2021. AlipayHK. Retrieved 2021-1 from https://www.alipayhk.com.

[39] Android Team. 2021. Android Developers Documentation. Retrieved 2021-1 from https://developer.android.com.

[40] Android Team. 2021. Android Help. Retrieved 2021-1 from https://support.google.com/android.

[41] AnkiDroid Team. 2021. AnkiDroid. Retrieved 2021-1 from https://github.com/ankidroid/Anki-Android.

[42] Android Team. 2021. Request App Permissions. Retrieved 2021-1 from https://developer.android.com/training/permissions/requesting#perm-check.

[43] CapCut Team. 2021. CapCut. Retrieved 2021-1 from https://lv.faceueditor.com.

[44] Clover Team. 2021. Clover. Retrieved 2021-1 from https://github.com/chandevel/Clover.

[45] Commons Team. 2021. Commons. Retrieved 2021-1 from https://github.com/commons-app/apps-android-commons.

[46] Frost Team. 2021. Frost. Retrieved 2021-1 from https://github.com/AllanWang/Frost-for-Facebook.

[47] GitHub Team. 2021. GitHub REST API. Retrieved 2021-1 from https://docs.github.com/en/rest/.

[48] K-9 Team. 2021. K-9. Retrieved 2021-1 from https://github.com/k9mail/k-9.

[49] Monkey Team. 2021. Android Monkey. Retrieved 2021-1 from https://developer.android.com/studio/test/monkey.

[50] NextCloud Team. 2021. NextCloud. Retrieved 2021-1 from https://github.com/nextcloud/android.

[51] OpenFoodFacts Team. 2021. OpenFoodFacts. Retrieved 2021-1 from https://github.com/openfoodfacts/openfoodfacts-androidapp.

[52] Openlauncher Team. 2021. Openlauncher. Retrieved 2021-1 from https://github.com/OpenLauncherTeam/openlauncher.

[53] QQMail Team. 2021. QQMail. Retrieved 2021-1 from https://en.mail.qq.com.

[54] Signal Team. 2021. Signal. Retrieved 2021-1 from https://github.com/signalapp/Signal-Android.

[55] Status Team. 2021. Status. Retrieved 2021-1 from https://github.com/status-im/status-react.

[56] Syncthing Team. 2021. Syncthing. Retrieved 2021-1 from https://github.com/syncthing/syncthing-android.

[57] TikTok Team. 2021. TikTok. Retrieved 2021-1 from https://www.tiktok.com.

[58] WeChat Team. 2021. WeChat. Retrieved 2021-1 from https://www.wechat.com.

[59] WordPress Team. 2021. WordPress. Retrieved 2021-1 from https://github.com/wordpress-mobile/WordPress-Android.

[60] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. In *Software Quality Journal (SQJ)*. 149–201. https://doi.org/10.1007/s11219-018-9418-6

[61] uiautomator2 Team. 2021. uiautomator2. Retrieved 2021-1 from https://github.com/openatx/uiautomator2.

[62] Mario Linares Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: a new perspective for automated mobile app testing. In *International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME.2017.27

[63] 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)*. https://doi.org/10.1145/3238147.3240465

[64] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*. 619–634. https://dl.acm.org/doi/10.5555/3026877.3026925

[65] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. 159–172. https://doi.org/10.1145/2043556.2043572

[66] Sai Zhang and Michael D Ernst. 2014. Which configuration option should I change?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 152–163. https://doi.org/10.1145/2568225.2568251