Characterizing and Finding System Setting-Related Defects in Android Apps

Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su

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* large-scale empirical study to understand and characterize these *system 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 four person-months*) to analyze 1,074 setting defects from 180 popular apps on GitHub. We investigate the impact, root causes, and consequences of these setting defects and their correlations. We find that (1) setting defects have a wide impact on apps' correctness with diverse root causes, (2) the majority of these defects (≈70.7%) cause non-crashing (logic) failures, and (3) some correlations exist between the setting categories, root causes, and consequences. Motivated and informed by these findings, we propose two bug-finding techniques that can synergistically detect setting defects from both the GUI and code levels. Specifically, at the GUI level, we design and introduce *setting-wise metamorphic fuzzing*, the *first* automated dynamic testing technique to detect setting defects (causing crash *and* non-crashing failures, respectively) for Android apps. We implement this technique as an end-to-end, automated GUI testing tool named SETDROID. At the code level, we distill two major fault patterns and implement a static analysis tool named SETCHECKER to identify potential setting defects. We evaluate SETDROID and SETCHECKER on 26 popular, open-source Android apps, and they find 48 unique, previously-unknown setting defects. To date, 35 have been confirmed and 21 have been fixed by app developers. We also apply SETDROID and SETCHECKER 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. After that, we collaborate with ByteDance and deploy these two bug-finding techniques internally to stress-test TikTok, one of its major app products. Within a two-month testing campaign, SETDROID successfully finds 53 setting defects, and SETCHECKER finds 22 ones. So far, 59 have been confirmed and 31 have been fixed. All these defects escaped from prior developer testing. By now, SETDROID has been integrated into ByteDance's official app testing infrastructure named FASTBOT for daily testing. These results demonstrate the strong effectiveness and practicality of our proposed techniques.

Index Terms—Empirical study, System Settings, Android Apps, GUI Testing, Static Analysis

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 type of 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* [1], a popular website and blog management app (which has 10,000,000∼50,000,000 installations on Google Play and 2,600 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 [2]. The other defect was triggered when a post-draft was created under the airplane mode; *WordPress* constantly crashed at the next startup [3]. Both defects were labeled as *critical* but escaped from pre-release developer testing.

Moreover, these setting defects can be frustrating. For example, *NextCloud* [4] is a popular on-premise file-sharing app (which has 1,000,000∼5,000,000 installations on Google Play and 2,600 stars on GitHub). A user reported that he could not use the auto-upload functionality for unknown reasons [5]. 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.

- *Jingling Sun, Ting Su, Kai Liu and Geguang Pu are with Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China. Kai Liu is also a research intern at Bytedance when this work is conducted.*
- *Chao Peng and Zhao Zhang are with Product RD & Infrastructure, Bytedance Network Technology, Beijing, China.*
- *Tao Xie is with the Key Laboratory of High Confidence Software Technologies, Peking University.*
- *Zhendong Su is with Department of Computer Science, ETH Zurich, Switzerland.*

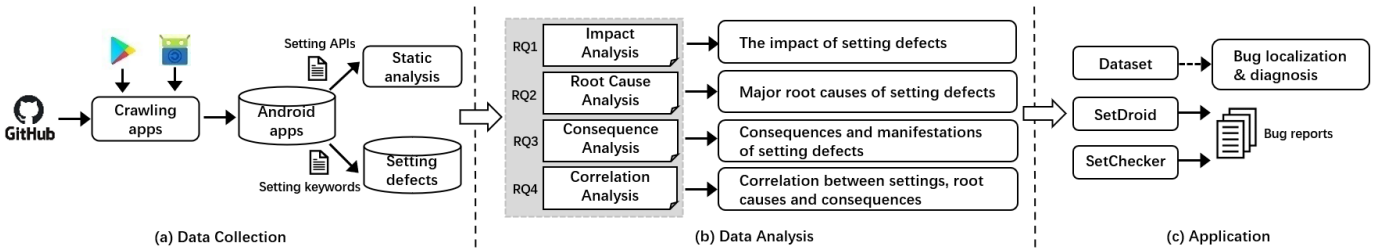


Fig. 1. Overview of our study, including three steps: (a) data collection, (b) data analysis, and (c) application.

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 [6], [7] and screen orientation [8]). Thus, we are lack of comprehensive understanding on these setting defects. On the other hand, state-of-the-art fully-automated GUI testing techniques [9], [10], [11] 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 [11], [12], [13] due to the lack of strong test oracles [14], while many setting defects are non-crashing logic ones that lead to app freezing, functionality failures, or problematic GUI display, which our study will demonstrate.

To fill this gap, we conduct the *first* systematic study to understand and characterize 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 themselves?
- **RQ4 (correlations)**: Are there any correlations between setting categories, root causes, and consequences?

Answering these questions can help understand the impact, root causes, and consequences of these setting defects, and also benefit bug finding and diagnosing. Specifically, Fig. 1 shows the overview of this study, which contains three steps. In step (a), we study the Android documentation [15], [16] to summarize the main setting categories and the keywords of settings (Section 2.1). We then use these keywords to mine 1,074 setting defects from the issue repositories of 180 popular Android apps on GitHub (Section 2.2). In step (b), we carefully study these defects by reviewing the bug reports and analyzing the root causes, fixes, consequences, and the correlations (Section 2.3), and present the answers to RQ1∼RQ4 in Section 3. In step (c), informed by the study, we propose SETDROID and SETCHECKER to help find setting defects in Section 4.

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 ones. We also note that only a few setting defects (≈2%) are caused by the mutual influence between two settings, while some were device-specific due to the fast evolution of Android. 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-crashing* failures, which indeed poses a significant challenge on existing fully-automated GUI testing techniques.

By analyzing the correlations between the setting categories and their common root causes and consequences, we find that the setting defects from *all* setting categories could lead to non-crashing failures. Moreover, the lack of setting checks is the *most common* root cause for almost all setting categories. This finding inspires us to use static analysis to detect such defects by summarizing code-level fault patterns. On the other hand, we find that *almost all* setting defects manifest themselves on the GUI pages. This finding inspires us to find setting defects via black-box GUI-level testing. In all, these correlations offer us deep understanding of setting defects and useful insights for designing effective bug-finding techniques (detailed in Section 3.4).

Specifically, guided by the preceding findings, we propose *setting-wise metamorphic fuzzing*, the *first* automated testing approach at the GUI level to effectively detect setting defects without requiring 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 crashing and non-crashing setting defects. We apply SETDROID on 26 popular, open-source Android apps. SETDROID has successfully discovered 42 unique, previously-unknown setting defects. So far, 31 have been confirmed and 20 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 [17] and QQMail [18] from Tencent, TikTok [19] and CapCut [20] from ByteDance, and AlipayHK [21] 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-crashing failures, which cannot be detected by existing fully automated dynamic testing tools (corroborated by our evaluation in Section 5).

Further, we distill two major fault patterns (tackling the major root cause, *i.e.*, *lack of setting checks*) at the code level, and build a static analysis tool named SETCHECKER to identify potential setting defects. We apply SETCHECKER on the same 26 open source apps tested by SETDROID.

SETCHECKER successfully finds 17 unique, previously-unknown setting defects, 6 of which have not been found by SETDROID. So far, 4 have been confirmed and 1 fixed by the developers. The result shows that SETCHECKER can synergistically complement SETDROID. We give more detailed analysis on SETDROID and SETCHECKER in Section 5.5.

Afterward, we collaborate with ByteDance and deploy SETDROID and SETCHECKER internally to help find setting defects in TikTok, another major app product from ByteDance with billions of monthly active users worldwide. Within a two-month testing campaign, SETDROID successfully finds 53 setting defects. To date, 48 of them have been confirmed, and 20 have been fixed. On the other hand, SETCHECKER successfully detects 22 defects. To date, 11 have been confirmed and fixed. By now, SETDROID has been intergated into ByteDance's official app testing infrastructure named FASTBOT [22], [23]. These results clearly demonstrate the strong effectiveness and practicality of our techniques. In summary, this article makes the following major contributions:

- We conduct the *first* systematic study on setting defects to understand and characterize their impact, root causes, consequences, and correlations.
- Informed by this study, we propose two bug-finding techniques to detect setting defects. At the GUI level, we introduce setting-wise metamorphic fuzzing, the *first* automated GUI testing technique to effectively detect setting defects. At the code level, we distill two major fault patterns and use static analysis to find setting defects.
- We implement the two bug-finding techniques as two tools namely SETDROID and SETCHECKER, respectively. SETDROID and SETCHECKER have revealed 48 setting defects in 26 open-source apps and 92 defects in five industrial apps. The majority of these defects cause non-crashing failures and could not be detected by existing testing tools. Our evaluation also shows that SETDROID and SETCHECKER outperform prior techniques and complement each other in finding setting defects.
- We have made our tools and dataset publicly available at *https://github.com/setting-defect-fuzzing/home*, to facilitate the replication of our work as well as the future research in this direction.

In an earlier version [24] of the work in this article, we studied the setting defects in Android apps and developed the setting-wise metamorphic testing tool named SETDROID. In this article, we have made substantial extensions in six aspects. (1) We investigate the correlations between the setting categories and their root causes and consequences (corresponding to the new research question RQ4). Based on this analysis, we obtain some new observations and insights, which were not identified by our prior work (see Section 3.4). Based on the correlation analysis, we find that the setting defects from some setting categories are more suitable for static analysis than dynamic analysis. (2) We propose two generic optimizations for SETDROID in terms of testing efficiency and precision (see Section 4.1.4). To improve the efficiency of SETDROID, we use static analysis to identify and run only relevant setting changes against the app under test. This optimization reduces 15∼85% testing time. To improve the precision of SETDROID, we analyze the reasons of false positives and reduce one major type of false positives related to the language setting. This optimization reduces the prior false positive rate from 80.8% to 19.4%. We confirm that both optimizations do not incur any false negatives. (3) Informed by the answers to RQ4, we develop a static analysis tool named SETCHECKER to help find setting defects. Specifically, we summarize two major fault patterns and use control/data-flow analysis to find setting defects at the code level (see Section 4.2). (4) We evaluate the effectiveness of SETCHECKER on 26 apps (see Section 5.3), and compare SETCHECKER with other static analysis tools (see Section 5.4). SETCHECKER finds 6 new setting defects, which were not found by SETDROID; SETCHECKER also outperforms other static analysis tools in finding setting defects. We further compare and analyze the effectiveness of SETDROID and SETCHECKER in Section 5.5 (corresponding to the new research question RQ7). Indeed, SETDROID and SETCHECKER can complement each other. (5) We collaborate with ByteDance and apply SETDROID and SETCHECKER internally to test TikTok, one major app product of ByteDance. SETDROID and SETCHECKER successfully find 53 and 22 setting defects, respectively, which escaped from developer testing. SETDROID has been integrated into ByteDance's official app testing infrastructure named FASTBOT. (6) We open-source our tools and dataset at *https://github.com/setting-defect-fuzzing/home*, to facilitate the replication of our work and motivate the future research in this direction.

2 EMPIRICAL STUDY METHODOLOGY

2.1 Summarizing Setting Categories

To systematically summarize the setting categories, we inspect the Android documentation [15], [16] 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; 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 follow the 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, defect descriptions, reproducing steps, fix patches, and discussions. Specifically, we collect the app subjects as follows.

- We use GitHub's REST API [25] 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

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.

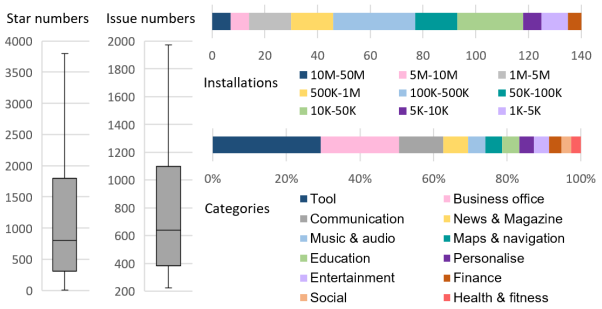


Fig. 2. Characteristics of the 180 apps under study.

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 over 200 closed 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 simple demo apps to illustrate third-party Android libraries). Finally, we attain 180 Android apps as our study subjects.

Fig. 2 shows the characteristics of the 180 apps in terms of the numbers of stars and issues (*i.e.*, bug reports) on GitHub, the installations on Google Play, and the 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. For example, we exclude the bug reports that just mention settings.
- When we do not have clear clues from bug reports, we reproduce the failures to confirm whether they reflect setting defects. To reproduce the failures, we need to obtain the corresponding app versions. If the app versions are no longer available on the app markets (Google Play or F-Droid), we will check whether the app's GitHub repository releases the corresponding app versions. Otherwise, we will pull the corresponding version of source code and manually build the app. We reproduce the failures based on the steps described in the corresponding bug reports.

Finally, we successfully attain 1,074 valid bug reports as the dataset for our subsequent analysis. Among these bug reports, 482 are closed and linked with 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; apps use method `checkSelfPermission` to check whether specific permissions have been granted; apps use the system variable `Settings.System.SCREEN_BRIGHTNESS` to read the current screen brightness. 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 [26]. To investigate the impact of setting defects, we focus on the 1,074 setting defects collected from the 180 apps. We inspect how many apps out of the 180 apps were affected by these setting defects and which setting categories these setting defects belong to.

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 [15] or Stack Overflow [27] 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. If necessary, we reproduce the failures to observe the consequences (Section 2.2.3 describes the reproduction process).

Specifically, similar to the analysis on root causes, the two co-authors first work on a common set of bug reports to identify the consequences. Then, the two co-authors discuss together with the other co-authors to reach a consensus on the initial categories. After that, the four co-authors work separately on the remaining bug reports to classify the consequences. They discuss and cross-check together when the categories need to be updated (*e.g.*, add, merge, or modify categories).

In addition, we studied the manifestations of these 1,074 setting defects, *i.e.*, whether these defects will manifest as GUI defects (*i.e.*, whether a defect would be manifested through GUI pages when it is triggered) and whether these defects would lead to some GUI differences (*i.e.*, whether a defect would lead to some GUI differences when the relevant setting is on or off). We count the number of bugs that will manifest as GUI defects and lead to some GUI differences. These information helps us to design effective bug finding techniques.

2.3.4 Analysis Method of RQ4

To answer RQ4, we use the analysis results from RQ1∼RQ3 as the basis, and summarize the root causes and the consequences within each setting category. We investigate the correlations between the setting categories, root causes, and consequences from different aspects. For example, we inspect which root causes are the most common across different categories, whether one root cause could lead to different consequences within one setting category, *etc*.

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 multiple settings. Row "#Total" gives the unique number of apps that use settings. Additionally, the data in Table 2 may not characterize the *exact* numbers of apps affected by the settings because it is difficult in practice to collect all possible setting-related APIs. In some cases, some settings do not provide explicit APIs. For example, 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. But when an app uses the APIs in our list, we are sure that the app depends on the corresponding settings. Thus, the current results in Table 2 can be viewed as the 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. For the remaining 9 apps, we manually examined their source code. Indeed, they do not use system settings, and no setting defects were reported. For example, one of the nine app is a calculator app, which has simple functionalities and does not use any system setting. 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. For "Display", we find many apps

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

check the changes of font size, screen brightness, screen orientation, and multi-window mode. For "Location and security", apps use the APIs like `android.location` and `com.google.android.gms.location` for localization. Apps use `android.media.AudioManager` to access volume and ringer mode control in the "Sound" category. The API `PowerManager#isPowerSaveMode` is usually used to check whether the device is in power saving mode by apps in the category "Battery". Although there are only 57 apps that check the battery settings in the code, we find there are more apps that are actually affected by this setting.

Table 2 (column "#Apps were affected") counts which apps were ever affected by setting defects according to the 1,074 bug reports in our dataset. We find that most apps were 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 were 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. We finally identified five major categories of root causes based on the reasons behind these setting defects and the corresponding defect fixing strategies. 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 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 callbacks.

For example, in *AnkiDroid* [28]'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. Fig. 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). As another example, some users will enable the "Do not keep activities" setting option to reduce system resource consumption. However, this setting change requires developers to properly handle Activity lifecycle callbacks (*e.g.*, properly saving app data) because Android may kill any activity running in the background. WordPress's issues #9685 and #5456 lead to fatal crashes due to the developers fail to properly handle app data in the callbacks.

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 Fig. 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* [29]'s Issue #914 leads to the disappearance of some UI elements when the app adapts itself to the multi-window display mode. In *Frost* [30]'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.

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

```
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   }}
```

Fig. 3. Patch for AnkiDroid's Issue #4951

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* [31]'s Issue #1735, the app 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* [32]'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* [33]'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 Fig. 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). Take another case as an example, in *TUM Campus*'s issue #714, a user reported that when he opens WiFi on his device (which is shipped with Android 4.4.2), the app will crash. The root cause is that DFN-PKI, the service used by *TUM Campus* to issue, distribute and check digital certificates, is

```
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   }
```

Fig. 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);
```

Fig. 5. Patch for Openlauncher's Issue #67

not compatible with Android 4.4.2 or lower. These setting defects are induced by compatibility issues but triggered by changing specific system settings.

3.2.6 Other minor reasons

The other reasons for setting defects are usually related to the app's domain-specific logic. For example, in WordPress, when a user edits and saves a post's draft, the app will automatically upload the draft to the server and overwrite the previous saved draft (on the server). However, in WordPress's issue #10525, if a user turns on the Airplane mode and then edits a post's draft, and turns off the Airplane mode while saving the draft, the `UploadStarter()` method in WordPress will be called twice to update the draft on the server. However, these two method calls refer to the same piece of draft. As a result, the latter one will access the draft deleted by the former one, and eventually leads to app crash. Since the bugs in this category are caused by app-specific erroneous logics, we do not categorize them into other generic categories of root causes.

> **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. We detail the four major consequences *w.r.t.* their numbers of defects from the most to least. Specifically, we find that the majority (759/1,074 ≈70.7%) of setting defects lead to non-crashing consequences, in addition to the crashing failure.

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* [34]'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* [35]'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* [36], 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* [37]'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, data loss, and unable to refresh.

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.

In addition, according to our investigation of bug manifestation in Section 2.3.3, most of the setting defects would manifest through GUI pages. According to our statistics, out of all the 1,074 setting defects, 1,000 setting defects (93.1%≈1,000/1,074) manifest as GUI defects, 841 of which (78.3%≈841/1,074) would lead to some GUI differences when the relevant setting is turned on or off.

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

3.4 RQ4: Correlations between Setting Categories, Root Causes, and Consequences

To investigate the correlations between the setting categories and their common root causes and consequences, we count the settings, root causes, and consequences of all bug reports in our dataset as described in Section 2.3.4 and summarize them in Table 4. In Table 4, column "Setting" denotes the setting categories. For column "Root Causes", we count the root causes of the bug reports under the corresponding setting category and sort the root causes in terms of their occurrences in the bug reports from the most to the least (*i.e.*, the root causes are sorted by their popularity within each setting category). Note that we list only the common root causes (which are responsible for more than 5% of the bug reports). For column "Consequences", we sort the consequences in a similar way within each setting category.

3.4.1 What are the correlations?

Based on the results in Table 4, we have three important observations. *First*, the setting defects from *all* the setting categories could lead to non-crashing failures, in addition to crashes. *It indicates that designing effective bug finding techniques for these non-crashing failures is indeed important and useful for all the settings.* This observation also explains why our proposed bug finding techniques in Section 4 can find setting defects from different setting categories. Specifically, "Disrespect of setting" and "Problematic UI display" are the two most common consequences for the five setting categories (*i.e.*, "Network", "Sound", "Battery", "Language", and "Time"). In addition, some consequences happen in only some specific settings, *e.g.*, data loss affects only "Network" and "Display".

Second, "Lack of setting checks" is the most common root cause for almost all setting categories. *It indicates that capturing this type of root cause is most beneficial for finding setting defects*. This observation motivates the design of our static analysis technique in Section 4.2 to focus on this root cause. Moreover, *we find that one type of root cause may lead to different consequences*. For example, due to the lack of setting checks on the network connection, Signal suffers from two defects with distinct consequences. In issue #6447 [38], a user reports that when he switches from WiFi to data (and vice versa), *Signal* cannot receive notifications anymore. In issue #5353 [39], another user complains that if he calls his friends offline, Signal will crash. *On the other hand, we find that one type of consequence could be caused by different root causes*. For example, WordPress's issues #9685 and #5456 discussed in Section 3.2.1 are caused by the incorrect callback implementation, while Issue #67 of Openlauncher [33] discussed in Section 3.2.5 is caused by lack of considering Android versions. But both defects lead to crashes.

Third, although the consequences of setting defects are diverse, the triggering conditions of most setting defects are similar, *i.e.*, when some related setting is changed, the app may show some unexpected behaviors on the GUI pages. In other words, an app can work well under some setting, but when the setting changes, the app may go wrong (*e.g.*, crash, data loss, and functionality failures). This observation motivates the design of our dynamic analysis technique via injecting setting changes in Section 4.1.

TABLE 4
Correlations between the setting categories and their common root causes and consequences.

Setting	Root Causes	Consequences	Suitable techniques
Network	Lack of setting checks, Incorrect callback implementations	Disrespect of setting, Crash, Wrong prompt, Infinite loading, Data loss	Dynamic analysis
Location	Lack of setting checks	Crash, Wrong prompt, Incorrect positioning	Static analysis
Sound	Lack of setting checks, Lack of considering Android version	Disrespect of setting, Crash	Static analysis
Battery	Lack of setting checks	Disrespect of setting, Crash	Static analysis
Display	Incorrect callback implementations, Fail to adapt user interfaces	Crash, Problematic UI display, Data loss, Disrespect of setting	Dynamic analysis
Permission	Lack of setting checks, Incorrect callback implementations	Crash, Wrong prompt	Dynamic/Static analysis
Language	Lack of setting checks, Incorrect callback implementations	Problematic UI display, Crash, Disrespect of setting	Dynamic/Static analysis
Time	Lack of setting checks	Disrespect of setting, Crash	Dynamic/Static analysis

3.4.2 Which bug finding technique is more suitable?

Based on the preceding correlation analysis, we find that the setting defects from some setting categories are more suitable for dynamic analysis (*i.e.*, testing), while others are more suitable for static analysis. In column "Suitable techniques", we indicate which bug finding technique(s) we believe is more suitable or are both suitable. We next discuss the reasons why we reach such conclusions. In Section 5, we empirically compare the effectiveness between the proposed dynamic and static analysis techniques, and the experimental results corroborate our conclusions here.

Next, we discuss our observations and conclusions in detail for each setting category.

- ***Network***. We observe that Android provides many different network-related APIs, and the apps use different ways to manage network connections. In addition, network-related defects have different fault patterns. For example, the network connection is not checked before or during the usage of network-related APIs, and the network connection is not properly recovered after the network type is switched. Thus, it could be difficult to summarize all possible fault patterns with different network-related APIs at the code level. Meanwhile, from the GUI level, the possible consequences of network-related defects (*e.g.*, disrespect of setting, crash, wrong prompt, infinite loading, and data loss) are clear and easier to be identified. Therefore, network-related defects are more suitable for dynamic analysis than static analysis.
- ***Location***. The location-related defects lead to crash, wrong prompt, and incorrect positioning. For the consequence of incorrect positioning, it is difficult for dynamic analysis to identify such defects as these changes will not reflect on the attributes of UI widgets. However, it is simpler to detect such defects at the code level, as they have the similar fault patterns, *i.e.*, the state of location setting fails to be checked before or after the location-related APIs are used. Thus, static analysis is more suitable in this case.
- ***Sound and battery***. According to our observation, it is difficult for dynamic analysis to identify abnormalities for volume (*e.g.*, no sound can be heard) and battery related defects at the GUI level. On the other hand, static analysis is more suitable for these two settings because there are some specific fault patterns at code level. For example, for battery, we can check whether an app fails to check whether it is in the power saving whitelist when the power saving mode is on.
- ***Display***. The changes of display settings could affect app lifecycle. Specifically, the major root cause, *i.e.*, incorrect (lifecycle) callback implementations, can lead to crash, problematic UI display, data loss, and disrespect of setting. Finding such defects at the code level requires tracking the storage and recovery of application specific data, but this tracking is hard to be precise. Meanwhile, it is easier to observe these consequences at the GUI level. Thus, dynamic analysis is more suitable in this case.
- ***Other settings***. For the setting defects from other setting categories (*i.e.*, "Permission", "Language", and "Time"), some fault patterns can be characterized at the code level, but they may not cover all possible faulty cases. In some cases, dynamic analysis will be able to detect more types of setting defects, when we add appropriate oracle rules. Thus, for these settings, both dynamic and static analysis may help.

> **Answer to RQ4**: There are indeed some correlations between setting categories and their root causes and consequences. These correlations provide guidance for designing effective and appropriate bug finding techniques for the setting defects as well as justifying the design of our techniques.

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 [40] to tackle the oracle problem.

4.1.2 Approach

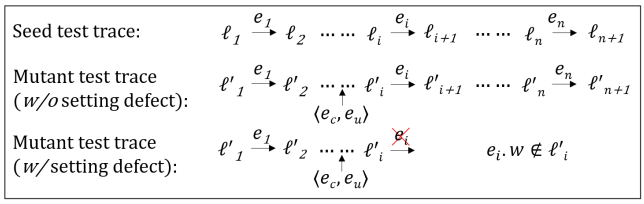
Our approach, *setting-wise metamorphic fuzzing*, *randomly* injects a pair of events $\langle e_c, e_u \rangle$ into a given seed GUI test E to obtain a mutant test E', where e_c changes a given setting, while e_u properly restores the setting or does nothing. By comparing the GUI consistency between 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) $L' = [\ell'_1, \ldots, \ell_c, \ldots, \ell_u, \ldots, \ell'_{n+1}]$. To ease the illustration of our technique, we assume the injection of $\langle e_c, e_u \rangle$ will not break the execution of seed test E. In Section 4.1.3, we explain how our design holds this assumption. Next, 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. The 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. Here, e_c globally changes the power saving setting (affecting the app under test), while e_c can be viewed as restoring the setting for the app under test.
- *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 [41].



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 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 [42] 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*.

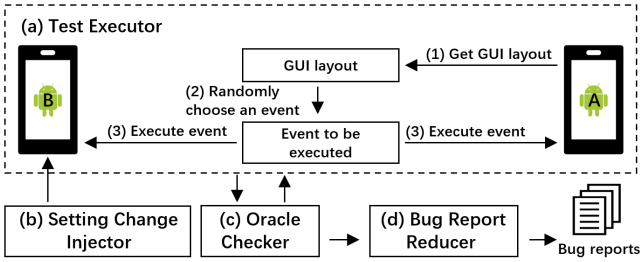


Fig. 6. Workflow of SETDROID to find setting defects.

4.1.3 Design and Implementation of SETDROID

We design and implement our approach as an automated GUI testing tool named SETDROID. Fig. 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. Specifically, 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, which avoids breaking the seed test. We require that devices A and B are identical (*e.g.*, the same Android versions and screen sizes) because doing so ensures that the testing environment is exactly the same and reduces uncertain environmental factors during testing.

We use random seed tests because they are diverse, practical, and scalable to obtain. In practice, we use the UI AUTOMATOR test framework [43] to execute events and obtain GUI layouts. Note that the event generation procedure (*i.e.*, steps (1) and (2)) in the test executor module can be replaced by any automated test input generation algorithms [44], [45], [11] or existing developer-written tests.

Setting Change Injector. During the above process, the setting change injector will randomly inject a pair of events $\langle e_c, e_u \rangle$ into device B. We clarify the design and implementation of this module as follows.

What kinds of $\langle e_c, e_u \rangle$ are supported? Table 5 lists the supported pairs of events for setting changes (see Column "Setting" and "Pair of events for setting changes"). According to our empirical study, these pairs of events in Table 5 can manifest the majority of setting defects and cover the other forms of setting changes. Note that because our oracle checking rules are generic, it is easy and flexible to include other pairs of setting changes in the future. Specifically, the two devices A and B are initialized with the same default setting environment before testing: *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.*

Among the nine setting categories in Table 1, Table 5 only considers seven categories except for the "developer" and "accessibility" ones. We did not consider the "developer" category because we find that developer options are not commonly used by app users but are mainly used by app

TABLE 5
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, -⟩

developers for debugging. Also, the number of setting defects caused by the "developer" category is small, and app developers are usually not interested in such setting defects. We did not consider the "accessibility" category because adapting oracle checking rule I for the accessibility options is difficult and applying oracle checking rule II needs to be ad-hocly designed for each accessibility option. Based on the preceding considerations, we only target the setting categories which can manifest the majority of setting defects according to our study.

How to inject $\langle e_c, e_u \rangle$***?*** Typically, the setting change injector injects e_c after a GUI event in the seed test by coin-flipping and later injects e_u according to the two mutation strategies defined in Section 4.1. Under oracle checking rule I, the setting change injector restores the changed setting (*i.e.*, executing e_u) when necessary. For example, e_c revokes/denies app permission, and e_u grants the permission only when the app requests that permission via a permission request dialog. Note that prompting proper alerts to users is demanded by Android design guidelines to improve user experience. Under oracle checking rule II, the setting change injector injects only e_c and does not inject e_u. Besides the preceding main injection strategies, informed by our study, we adopt the following two important considerations 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 this insight, SETDROID randomly injects the pair of events $\langle e_c, e_u \rangle$ at any position of a seed test (*i.e.*, injecting e_c after each GUI event by coin-flipping and later injecting e_u) rather than only at the beginning of a seed test. 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. In Section 5.2, the comparison between SETDROID and Baseline B (which changes the settings only before starting the apps) justifies the usefulness of this strategy because SETDROID finds much more defects than Baseline B.

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 this 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.

Oracle checker. After each event is generated by the test executor, the oracle checker dumps the GUI layouts from devices A and B, and 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.

Specifically, to improve the reproducibility, we have made several decisions in tool implementation: (1) waiting each GUI event to take effect before executing the next one to reduce the risk of flaky tests, (2) limiting the length of each test case (100 events in our case), (3) clearing the app data and resetting the mutated settings between each test case, and (4) recording all the executed events, the screenshots and the mutated setting options for bug reproduction. Specifically, SETDROID removes a bug if it cannot be faithfully reproduced before reporting. Therefore, all the defects reported by SETDROID are reproducible.

4.1.4 *Optimizations of* SETDROID

Compared to our prior work [24], we optimize SETDROID in terms of testing efficiency and precision.

Optimization I: Identify the relevant settings of an app. We add a generic optimization strategy in the setting change injector (module (b) in Fig. 6) to identify the relevant settings of an app (*i.e.*, which settings are used by the app). Specifically, we perform static analysis to determine which settings are relevant to the app. In particular, we analyze whether an app uses specific APIs (classes, methods, or variables) of each setting category in its code. This analysis is similar to that used by RQ1. In this way, SETDROID needs to apply only the relevant pairs of events for setting changes (see Table 5) to stress test an app, and thus substantially improves testing efficiency. This optimization strategy will not introduce false negatives because the setting defects (*e.g.*, network accessing issues) only appear when the app fails to properly adapt to the change of some specific setting (*e.g.*, network). If the app's functionality does not rely on the specific setting, it is safe to avoid changing the relevant setting (*e.g.*, closing/opening the network connection). In our case, our technical insight is that if an app does not use any API related to that setting, the app does not rely on that setting. In Section 5.3.2, we show the improvement of testing efficiency by applying this optimization strategy.

Optimization II: Reduce the false positive rate. In our prior work [24], the false positive rate of Rule II is high (535/662≈80.8%). We analyze the experimental results and find that *materialistic* and *RedReader* incur the majority of false positives (435/535≈81.3%) of Rule II. These two apps

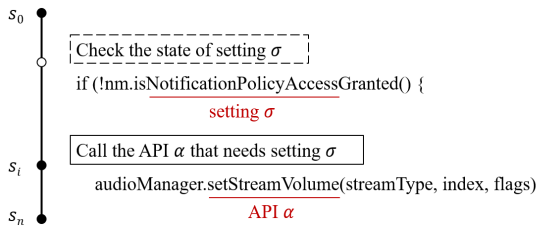


Fig. 7. Fault Pattern I.

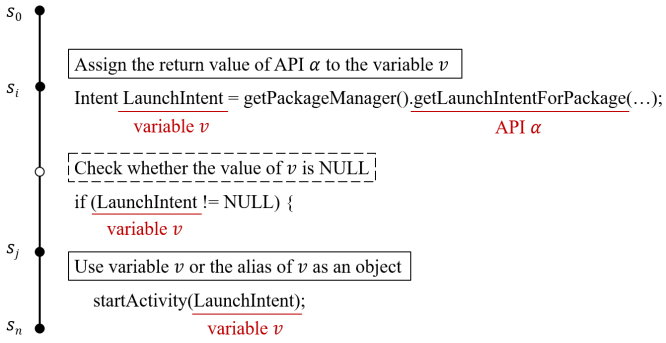


Fig. 8. Fault Pattern II.

are newsreaders in English. When SETDROID changes the default language, the texts of these newsreaders do not need to be 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 leading to a large number of false positives.

To this end, we add a generic optimization strategy in the bug report reducer (module (d) in Fig. 6) to reduce the false positives. Our key observation is that only local strings specified in an app should be translated, while non-local strings such as the texts in a news article loaded from a remote server should be escaped. We find that each app has a `strings.xml` file in its resource folder, and this file stores the app's local strings. Thus, the bug report reducer can automatically remove the false positives if the reported untranslated strings are not in `strings.xml`. This optimization strategy will not introduce false negatives. Because our key technical insight is that only local strings in the `strings.xml` file need to be translated. Deleting those bug reports related to non-local strings will not remove the valid setting defects. In Section 5.3.2, we will show the results by applying this optimization.

In addition, we find that some false positives are caused by the reserved keywords that do not need to be translated when the language is changed. However, the proportion of such false positives is small. Thus, we do not reduce them; otherwise, we have to use some ad-hoc strategies.

4.2 Static Analysis for Setting Defects

According to the results of our empirical study, we find that some setting defects can be detected at the code level. We summarize two major fault patterns and propose a static analysis tool named SETCHECKER to detect the setting defects.

4.2.1 Fault Pattern Analysis

We analyze 482 fixed bug reports in the dataset of our study. By analyzing these reports, we characterize these setting

```
1   private void goToSendFeedBack() {
2       LocationManager mgr = cxt.getSystemService(Context.LOCATION_SERVICE);
3       List<String> providers = mgr.getProviders(true);
4       Location last = null;
5       for (Iterator<String> i = providers.iterator(); i.hasNext(); ) {
6         Location loc = mgr.getLastKnownLocation(i.next());
7         if (LocationUtil.compareLocationsByTime(loc, last)) {
8           last = loc;}}
9 -     ReportActivity.start(this, loc.getLatitude(), loc.getLongitude(), mGoogleApiClient);
10 +    if (loc != null) {
11 +      ReportActivity.start(this, loc.getLatitude(), loc.getLongitude(), mGoogleApiClient);
12 +    } else {
13 +      ReportActivity.start(this, mGoogleApiClient); }}
```

Fig. 9. Example of Fault Pattern II.

defects as two fault patterns:

- ***Fault Pattern I.*** As shown in Fig. 7, if there exists some program trace S=$[s_0, ..., s_i, ..., s_n]$ in the the control-flow graph (CFG) of an app, where s_i is a statement denoting an invocation of API α. It is known that all settings in the device have two states: *on* and *off*. The first fault pattern is that API α (*e.g.*, `AudioManager#setStreamVolume()` in Fig. 7) can be executed only if the state of setting σ (permission `ACCESS_NOTIFICATION_POLICY` in this case) is *on*; otherwise, an exception will be thrown. However, if none of the statements between s_0 and s_i checks whether the state of setting σ is *on* (*e.g.*, `isNotificationPolicyAccessGranted()` in Fig. 7 is an API that checks whether the state of permission `ACCESS_NOTIFICATION_POLICY` is *on*), the app has a suspicious setting defect.
  Example of Fault Pattern I. Fig. 5 in Section 3.2.5 shows such a defect. Before calling `AudioManager#setStreamVolume()` (Line 7), the `ACCESS_NOTIFICATION_POLICY` permission is not checked (Line 3). As since Android's Nougat version, if an app is in the DND mode, the app needs to get the `ACCESS_NOTIFICATION_POLICY` permission before calling `setStreamVolume()`. If an app never checks this permission before calling `setStreamVolume()`, `SecurityException` will be thrown during the running of the app.
- ***Fault Pattern II.*** As shown in Fig. 8, if there exists some program trace S=$[s_0, ..., s_i, ..., s_j, ..., s_n]$ in the control-flow graph (CFG) of an app, where s_i is a statement assigning the return value of an API α to the variable v, and s_j is a statement using the variable v (or the alias of variable v). The second fault pattern is that s_j uses v (or the alias of variable v), but none of the statements between s_i and s_j checks whether the value of v (or the alias of variable v) is valid or not. In the illustrative example of Fig. 8, the API α is `LocationManager #getLastKnownLocation()`, and the variable *loc* accepting the return value of API α, which is later used by s_j. If v is not checked (*i.e.*, whether v is a `NULL` value), a suspicious setting defect is found in the app.
  Example of Fault Pattern II. Fig. 9 shows such a defect of *OneBusAway* [46], which is an app serving real-time transit information. The return value of a setting-related API is not checked before being used. Specifically, `LocationManager #getLastKnownLocation()` (Line

6) returns the current geographic location via GPS. When the location setting is closed or the GPS signals is not stable, `NULL` value will be returned. Therefore, a non-NULL check (Line 10) should be performed before using the return value of this API (Line 11).

Note that fault patterns I and II aim at finding different types of setting defects, which require different static analysis strategies.

- Fault pattern I aims at those setting defects in which the states of related settings are *deterministic* (that is, the states of these settings only change when an app user turns on/off these settings). For example, the states of permission, sound, and battery settings are deterministic. Thus, checking the states of these settings before the API invocation is safe. In Section 4.2.2 (Algorithm 1), we adopt the backward control-flow analysis to detect such defects.
- Fault pattern II aims at those setting defects in which the states of related settings are *non-deterministic* (that is, the states of these settings may be changed by an app user or affected by the external environment). For example, the states of network and location settings are non-deterministic. Take the location setting as a concrete example, even if the state of a device's location setting (*e.g.*, the GPS) is *on*, the return value of `LocationManager#getLastKnownLocation()` (*i.e.*, the location information obtained from the GPS) may still be invalid (*e.g.*, `NULL`) if the GPS signal is weak or lost due to the external environment. Thus, for these settings, only checking their states before calling the relevant API is unsafe, and we should additionally check their return values before used by the API. In Section 4.2.2 (Algorithm 2), we adopt the forward data-flow analysis to detect such defects.

4.2.2 Static Defect Detection

To detect the setting defects *w.r.t.* the preceding two fault patterns, we propose a static analysis tool named SETCHECKER. We implement SETCHECKER based on SOOT [47]. SOOT is an analysis and transformation framework, which provides some analysis functionalities including call graph construction, def/use chain analysis, and data-flow analysis, *etc*. SETCHECKER combines control-flow and data-flow analyses to detect setting defects.

(1) Algorithm 1. The algorithm shown in Algorithm 1 detects defects *w.r.t.* fault pattern I. First, we summarize a mapping list based on our empirical study. If there exist two APIs, the first of which requires the state of setting σ to be on, and the second of which can check the state of setting σ, we store them as a tuple $p < requireSettingAPI, checkSettingAPI >$ and add them to the mapping list. The algorithm takes the APK file of an Android app and a tuple $p < requireSettingAPI, checkSettingAPI >$ in the mapping list as input. After initialization (Line 1), the algorithm first gets all methods (Line 2) and the method call graph (Line 3) of the given Android app, then traverses every method, and looks for all methods using the API $p.requireSettingAPI$ (Lines 4-5). If a method m is found to use API $p.requireSettingAPI$, the algorithm searches the method call graph and gets all

Algorithm 1: Finding defects *w.r.t.* fault pattern I

inputs: APK: the APK file of an Android app,
$p < requireSettingAPI, checkSettingAPI >$: a tuple p, in which the first element represents an API that requires the state of setting σ to be on, and the second element represents another API that can check the state of setting σ
output: $defectSet$: the set of defects matched by the fault pattern, and each defect is represented as a trace of method invocations.

```
1  defectSet ← ∅;
2  allMethods ← getAllMethods(APK);
3  cg ← buildCallGraph(APK);
4  foreach method m ∈ allMethods do
5      if p.requireSettingAPI in m then
6          methodsTraces ← getAcyclicCallers(m, cg);
7          foreach methodsTrace t ∈ methodsTraces do
8              isCheckSettingAPIExist ← false;
9              target ← p.requireSettingAPI;
10             foreach method caller ∈ t do
11                 dominators ←
                     caller.findDominatorsOf(target);
12                 foreach dominator ∈ dominators do
13                     if dominator uses p.checkSettingAPI then
14                         isCheckSettingAPIExist ← true;
15                         break;
16                 target ← caller
17             if isCheckSettingAPIExist == false then
18                 defectSet ← defectSet ∪ {t}
19 return defectSet;
```

method call traces that can reach m as a method list named $methodsTraces$ (Line 6). For each method call trace $methodsTrace$ in the $methodsTraces$ (Line 7), a variable $isCheckSettingAPIExist$ is used to represent whether the setting check API $p.checkSettingAPI$ is found in this $methodsTrace$. The initial value of $isCheckSettingAPIExist$ is $false$ (Line 8). For each method list $methodsTrace$, the $i+1^{th}$ element of $methodsTrace$ is the caller of the i^{th} element (where i is the index of list $methodsTrace, i >= 0$). Specifically, the first element of $methodsTrace$, *i.e.*, $methodsTrace[0]$, is method m. The algorithm first sets $target$ to API $p.requireSettingAPI$ (Line 9), sets $caller$ to method m (Line 10), and then starts the loop. In each loop iteration, the algorithm checks whether the $caller$ calls the setting check API $p.checkSettingAPI$ before calling the $target$ by performing dominator analysis [48] (Lines 11-13). If there exists a dominator that uses $p.checkSettingAPI$, the algorithm sets the value of $isCheckSettingAPIExist$ to true (Line 14) and exits the loop (Line 15). Otherwise, it sets $target$ to the current $caller$ (Line 16), updates $caller$ to the caller of the current $caller$, and enters the next loop iteration (Line 10). In the end, if $isCheckSettingAPIExist$ is still $false$ (Line 17), then on this trace, the state of setting σ has not been checked from the beginning of the entry method to the point where $p.settingDependentAPI$ is called. Therefore, a suspicious setting defect is found, and the method call trace is added to the set $defectSet$ (Line 18).

(2) Algorithm 2. The algorithm shown in Algorithm 2 detects defects *w.r.t.* fault pattern II. We implement this algorithm based on the forward data-flow analysis framework provided by SOOT. Algorithm 2 takes the APK file of an Android app and an API $settingDependentAPI$ as input. Whether the return value of the API $settingDependentAPI$ is NULL depends on the state of setting σ. After initialization (Line 2), Algorithm 2 first gets all methods (Line 3) of the given Android app, then traverses every method,

Algorithm 2: Finding defects *w.r.t.* fault pattern II

```
inputs: APK: the APK file of an Android app,
        settingDependentAPI: an API, whether the return value of
        this API is a NULL value depends on the state of setting σ
output: defectSet: the set of defects w.r.t. fault pattern II, and each
        defect is represented as a trace of methods
1  Function Main:
2      defectSet ← ∅
3      allMethods ← getAllMethods(APK)
4      foreach method m ∈ allMethods do
5          if m use settingDependentAPI then
6              defectSet ← defectSet ∪ findDefect(m, ∅, ∅)
7  Function findDefect(m, initSet, methodsTrace):
8      nodes ←
         nullnessAnalysis(m, settingDependentAPI, initSet)
9      defectSet ← ∅
10     methodsTrace.append(m)
11     foreach node n ∈ nodes do
12         defectFlag ← false
13         foreach value v ∈ n.inSet do
14             if v is used in n then
15                 defectSet ← defectSet ∪ {methodsTrace}
16                 defectFlag ← true
17                 break
18         if n is a method call site and defectFlag == false then
19             m ← the method called by n
20             foreach value v ∈ n.inSet do
21                 if v is used as a parameter of m then
22                     initSet ← ∅
23                     defectSet ← defectSet ∪
                         findDefect(m, initSet, methodsTrace)
24                     break
25     return defectSet
```

and finds all methods using the API $settingDependentAPI$ (Lines 4-5). If a method m uses API $settingDependentAPI$, the algorithm calls function $findDefect$ to detect whether there is a setting defect in method m (Line 6). Function $findDefect$ first performs the branched nullness analysis on target method m to confirm which statements are at risk of using NULL values (Line 8). We implement the function $nullnessAnalysis$ in reference to the classic reaching-definition data-flow framework [48]. Let s be a statement. Let $gen(s)$ and $kill(s)$ be the dataflow facts generated or killed by s. Let $pred(s)$ be the predecessor statements of s. The data-flow equations used for a given basic block s in reaching definitions are

$$in\,(s) = \bigcup_{p \in pred[s]} out\,(p) \tag{2}$$

$$out(s) = gen\,(s) \cup (in\,(s) - kill\,(s)) \tag{3}$$

Based on the generic equations, we customize the gen and $kill$ sets. If s is an assignment statement in which any variable in the set $in(s)$ or the return value of API $settingDependentAPI$ is assigned to a variable v, variable v will be added to the set $gen(s)$. If s is a conditional statement that performs the non-NULL check at any variable v in the set $out(s)$, variable v and its aliases will be added to the set $kill(s)$. It is worth noting that the algorithm computes out sets from in sets sensitive to branches as we implement a branched forward flow analysis.

The return value of $nullnessAnalysis$ is a list of nodes, each of which represents a statement s with two sets $in(s)$ and $out(s)$ (denoted as $n.inSet$ and $n.outSet$, respectively). These two sets describe the variables that may have NULL value before and after the execution of this statement. After executing $nullnessAnalysis$ (Line 8), the next step (implemented in function $findDefect$ shown in Line 7) is to find which statements use variables that may have NULL values. Since $findDefect$ is a recursive function, a variable $methodsTrace$ is used to record how many methods of the app have gone through (Line 10). For each statement in the target method m (Line 11), if any variable of the $inSet$ is used in this statement, a setting defect is found, and the $methodsTrace$ is added to the set of recorded defects $defectSet$ (Lines 13-17). If the statement calls another method of the app, and the variable v in the $n.inSet$ is used as the parameter of the call, then $findDefect$ will enter the called method to find the suspicious setting defects, and set the $initSet$ to variable v. If setting defects are found in the called method, they will be merged into the $defectSet$ (Lines 18-24).

5 Evaluation

We evaluate the effectiveness of our two bug finding techniques and the usefulness of our insights gained from the study by answering RQ5, RQ6, and RQ7:

- **RQ5**: How effectively can SetDroid and SetChecker find previously-unknown setting defects in real-world Android apps (including both open-source and industrial apps)?
- **RQ6**: Do our insights gained from the study help SetDroid and SetChecker find setting defects that cannot be found by prior tools?
- **RQ7**: What are the differences between SetDroid and SetChecker in finding setting defects? Can they complement each other?

5.1 Evaluation Setup of RQ5

We consider the 30 apps from prior work [49] as the evaluation subjects, because most of these apps are selected from the popular open-source apps on GitHub [50]. Considering our experiment requires developers' feedback, we focus on those actively-maintained apps. Thus, our evaluation subjects include 26 apps as the other 4 apps are obsoleted.

5.1.1 *Setup of* SetDroid

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 5) 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 14 CPU days). Here, the whole testing time is computed assuming the optimization strategy in Section 4.1.4 is not enabled. In Section 5.3.2, we discuss the reduction of testing time when the strategy is enabled. Specifically, 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 the consequence 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 Tencent, ByteDance and Alibaba, *i.e.*, WeChat [17], QQMail [18], TikTok [19], CapCut [20], and AlipayHK [21], all of which each have billions of monthly active users. We allocate a 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.

5.1.2 Setup of SETCHECKER

We run SETCHECKER on the same Ubuntu 18.04 machine in Section 5.1.1 to analyze each app. For each suspicious defect reported by SETCHECKER, we manually verify whether it is a true positive or a false positive by inspecting the suspicious faulty code to infer a failure-triggering test on the GUI pages. If the test can witness the suspicious defect, we count it as a true positive. Specifically, we constrain our manual inspection time on each suspicious defect as 10 minutes. If we cannot verify a suspicious defect within 10 minutes, we give up this suspicious defect and do not report it to the developers (*i.e.*, we count it as neither a true positive or a false positive). We verify the reported suspicious defects on both an Android emulator (Android 8.0, Pixel XL) and a Galaxy A6s mobile phone (Android 8.1.0).

5.2 Evaluation Setup of RQ6.

5.2.1 Setup of dynamic testing tools

Existing fully automated dynamic testing tools for Android can be divided into two categories. The first category includes *generic* testing tools [51], [52], [53], [54], [55], [56], [57], [58]. These tools focus on only the app under test and do not interact with the system app `Settings` to change settings which were confirmed by a recent study [11]. The second category includes tools for detecting specific failures [49], [6], [59], [60]. To our knowledge, PREFEST [49] and PATDROID [6] are the two relevant dynamic testing tools for 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-crashing 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 [51], which randomly explores the app under test without explicitly changing settings. This baseline follows the same testing strategy of Monkey. We do not choose to directly run Monkey. Because Baseline A can generate tests based on widgets (which are intuitive), while Monkey generates pixel-based events (which are hard to understand). In addition, Baseline A can help reproduce setting defects much more easily. In practice, Baseline A just runs Module (a) in Fig. 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 5, including all the settings in PREFEST and PATDROID. In practice, Baseline B just runs Modules (a) and (b) in Fig. 6.
- We also run PREFEST and PATDROID for direct comparison with SETDROID.

Note that we allocate 13 hours (the same testing time used by SETDROID) for each setting change of 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. To eliminate randomness, we test each app three times and take the average values as the final results.

5.2.2 Setup of static analysis tools

Existing static analysis tools for Android can be divided into two categories. One category is generic static analysis tools that detect various types of code faults, while the other category focuses on specific types of code faults. For the first category, we choose Android LINT [61] to compare with SETCHECKER because LINT is the most popular Android static analysis tool. It can check an Android project with source code for diverse types of issues for correctness, security, performance, usability, accessibility, and internationalization. For the second category, to our best knowledge, there is no static analysis tool for detecting all types of setting defects. We note that two tools REVDROID [62] and ARPDROID [63] can find defects caused by runtime permissions. However, we find that ARPDROID reports a lot of false alarms. The high number of false alarms prevents us from identifying real defects within reasonable manual efforts. So we decide to compare REVDROID with only SETCHECKER. REVDROID is a static analysis tool to detect the potential defects caused by permission revocation (similar to the setting changes in our context). We run LINT [61] and REVDROID on the same Ubuntu 18.04 machine and manually verify and count the number of found setting defects for each tool.

5.3 Results of RQ5

5.3.1 Effectiveness

Table 6 shows the setting defects found by SETDROID and SETCHECKER, respectively. 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 information of detected defects, which contains issue ID, issue state (fixed, confirmed, under discussion with developers, or waiting for the reply), related setting, and consequence. As shown in Table 6, out of the 26 apps, SETDROID and SETCHECKER detect 48 unique and previously-unknown setting defects in 26 apps. so far, 35 have been confirmed and 21 have been fixed. The results demonstrate the effectiveness of SETDROID and SETCHECKER. 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 and SETCHECKER can find defects cared by developers.

TABLE 6
List of the 48 setting defects found by SETDROID and SETCHECKER and the detailed statistics of the evaluation results.

App ID	App name	#Downloads	#Stars	Issue ID	Issue state	Cause setting	Consequence	#FP^{SI}	#TP^{SI}	#FP^{SII}	#TP^{SII}	#Strategy^{S}	#Reported^{P}	#Verified^{P}	Time^{P}
1	APhotoManager	-	187	#175	Confirmed	Permission	Crash	0	3	0 → 0	0→0	13 → 4	0	0	22s
2	A2DP Volume	100K-500K	81	#295	Fixed	Display	Crash	0	10	0 → 0	0→0	13 → 5	8	8	21s
				#291	Fixed	Display	Data lost								
				#290	Fixed	Display	Crash								
				#289	Fixed	Display & Permission	Data lost								
				#294	Confirmed	Developer	Crash								
				#301	Confirmed	Permission	Crash								
3	Always On	10M-50M	129	#2476	Confirmed	Language	Disrespect of Settings	3	0	3 → 3	6→6	13 → 8	5	2	26s
				#2475	Confirmed	Language	Incomplete translation(5)								
4	AnkiDroid	5M-10M	4.3K	#5407	Fixed	Permission	Stuck	0	4	7 → 1	0→0	13 → 11	2	1	38s
5	AntennaPod	500K-1M	3.9K	#4227	Fixed	Network	Lack of refresh	1	2	7 → 2	1→1	13 → 11	0	0	4s
6	Commons	50K-100K	716	#3134	Discussion	Location	Infinite loading	0	9	30 → 3	0→0	13 → 10	6	2	45s
				#3906	Confirmed	Permission	Crash								
				#4594	Confirmed	Permission	Crash								
				#4498	Confirmed	Display	Crash								
7	ConnectBot	1M-5M	1.7K					1	8	8 → 2	0→0	13 → 8	0	0	4s
8	FillUp	100K-500K	31					3	0	0 → 0	0→0	13 → 2	0	0	13s
9	Forecastie	10K-50K	692	#358	Fixed	Permission	Lack of prompt	1	3	1 → 1	5→5	13 → 8	0	0	16s
				#504	Fixed	Language	Incomplete translation(5)								
				#505	Confirmed	Display	Data lost								
10	Good Weather	5K-10K	204	#61	Waiting	Network	Infinite loading	0	6	4 → 4	51→51	13 → 8	10	10	14s
				#55	Waiting	Location	Lack of prompt								
				#62	Waiting	Language	Language confusion								
11	Materialistic	100K-500K	2.2K	#1429	Waiting	Network	Lack of refresh	1	8	144 → 1	1→1	13 → 7	10	1	13s
12	Notepad	100K-500K	197					0	3	4 → 1	1→1	13 → 4	0	0	54s
13	Omni Notes	100K-500K	2.3K	#695	Fixed	Permission	Lack of prompt	0	9	3 → 3	3→3	13 → 7	2	1	31s
				#764	Fixed	Location	Error prompt								
				#776	Fixed	Language	Disrespect of Settings								
				#775	Fixed	Language	Incomplete translation(2)								
14	Opensudoku	10K-50K	279	#93	Confirmed	Language	Incomplete translation(7)	1	2	1 → 1	7→7	13 → 1	0	0	52s
15	RedReader	100K-500K	1.2K	#749	Discussion	Network	Infinite loading	0	4	291 → 1	30→30	13 → 8	0	0	38s
				#783	Confirmed	Language	Incomplete translation(23)								
16	Timber	100K-500K	6.6K	#454	Confirmed	Display	Data lost	0	4	0 → 0	9→9	13 → 8	0	0	35s
				#458	Waiting	Permission	Crash								
				#459	Waiting	Language	Incomplete translation(9)								
17	Vanilla Music	500K-1M	862	#1048	Waiting	Display	Crash	1	7	0 → 0	0→0	13 → 5	0	0	22s
18	Wikipedia	50M-100M	1.5K					0	6	4 → 0	0→0	13 → 7	7	1	54s
19	OpenBikeSharing	1K-5K	63	#55	Confirmed	Display	Function failure	3	8	0 → 0	0→0	13 → 4	3	3	13s
				#59	Confirmed	Permission	Error prompt								
20	Suntimes	-	162	#420	Fixed	Location	Infinite loading	1	2	0 → 0	0→0	13 → 10	6	1	40s
21	RadioBeacon	-	51	#234	Confirmed	Network	Stuck	3	2	10 → 4	1→1	13 → 10	13	11	19s
				#249	Confirmed	Permission	Crash								
22	RunnerUp	50K-100K	583	#923	Fixed	Permission	Lack of prompt	0	1	0 → 0	0→0	13 → 8	4	4	36s
				#1082	Waiting	Network	Lack of prompt								
23	Amaze	1M-5M	3.6K	#1885	Fixed	Display & Permission	Black screen	4	21	18 → 1	0→0	13 → 8	0	0	35s
				#1964	Fixed	Display & Permission	Data lost								
				#1920	Fixed	Network	Lack of prompt								
				#1919	Fixed	Display & Permission	Crash								
				#1965	Fixed	Permission	Crash								
24	Kiss	1M-5M	2K	#1835	Waiting	Location	Wrong Prompt	0	0	0 → 0	0→0	13 → 3	4	1	18s
25	Habits	1M-5M	4.5K	#599	Fixed	Display	Data lost	2	2	0 → 0	1→1	13 → 2	0	0	28s
				#620	Fixed	Language	Incomplete translation(2)								
26	Signal	100M-500M	22.3K					0	0	0 → 0	0→0	13 → 11	0	0	59S

5.3.2 Usability

Table 6 shows the detailed evaluation results of SETDROID and SETCHECKER on each app. Specifically, Columns 9-12, respectively, show the results of SETDROID, including the FPs of oracle checking rule I (#FP^{SI}), the TPs of oracle checking rule I (#TP^{SI}), the FPs of oracle checking rule II (#FP^{SII}, the numbers *preceding* and *following* the symbol "→", respectively, denote the FPs of SETDROID *before* and *after* using the FP reduction strategy (see Optimization II in Section 4.1.4)) and the TPs of oracle checking rule II (#TP^{SII}, the numbers *preceding* and *following* the symbol "→", respectively, denote the TPs of SETDROID *before* and *after* using the FP reduction strategy).

During testing, SETDROID reports 293 defects in total. Among them, 149 defects are reported by oracle checking rule I, 124 of which are TPs (124/149≈83.2%); the remaining 144 defects are reported by oracle checking rule II, 116 of which are TPs (116/144≈80.6%). 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 a 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, app *Always On* pops up animation on top of the screen, leading to some GUI inconsistencies between the two devices.
- *FP^{II} of oracle checking rule II*. In our prior work, the false-positive rate of Rule II is high (535/662≈80.8%). To reduce the false positive rate, we analyze the experimental results and add a generic FP reduction strategy for SETDROID (see Section 4.1.4) in this journal version. The current result clearly shows that the strategy is useful as it substantially reduces the false positive rate of Rule II from 80.8% to 19.4% (≈28/144). And no true positive is removed after the FP reduction strategy is applied. The remaining FPs are caused by some reserved keywords that do not need to be translated after the language is changed.

On the other hand, in our prior work, we have to run 13 hours for each pair of events for setting changes in Table 5 for one app. To improve testing efficiency, we add a generic optimization strategy for SETDROID (see Section 4.1.4) in this journal version, i.e., we perform static analysis to determine which settings are relevant to the app and inject only the relevant pairs of events for setting changes in Table 5. In Table 6's Column 13 (*i.e.*, (#Strategy^{S})), the numbers *preceding* and *following* the symbol "→", respectively, denote the numbers of pairs of events for setting changes needed to run on each app *before* and *after* the optimization strategy (see Optimization I in Section 4.1.4) is applied (note that our prior work applies all the 13 pairs of events for setting changes on each app). We can see that this optimization strategy reduces 2∼11 irrelevant pairs of events for setting changes, reducing 15∼85% testing time. The results clearly shows that the strategy is useful. Moreover, to investigate whether this optimization strategy will bring false negatives, we manually confirmed that for all the benchmark apps, *none* of the settings (which induce the setting defects found by SETDROID before optimization) are filtered out by this optimization strategy. The result shows that this strategy does not incur any false negatives in our experiment.

As for SETCHECKER, 78 suspicious setting defects are found in 26 apps. In Table 6, Columns 14-15 respectively

TABLE 7
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	Location	Functionality failure
16	AlipayHK	Language	Functionality failure
17	AlipayHK	Location	Functionality failure

give the number of setting defects reported by SETCHECKER (#Reported^P) and the number of verified setting defects (#Verified^P). Specifically, out of 78 defects, 46 are successfully verified with real tests, achieving the hit rate of 59.0% (≈46/78). For each of the remaining 32 defects, there is one of three main reasons for failing to verify. (1) The mapping between the source code and the corresponding GUI widgets are hard to be set up. As a result, we sometimes cannot know which UI widgets should be executed to reach the code of interest. (2) The preconditions required to manifest the defect are difficult to be satisfied. As a result, even if the faulty code is successfully reached, the defect still cannot be triggered. (3) The found defect is located in a piece of dead code, which could not be triggered in reality. In addition, Column 16 (Time^P) in Table 6 reports the static analysis time cost on each app. We can see that the static analysis on these open-source apps is efficient.

5.3.3 Diversity of found defects

From Table 6, we can see that the setting defects found by SETDROID and SETCHECKER are diverse: the apps are affected by different settings with different consequences.

In terms of root causes, we inspect all 20 fixed defects and find that most of them are due to the lack of setting checks. For example, *RadioBeacon* will stay in the infinite loading status when the network is disconnected during uploading, and cannot recover after the network is connected. Some defects are caused by incorrect callback implementations (*e.g.*, *AnkiDroid* has one defect that fails to properly handle permission callbacks), while some defects are caused by mutual influence between settings (*e.g.*, as some apps may use both GPS and the network for positioning, network fluctuations may affect the positioning function. *Suntimes* has a setting defect caused by calling the location obtained when the network is lost.) On the other hand, most of the language defects are caused by the incomplete translation.

These setting defects also lead to different consequences in addition to crashes. For example, some apps lack necessary prompts or give wrong prompts when their functions fail. When the device-only mode is turned on, *Omni Notes* cannot insert the current location into the notes and prompts the user with a wrong, confusing message "location not found". As an example of disrespect of settings, when the users 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 rolls back to the default language.

5.3.4 Practicality

As shown in Table 7, SETDROID detects 17 unique and previously-unknown setting defects in 5 commercial apps, all of which have been confirmed and fixed by Tencent, ByteDance, and Alibaba. Note that we report only those defects that we believe to be true positives (TP) to the app vendors, and the TP rates are high (80.8% on average across the five apps). Table 7 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. Afterwards, ByteDance collaborates with us and deploys SETDROID to stress test TikTok, one of its major app products. Within a two-month testing campaign, SETDROID successfully finds 53 setting defects in TikTok. So far, 48 of them have been confirmed, and 20 have already been fixed.

We also apply SETCHECKER on the latest version of TikTok at the time of study. Due to the large code base, it takes SETCHECKER about 17 hours to scan TikTok for setting defects. Finally, SETCHECKER successfully finds 22 defects in TikTok. 14 of them are due to the lack of permission checks, and the remaining 8 are related to network settings. So far, 11 defects have been fixed.

We find that SETDROID and SETCHECKER only have one common setting defect. We inspected the testing results and found two major reasons. One reason is that TikTok has many complex functionalities, and SETDROID failed to cover the functionalities where the defects reside due to its random seed tests. On the other hand, SETCHECKER can scan the whole code base. Another reason is that most of the fault patterns implemented in SETCHECKER complements the ability of SETDROID. We will discuss the differences between these two tools in detail in Section 5.5.

> **Answer to RQ5**: SETDROID and SETCHECKER are effective and practical in finding setting defects for real-world apps. The found defects are diverse in terms of root causes and consequences and are of developers' concern. These two tools also show reasonable usability as their false positive rates are low. Moreover, SETDROID's two optimizations did not incur false negatives on all the tested apps, and significantly improved testing precision and efficiency.

5.4 Results of RQ6

Table 8 shows the comparison results of SETDROID and other dynamic testing techniques and the baselines. Row "#Defects" denotes the number of unique setting defects. We can see that SETDROID can detect more crashing and non-crashing setting *unique* defects than the other techniques. 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

TABLE 8
Comparison of SETDROID and other dynamic testing techniques. C and NC represent crashing and non-crashing 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

TABLE 9
Comparison of SETCHECKER and other static analysis tools.

Setting	All setting		Permission	
Tool	LINT	SETCHECKER	REVDROID	SETCHECKER
#Defects	1	44	23	29

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 8). PREFEST does not detect any setting defect while PATDROID detects two crashes related to permissions. Note that the crashes detected by PATDROID and SETDROID do not overlap, likely caused by the randomness in test generation. In summary, (1) SETDROID detects 33 non-crashing 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 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.

Table 9 shows the comparison results of SETCHECKER, LINT, and REVDROID. Row "#Defects" denotes the number of setting defects reported by these static analysis tools after false positives are manually excluded. Initially, LINT finds 142 suspicious defects in 26 apps. But after manual inspection, we find only one of them is a real setting defect, which is due to the lack of permission checks. This result indicates that LINT lacks the fault patterns related to setting defects, and SETCHECKER can well complement LINT. On the other hand, REVDROID can detect only permission related defects. Initially, REVDROID reports 99 suspicious defects in 17 apps. We manually check these suspicious defects and confirm that 23 of them are true positives, and the remaining 76 are false positives. Note that all these 23 true positives are also found by SETCHECKER. We find two major reasons for explaining the false positives of REVDROID. (1) REVDROID fails to identify that some permissions are already checked before calling the permission-related APIs. For example, app *A2DP Volume* already checks `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` permissions before calling `LocationManager# removeUpdates()`. But REVDROID still reports it as an suspicious defect. We find 20 false positives are caused by such a factor. (2) REVDROID mistakenly reports defects related to non-dangerous permissions. In fact, non-dangerous permissions are automatically granted when an app is installed and cannot be revoked. So we do not need to check these non-dangerous permissions. 49 of 76 reported suspicious defects are such false positives.

Answer to RQ.6: Inspired by the insights of our study, SETDROID is designed to be able to detect non-crashing setting defects that cannot be detected by existing automated testing tools. SETCHECKER can detect more setting defects than prior static analysis tools with the help of the fault patterns distilled from our study and achieve higher accuracy.

5.5 Results of RQ7

Table 10 shows the differences between SETDROID and SETCHECKER in terms of the setting defects found in the 26 open-source apps. Specifically, SETDROID reports 293 defects, 240 of which are TPs, while SETCHECKER reports 78 defects, 46 of which are successfully verified with real tests. In total, 48 unique and previously-unknown setting defects are found, 11 of which are found by both tools. In detail, 31 setting defects are found by only SETDROID, and 6 defects are found by only SETCHECKER. Specifically, for the network, display location, and sound settings, SETDROID finds 11, 9, 5, and 0 defects, respectively, while SETCHECKER finds 8, 0, 8, and 12 defects, respectively. These results are consistent with our analysis in Section 3.4, i.e., dynamic analysis techniques (such as SETDROID) are better for finding the defects related to the network and display settings while static analysis techniques (such as SETCHECKER) are more suitable for finding those related to the location and sound settings. Indeed, these results show that SETDROID and SETCHECKER can complement each other in finding setting defects. We next give more detailed analysis on the effectiveness of these two tools.

Why are some setting defects missed by SETDROID? Through our analysis, we note three major reasons for explaining why SETDROID may miss some setting defects while SETCHECKER can still find them. *First*, some setting defects will not cause GUI inconsistency, so it is difficult for SETDROID to detect them. For example, SETCHECKER finds *OpenBikeSharing*'s issue #59 (affected by permission setting, the app cannot display the map of the user's location), while SETDROID misses it. The reason is that no matter whether *OpenBikeSharing* successfully locates the user's location, the dumped GUI layout of the map interface are consistent. As a result, SETDROID does not have the chance to manifest the GUI inconsistencies. *Second*, SETDROID's effectiveness is also limited by the coverage of random seed tests. For example, SETCHECKER finds *KISS*'s issue #1835 (a setting defect related to permission), while SETDROID misses it. In issue #1835, the app fails to prompt proper alerts if a user searches contacts when the "read contacts" permission is disabled. However, this defect can be triggered only when the searched text is a valid contact name. This condition is difficult to be satisfied by the random seeds of SETDROID. *Third*, some setting defects can be triggered on only specific

Android versions. Thus, they are easier to be found by static analysis such as SETCHECKER than dynamic analysis.

Why are some setting defects missed by SETCHECKER? Most of setting defects are non-crashing failures, and many of them are caused by application specific logic errors. Thus, summarizing the complete and accurate fault patterns at the code level to capture all setting defects is almost impossible. The fault patterns of some setting defects are ad-hoc from the perspective of root causes. For example, a user of *ownCloud* reports in issue #1498 that if the device is offline during the upload process, the upload of the app will be suspended immediately, and the user cannot resume the upload after the network is restored. The developer solves this defect by adding a button with the text "retry" and the response method of the "retry" button. From the code level, most apps just need to catch the exception and prevent the apps from crashing when offline. Not all network-related apps require a retry button to resume interrupted operations, so it is difficult to summarize a common fault pattern for this defect. Thus, we summarize only some common fault patterns to detect some categories of some common setting defects, and other setting defects cannot be caught.

Discussion. (1) **What are the ability boundaries of SETDROID and SETCHECKER?** As a static analysis tool, SETCHECKER can quickly find defects related to the supported settings As shown in Table 6, for the 26 open source apps we tested, it takes only 29 seconds on average to complete the analysis. But SETCHECKER has some limitations. First, the types of defects found by SETCHECKER are limited by the supported fault patterns. For example, in Section 5.5, there are some setting defects found by SETDROID which cannot be detected by SETCHECKER. Second, SETCHECKER requires the availability of the app source code (or at least the unobfuscated APK file). Third, it requires more effort to verify the setting defects reported by SETCHECKER by constructing the real tests. For SETDROID, it can detect various types of setting defects due to the generic oracle rules. SETDROID can also provide the real tests to reproduce the reported setting defects. But SETDROID also has some limitations. First, as we discussed in Section 5.5, SETDROID cannot detect the setting defects that will not lead to GUI inconsistencies (*e.g.*, volume and power consumption related issues). Second, SETDROID's effectiveness could be affected by the adequacy (*e.g.*, code coverage) of the seed tests used for mutation, and the Android versions used for testing (because some setting defects are compatibility issues).

(2) **How to apply SETDROID and SETCHECKER in practice?** In practice, both SETDROID and SETCHECKER can find some setting defects which are hard to be detected by the other, due to their respective technical limitations. Thus, these two tools are complementary in bug finding. From the tool users' perspective, they could apply the tools according to the actual scenarios. For example, if users cannot obtain the app source code (or at least the unobfuscated APK file), they could choose SETDROID because it is a black-box testing tool; in the scenario of in-house testing (when the app source code is available) with tight testing time, they could use SETCHECKER because it can quickly identify potential bugs without running the app. If the users are not subject to the preceding limitations, they could run both tools to find as many setting defects as possible. Note that in this case the running order of these two tools does not affect the bug finding results.

TABLE 10
Number of the setting defects found by our two tools.

Tool	#Reported	#Verified	#Unique Defects
SETDROID	293	240	42
SETCHECKER	78	46	17

Answer to RQ.7: In terms of the numbers of found setting defects, SETDROID is more effective than SETCHECKER. But SETCHECKER can complement SETDROID by finding more defects which are hard to be found by SETDROID.

6 DISCUSSION

6.1 Relationships between SETDROID's testing mechanism and the root causes of setting defects

We wish to discuss the relationships between SETDROID's testing mechanism and the root causes of setting defects. It can help readers understand why SETDROID can detect the various categories of setting defects discussed in our empirical study. In Section 3.2, we discuss six types of root causes of setting defects. Although the root causes are diverse, SETDROID's oracle checking rules are generic to capture the inconsistent app behaviors caused by setting defects when a given setting is changed and later properly restored, or the expected differences cannot be shown when the change is not restored. For example, Section 3.2.1 discusses the root cause of incorrect callback implementations. Take AnkiDroid's issue #4951 as an example (discussed in Section 3.2.1), AnkiDroid did not correctly implement the callback method when the permission setting is changed (*i.e.*, forgetting to redraw the menu bar after the permission setting changes). In this case, this setting defect can be captured by SETDROID because it checks the GUI consistency between the seed test and the mutant test (which revokes and later restores the permission).

6.2 Generality of Our Approach

Since our oracle checking rules are generic, adding other pairs of setting changes into SETDROID is feasible and only involves one-time effort. Similarly, for SETCHECKER, we can also detect other types of setting defects by adding new fault patterns. One interesting direction is to explore whether our techniques can be applied to finding app setting-related defects. According to our experience, it might be doable but could be ad-hoc to define oracle checking rules and fault patterns because different apps have different app settings and their expected behaviors are different and app-specific.

6.3 Threats to Validity

One 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, 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 [49]. Another 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 third 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. The last threat is that due to the fast evolution of Android systems and apps, the validity of the presented results (*e.g.*, the study findings, the performance of the proposed two tools) may be affected. To alleviate this threat, we studied all the reported issues of the subject apps which have already involved different Android system versions. Therefore, the classification and the findings we obtained could be still valid for future Android system versions. For the dynamic testing tool SETDROID, as long as the metamorphic relation is valid (*e.g.*, the app behaviors should keep consistent if a given setting is changed and later properly restored, or show expected differences if not restored), SETDROID should be still effective and applicable. For the static analysis tool SETCHECKER, although the APIs used by the fault patterns may change, its algorithmic detection strategy based on control- and data-flow analysis should be still valid, and only necessary update of APIs is needed. The main limitation of SETDROID is that it cannot ensure that all the target codes (or activities) are covered because it generates random seed tests. However, we believe SETDROID can be enhanced by existing powerful test generation techniques.

7 RELATED WORK

Configuration testing for traditional software. Prior work investigates misconfiguration defects for traditional software. Yin *et al.* [64] 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 [65], [66], [67] focus on effective configuration combination strategies for testing and show that simple algorithms such as *most-enabled-disabled* are the most effective. Efforts [68], [69], [70] 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 [59], [71], [72]. For example, Hu *et al.* [59] study and detect the WebView defects, while Fan *et al.* [71], [73] and Su *et al.* [74] study the framework-specific crash defects, and Kong *et al.* [75] locate framework-specific bugs which are not captured in the stack traces. But they do not cover setting defects addressed by our work. Some studies investigate Android configurations [76], [77], [78], 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. Some researchers study limited types of setting defects. Wang *et al.* [7] conduct a large-scale empirical study of runtime permission defects in the Android ecosystem.

Automated Android app GUI testing. A number of automated GUI testing techniques have been proposed [53], [56], [57], [58], [79], [80], using different approaches, such as symbolic execution [81], evolutionary algorithm [54], random [51] and model-based testing [82], [52], [55]. 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-crashing setting defects. Adamsen *et al.* [83] 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.* [6] propose PATDROID, which uses combinatorial testing to automatically detect permission defects. Similarly, Lu *et al.* [49] 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-crashing* setting defects, while PATDROID and PREFEST can detect only crash ones. Our evaluation in Section 5.4 also shows these differences. Riganelli *et al.* [60] 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.

Android app static analysis. There are various approaches to doing static analysis of Android apps differing in precision, runtime, scope, and focus [84], [85]. Some static analysis approaches focus on detecting defects caused by specific setting categories. Since the release of Android 6.0, researchers have proposed various approaches to help legacy apps automatically migrate to the runtime permission model. Dilhara *et al.* [63] present ARPDROID, an automated solution that detects and repairs incompatible permission uses, adapting the app under analysis to the new permission model. Fang *et al.* [62] build an automatic tool, REVDROID, to analyze the potential side effects of permission revocation. However, these approaches focus on only specific types of setting defects, while SETCHECKER targets more types of setting defects.

8 CONCLUSION

In this article, we have presented the first empirical analysis of setting defects in Android apps and have shown that most apps are affected by setting defects. We have identified five major root causes and four types of consequences of these defects, and we have also analyzed the correlation between setting categories and their root causes and consequences. Guided by our study findings, we have proposed a setting-wise metamorphic fuzzing tool named SETDROID and a fault-pattern-based static analysis tool named SETCHECKER to detect setting defects. SETDROID

and SETCHECKER find 123 previously-unknown setting defects from 26 open-source and 5 industrial apps. These defects have diverse root causes and consequences. We have also given an in-depth comparison between our proposed tools and prior tools, and shown the superiority of our bug finding techniques. We have open-sourced our dataset and tools to facilitate replication and future research at *https://github.com/setting-defect-fuzzing/home*.

REFERENCES

[1] W. Team, "WordPress," 2022, retrieved 2022-9 from https://github.com/wordpress-mobile/WordPress-Android.

[2] mzorz, "WordPress issue #6026," 2017, retrieved 2022-9 from https://github.com/wordpress-mobile/WordPress-Android/issues/6026.

[3] malinajirka, "WordPress issue #10096," 2019, retrieved 2022-9 from https://github.com/wordpress-mobile/WordPress-Android/issues/10096.

[4] N. Team, "NextCloud," 2022, retrieved 2022-9 from https://github.com/nextcloud/android.

[5] fpernice518, "NextCloud issue #2979," 2018, retrieved 2022-9 from https://github.com/nextcloud/android/issues/2979.

[6] A. Sadeghi, R. Jabbarvand, and S. Malek, "PATDroid: permission-aware gui testing of android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 220–232.

[7] Y. Wang, Y. Wang, S. Wang, Y. Liu, C. Xu, S.-C. Cheung, H. Yu, and Z.-l. Zhu, "Runtime permission issues in android apps: Taxonomy, practices, and ways forward," *IEEE Transactions on Software Engineering (TSE)*, 2022.

[8] D. Amalfitano, V. Riccio, A. C. Paiva, and A. R. Fasolino, "Why does the orientation change mess up my android application? from gui failures to code faults," in *Software Testing, Verification and Reliability (STVR)*, 2018, p. e1654.

[9] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino, "Automated functional testing of mobile applications: a systematic mapping study," in *Software Quality Journal (SQJ)*, 2019, pp. 149–201.

[10] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: a systematic literature review," in *IEEE Transactions on Reliability*, 2018, pp. 45–66.

[11] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 119–130.

[12] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, p. 429–440.

[13] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "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)*, 2018.

[14] M. L. Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: a new perspective for automated mobile app testing," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[15] A. Team, "Android Developers Documentation," 2022, retrieved 2022-9 from https://developer.android.com.

[16] ——, "Android Help," 2022, retrieved 2022-9 from https://support.google.com/android.

[17] W. Team, "WeChat," 2022, retrieved 2022-9 from https://www.wechat.com.

[18] Q. Team, "QQMail," 2022, retrieved 2022-9 from https://en.mail.qq.com.

[19] T. Team, "TikTok," 2022, retrieved 2022-9 from https://www.tiktok.com.

[20] C. Team, "CapCut," 2022, retrieved 2022-9 from https://lv.faceueditor.com.

[21] A. Team, "AlipayHK," 2022, retrieved 2022-9 from https://www.alipayhk.com.

[22] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A multi-agent model-based test generation system," in *IEEE/ACM 1st International Conference on Automation of Software Test (AST)*, 2020, pp. 93–96.

[23] F. Team. (2022) Fastbot(2.0). [Online]. Available: https://github.com/bytedance/Fastbot_Android

[24] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "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)*, 2021, p. 204–215.

[25] G. Team, "GitHub REST API," 2022, retrieved 2022-9 from https://docs.github.com/en/rest/.

[26] setting-defect fuzzing, "Dataset," 2022, retrieved 2022-9 from https://github.com/setting-defect-fuzzing/home.

[27] S. O. Team, "Stack Overflow," 2022, retrieved 2022-9 from https://stackoverflow.com.

[28] A. Team, "Ankidroid," 2022, retrieved 2022-9 from https://github.com/ankidroid/Anki-Android.

[29] S. Team, "Status," 2022, retrieved 2022-9 from https://github.com/status-im/status-react.

[30] F. Team, "Frost," 2022, retrieved 2022-9 from https://github.com/AllanWang/Frost-for-Facebook.

[31] C. Team, "Commons," 2022, retrieved 2022-9 from https://github.com/commons-app/apps-android-commons.

[32] ——, "Clover," 2022, retrieved 2022-9 from https://github.com/chandevel/Clover.

[33] O. Team, "Openlauncher," 2022, retrieved 2022-9 from https://github.com/OpenLauncherTeam/openlauncher.

[34] ——, "OpenFoodFacts," 2022, retrieved 2022-9 from https://github.com/openfoodfacts/openfoodfacts-androidapp.

[35] S. Team, "Signal," 2022, retrieved 2022-9 from https://github.com/signalapp/Signal-Android.

[36] K. Team, "K-9," 2022, retrieved 2022-9 from https://github.com/k9mail/k-9.

[37] S. Team, "Syncthing," 2022, retrieved 2022-9 from https://github.com/syncthing/syncthing-android.

[38] timotk, "Signal issue #6447," 2017, retrieved 2022-9 from https://github.com/signalapp/Signal-Android/issues/6447.

[39] haffenloher, "Signal issue #5353," 2017, retrieved 2022-9 from https://github.com/signalapp/Signal-Android/issues/5353.

[40] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," HKUST-CS98-01, Hong Kong University of Science and Technology, Tech. Rep., 1998. [Online]. Available: https://arxiv.org/abs/2002.12543

[41] A. Team, "Request App Permissions," 2022, retrieved 2022-9 from https://developer.android.com/training/permissions/requesting#perm-check.

[42] langid Team, "langid," 2022, retrieved 2022-9 from https://github.com/saffsd/langid.py.

[43] uiautomator2 Team, "uiautomator2," 2022, retrieved 2022-9 from https://github.com/openatx/uiautomator2.

[44] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[45] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "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)*, 2018, pp. 738–748.

[46] O. Team, "Onebusaway," 2022, retrieved 2022-9 from https://github.com/OneBusAway/onebusaway-android.

[47] S. Team, 2022, retrieved 2022-9 from http://soot-oss.github.io/soot/.

[48] A. V. Aho, *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[49] Y. Lu, M. Pan, J. Zhai, T. Zhang, and X. Li, "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)*, 2019, pp. 268–278.

[50] pcqpcq, "opensource-android-apps," 2022, retrieved 2022-9 from https://github.com/pcqpcq/open-source-android-apps/.

[51] M. Team, "Android Monkey," 2022, retrieved 2022-9 from https://developer.android.com/studio/test/monkey.

[52] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013, p. 641–660.

[53] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: automated system input generation for android applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 461–471.

[54] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 94–105.

[55] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 245–256.

[56] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 269–280. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00042

[57] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1–12.

[58] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, p. 153–164.

[59] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "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)*, 2018, pp. 702–713.

[60] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, "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)*, 2020, pp. 141–152.

[61] L. Team, 2022, retrieved 2022-9 from https://developer.android.com/studio/write/lint.

[62] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. S. Wang, Z. Qian, and H. Chen, "RevDroid: Code analysis of the side effects after dynamic permission revocation of android apps," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*, 2016, p. 747–758.

[63] M. Dilhara, H. Cai, and J. Jenkins, "Automated detection and repair of incompatible uses of runtime permissions in android apps," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, p. 67–71.

[64] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "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)*, 2011, pp. 159–172.

[65] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack," in *Empirical Software Engineering (EMSE)*, 2019, pp. 674–717.

[66] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 643–654.

[67] S. Souto, M. d'Amorim, and R. Gheyi, "Balancing soundness and efficiency for practical testing of configurable systems," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 632–642.

[68] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 619–634. [Online]. Available: https://dl.acm.org/doi/10.5555/3026877.3026925

[69] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, p. 445–456.

[70] S. Zhang and M. D. Ernst, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 152–163.

[71] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 408–419.

[72] A. Alshayban, I. Ahmed, and S. Malek, "Accessibility issues in android apps: state of affairs, sentiments, and ways forward," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1323–1334.

[73] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 486–497.

[74] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering (TSE)*, pp. 1115–1137, 2022.

[75] P. Kong, L. Li, J. Gao, T. Riom, Y. Zhao, T. F. Bissyandé, and J. Klein, "Anchor: locating android framework-specific crashing faults," *Automated Software Engineering*, vol. 28, no. 2, pp. 1–31, 2021.

[76] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in android testing: they matter," *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile)*, pp. 1–6, 2018.

[77] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 308–318.

[78] A. K. Jha, S. Lee, and W. J. Lee, "Developer mistakes in writing android manifests: an empirical study of configuration error," in *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 25–36.

[79] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving automated gui exploration of android apps via static dependency analysis," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 557–568.

[80] T. Su, "FSMdroid: guided GUI testing of android apps," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 689–691.

[81] J. Zhang, "Constraint solving and symbolic execution," in *Working conference on verified software: theories, tools, and experiments*. Springer, 2005, pp. 539–544.

[82] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 23–26.

[83] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 83–93.

[84] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[85] P. Kong, L. Li, J. Gao, T. F. Bissyandé, and J. Klein, "Mining android crash fixes in the absence of issue-and change-tracking systems," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 78–89.