Preference-Wise Testing of Android Apps via Test Amplification

MINXUE PAN and YIFEI LU, Nanjing University, China
YU PEI, The Hong Kong Polytechnic University, China
TIAN ZHANG and XUANDONG LI, Nanjing University, China

Preferences, the setting options provided by Android, are an essential part of Android apps. Preferences allow users to change app features and behaviors dynamically, and therefore their impacts need to be considered when testing the apps. Unfortunately, few test cases explicitly specify the assignments of valid values to the preferences, or *configurations*, under which they should be executed, and few existing mobile testing tools take the impact of preferences into account or provide help to testers in identifying and setting up the configurations for running the tests. This paper presents the Prefest approach to effective testing of Android apps with preferences. Given an Android app and a set of test cases for the app, Prefest amplifies the test cases with a small number of configurations to exercise more behaviors and detect more bugs that are related to preferences. In an experimental evaluation conducted on real-world Android apps, amplified test cases produced by Prefest from automatically generated test cases covered significantly more code of the apps and detected 7 real bugs, and the tool's test amplification time was at the same order of magnitude as the running time of the input test cases. Prefest's effectiveness and efficiency in amplifying programmer-written test cases was comparable with that in amplifying automatically generated test cases.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Android apps, Android testing, preference-wise testing

1 INTRODUCTION

The last decade has witnessed a rapid growth in Android apps, drawing attention from both academia and industry. To enable prompt response to user feedback and market changes, Android app developers have to work in short development cycles, causing a growing need for cost-effective testing approaches. For example, the automatic generation of test inputs [3, 7, 10, 21, 36] aiming at fully automatic testing of Android apps has attracted considerable attention in the past few years.

Most mobile apps allow some level of customization by having settings that enable users to tailor the apps' features and/or behaviors, and such settings are usually modeled using *preferences* [16] on the Android platform. Since an app may exhibit distinct behaviors when its preferences take different values, to thoroughly test the app becomes a more challenging task as it would require exercising the app not only with different user inputs but also under various assignments of valid values to preferences, or *configurations*. We call mobile testing with the impact of preferences taken into account *preference-wise* testing.

Despite the important role preferences play in apps, few mobile test cases explicitly specify the configurations under which they should be executed and few existing tools support effective preference-wise testing. On the one hand, some preferences cause minor or no differences to the

This research is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001) and the National Natural Science Foundation of China (Nos. 61972193 and 62032010). This work is also supported in part by the Hong Kong RGC General Research Fund (GRF) PolyU 152002/18E.

Authors' addresses: M. Pan (corresponding author), mxp@nju.edu.cn; Y. Lu, lyf@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology and the Software Institute, Nanjing University, 22 Hankou Road, Nanjing, Jiangsu Province, China, 210093; Y. Pei (corresponding author), csypei@comp.polyu.edu.hk, Department of Computing, The Hong Kong Polytechnic University, 11 Yuk Choi Road, Hung Hom, Kowloon, Hong Kong, China; T. Zhang, ztluck@nju.edu.cn; X. Li, lxd@nju.edu.cn, State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, 163 Xianlin Ave., Nanjing, Jiangsu Province, China, 210023.

appearance of apps, therefore they can be easily missed by black-box testing techniques that mainly derive the states of apps from their GUIs. On the other hand, it is challenging for existing white-box testing techniques to handle preferences effectively. Since preferences are often stored in the form of key-value pairs and apps usually access their preference values by the corresponding keys via method invocations, techniques like symbolic execution are needed to keep track of which and how preference values are actually utilized at runtime. Off-the-shelf symbolic execution techniques, however, can hardly work at the app level, partly because of the scalability issues it suffers from [24], and partly because of the event-driven nature as well as the underlying application development framework (ADF) [40] of the apps. As a result, although remarkable progress has been made in mobile testing recently, there is little tool support for helping testers effectively identify and set up differentiating configurations for Android apps in preference-wise testing.

In this paper, we propose the PREFEST approach to effective preference-wise testing for Android apps. PREFEST takes an Android app and a set of test cases for the app as the input and automatically amplifies each test case with a small group of configurations to exercise more behaviors and detect more bugs that are related to preferences. In practice, developers and testers are often not the same group of people, which makes the burden of identifying and applying appropriate configurations before running test cases even more overwhelming for testers. Being fully automatic, PREFEST can be of great help in easing the burden.

PREFEST is motivated by two key observations. The first observation is that each test case typically interacts with just a few preferences defined in the app. In view of that, PREFEST identifies for each test case a group of relevant preferences (i.e., preferences on which at least one branch condition executed by the test case has data-dependence) and focuses on altering the values of those relevant, instead of all, preferences when amplifying the test case. The second observation is that, even if we only consider the preferences relevant to a test case, exhaustively executing the test case under all possible value combinations of those preferences is often still prohibitively expensive and uneconomical, in terms of the testing time, the amount of code covered and the number of bugs detected. In light of this observation, the target mode of PREFEST drastically reduces the cost of preference-wise testing by aiming to execute each preference-related code branch at least once under some configuration, instead of exhausting all possible combinations of preference values.

We have implemented the PREFEST approach into a tool with the same name. To evaluate the effectiveness and efficiency of PREFEST, we applied PREFEST to amplify automatically generated test cases for 45 Android apps. The amplified test cases covered 9.3% and 15.3% more instructions and branches of the app code, respectively, and detected 7 real bugs. PREFEST's running time was at the same order of magnitude as the running time of its input test cases, suggesting the tool's performance was compatible with offline usage scenarios. Compared with test amplification based on another strategy where each test case is amplified with all configurations covering 2-way combinations of relevant preference values, preference-wise testing with PREFEST was able to achieve 70.2% and 73.7% of the additional instruction and branch coverage and detect the same bugs, while reducing the testing costs in the number of test executions involved and testing time required to 1.8% and 6.0%, respectively. PREFEST's effectiveness and efficiency in amplifying programmer-written test cases was comparable with that in amplifying automatically generated test cases.

This work significantly extends our previous work [35] in the following important aspects. First, we perform a study on the prevalence of preferences and preference related bugs in Android apps to strengthen the motivation for preference-wise testing. Second, we revise the description of the approach to include a formal presentation of the amplification process. Third, we extend the PREFEST tool to handle more types of preferences and make test case amplification with PREFEST more systematic. Fourth, we conduct larger scale experiments on Prefest and carry out more detailed analyses of the experimental results.

The main contributions this paper makes are as the following:

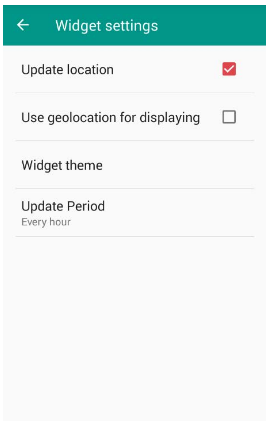
(1) We identify the problem of *preference-wise testing*, which is an important aspect of Android testing but has been largely overlooked by existing work in the area.
(2) We propose the Prefest approach to effective preference-wise testing for Android apps. The approach has been implemented into a tool with the same name.
(3) We conduct an experimental evaluation on 45 real-world apps to assess the effectiveness and efficiency of Prefest. Amplification helped the input test cases exercise more app behaviors and detect real bugs. The tool and the experimental data are available for download at https://github.com/Prefest2018/Prefest.

The rest of this paper is organized as follows. Section 2 illustrates how Prefest amplifies test cases from a user's perspective. Section 3 introduces background knowledge about preferences on the Android platform. Section 4 describes the study we conduct to assess the prevalence of preferences and preference-related bugs in Android apps. Section 5 explains in detail the techniques employed by Prefest and how Prefest works step by step. Section 6 reports on the experimental evaluation we carried out and presents the findings from the experiments. Section 7 reviews related work. Section 8 concludes the paper.

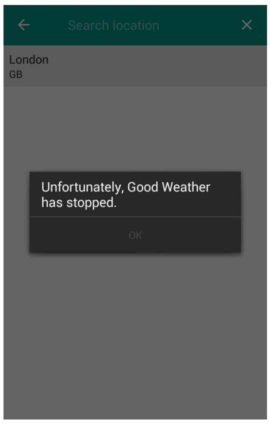
2 PREFEST IN ACTION

In this section, we use a real-world app called *good-weather* to demonstrate how Prefest can amplify test cases to detect a bug that causes the app to crash only under specific configurations.

The *good-weather* app is a weather application and it can be customized to show weather information for selected locations. The app also features a widget that can deck out the home screen with up-to-date weather reports. To allow users to customize how the widget looks and works, the app offers a list of preferences as shown in Figure 1a.


Widget settings
Update location ✓
Use geolocation for displaying ☐
Widget theme
Update Period
Every hour

(a) The preferences for the *good-weather* widget.



Unfortunately, Good Weather has stopped.

(b) The app crashes when changing the location.

Fig. 1. A preference-related crash in *good-weather*.

Some of these preferences affect the widget's behaviors. For instance, preference `Update location` determines whether or not to start a service to synchronize the location used by the widget with the one set in the *good-weather* app. Other preferences, e.g., preference `Widget theme`, affect the widget's appearance. Nevertheless, all features and functionalities of both the app and the widget should work as expected under all possible configurations. For example, a user should always be able to change the location used by the app for showing weather information, no matter whether the widget is enabled or how it is configured. This, however, is not the case with Version 4.4 of the app. Particularly, if preference `Update location` is set to `true` for the widget, a crash, as shown in Figure 1b, would occur when the user tries to change the location in the app. To reproduce this crash, a test case needs to set preference `Update location` to `true` first—the default value of the preference is `false`—and then try to change the location in the app. However, since there is no clear connection between the widget's preference and the behavior of the main app, it is less likely that the two actions are tested together, making the bug hard to discover.

The *good-weather* example suggests that, for the testing of an app to be systematic and thorough, it is critical to take the app's preferences into account. However, cost-effective preference-wise testing can be challenging. Since some errors only occur when executing certain operations under specific configurations and the connection between those operations and preferences may be buried in the code, to intentionally reveal, rather than accidentally bump into, preference related bugs, human testers often need to run test cases repeatedly under a large number of different configurations, which can be prohibitively expensive. The need for a cost-effective approach to preference-wise testing is pressing.

Taking the *good-weather* app and a test case for the app that changes the location as the input, Prefest automatically discovers that preference `Update location` is relevant to the test case and amplifies the test case with another configuration where the preference is set to `true`. Running the test case under the new configuration soon triggers the bug mentioned above.

In the following sections, we first explain how preferences are utilized in Android apps and why preference-wise testing is crucial for Android apps, and then describe in detail how Prefest amplifies test cases to facilitate effective and efficient preference-wise testing.

3 BACKGROUND

On Android, the recommended way to integrate user configurable settings into apps is to use the AndroidX Preference Library (APL), where class `Preference` is devised to model the key-values pairs of settings while activities and fragments showing lists of preferences are called *settings screens.*

To construct a settings screen based on the APL, a programmer needs to first define a hierarchy of preferences—either statically using a resource file in the XML format or programmatically by manipulating corresponding preference objects, then prepare the screen for displaying the preferences, and in the end link the preference hierarchy to the screen during the screen's initialization. In this work, we focus on apps that define preference hierarchies statically since using resource files is the most popular way to define preference hierarchies on Android. For example, the preference hierarchy of the *good-weather* app was defined using a resource file, and the snippet shown in Listing 1 is excerpted from the resource file. The snippet defines three preferences: 1) a `CheckboxPreference` that stores a boolean value and will be rendered as a checkbox widget; 2) a `ListPreference` that stores a string value and will be rendered as a list of all possible values for that preference; and 3) a `PreferenceScreen` that groups multiple preferences and will be associated with a specific settings screen. All these three preference types are defined in the APL and the library also defines other types of preferences like `SwitchPreference` and `EditTextPreference`.

Besides of stipulating the types of the preferences, a preference resource file usually also provides the following essential information about each preference defined in it: (1) `key`: a unique key that can be used to access the preference value; (2) `title`: a piece of text to be shown on the settings screen when the preference is displayed; (3) `defaultValue`: the initial value of the preference; and (4) `entryValues`: a list of possible values that the preference may take. Note that attribute `entryValues`

```
<PreferenceScreen>
   <CheckBoxPreference
      key="widget_update_location_pref_key"
      title="Update Location"
      defaultValue="false"/>
   <ListPreference
      key="widget_theme_pref_key"
      title="Widget theme"
      defaultValue="Light"
      entryValues={"Dark", "Light"}/>
      ...
</PreferenceScreen>
```

Listing 1. Excerpt from the preference resource file of app *good-weather*.

of a `CheckBoxPreference` is typically omitted in such definitions, since it always contains two values `true` and `false`, and that we sometimes refer to preferences simply by their titles or keys when the meaning is clear from the context.

We identify in total four common patterns, namely patterns APA, APF, SPF, and LHA, that programmers often follow when instantiating settings screens based on preference hierarchies defined by resource files. Patterns APA, APF and LHA [5, 22, 26] were recommended by the Android official guidance website [16] before 2019 but SPF has been the recommended pattern to adopt afterwards. In all the four patterns, a preference hierarchy is essentially linked to its corresponding settings screen(s) during the creation of an activity, which is referred to as the *anchor activity* of the hierarchy.

Particularly, in pattern APA (Adding Preferences to an Activity) a preference hierarchy is linked to its anchor activity by using the resource file as the parameter to invoke method `addPreferencesFromRes-ource` on the activity in the activity's `onCreate` method; In both patterns APF (Adding Preferences to a Fragment) and SPF (Setting Preferences to a Fragment), a preference hierarchy is linked to a fragment and the fragment is instantiated in method `onCreate` of the hierarchy's anchor activity. The difference between the two patterns is that, in pattern APF the preference hierarchy is linked to the fragment by using the resource file as the parameter to invoke method `addPreferencesFromResource` on the fragment in method `onCreate` of that fragment, while the preference hierarchy is linked to the fragment in pattern SPF by using the resource file as the parameter to invoke method `setPreferencesFromResource` on the fragment in method `onCreatePreferences` of that fragment. In patterns APA, APF, and SPF, if the parameter resource file contains nested `PreferenceScreen` elements, multiple settings screens will be instantiated by the APL to render the preferences, with each screen showing only the preferences *directly* contained by a specific `PreferenceScreen`. Particularly, in such a case, the `title` of a child `PreferenceScreen` will be shown on the settings screen for its parent `PreferenceScreen`, and tapping the text will cause the settings screen for the child `PreferenceScreen` to be displayed. The LHA pattern (Loading Headers to an Activity) describes another common way to organize preferences into different settings screens. In pattern LHA, a preference hierarchy with a root element of type `preference-headers` is linked to its anchor activity by using the resource file as the parameter to invoke method `loadHeadersFromResource` on the activity in method `onCreate` of that activity. Here, a `preference-headers` element may contain a list of `header` elements, while each `header` element has a textual description stored in its attribute `title` and is associated with a fragment that is linked to a preference hierarchy as in pattern APF or SPF. At runtime, the anchor activity will show the titles of those `header` elements, and tapping the title of a particular `header` will cause the fragment associated with that `header` to be displayed.

```
String key = "widget_update_location_pref_key";
SharedPreferences sp = SharedPreferences.getDefaultSharedPreferences(...)
boolean val = sp.getBoolean(key, ...);  // to get
sp.setBoolean(key, ...);                // to set
```

Listing 2. To get or set a preference value via `SharedPreferences`.

The APL also defines APIs that enable Android apps to easily access preference values by keys at runtime. For example, to get and set the value of preference `Update location` defined in app *good-weather*, methods `getBoolean` and `setBoolean` from the APL can be invoked, respectively, on the singleton `SharedPreferences` object. The code snippet in Listing 2 demonstrates the use of these methods.

It is worth noting that not all implementations of app settings in Android are based on the APL. For example, some developers may decide to implement their app settings by programming from scratch their own settings classes. In the rest of this paper, we refer to app settings that are implemented based on the APL as *APL preferences* and the settings implemented in other ways as *non-APL preferences*.

4 MAGNITUDE OF THE PROBLEM

In this section, we empirically evaluate the prevalence of preferences and preference-related bugs, as well as the popularity of various preference types, in Android apps. In particular, we aim to address the following three questions: Q1) How often settings in Android apps are implemented using APL preferences? Q2) How many bugs in Android apps are related to APL preferences? Q3) Which preference types are the most frequently used? Answers to these questions will not only help us gain a better understanding about the necessity of preference-wise testing for Android apps, but also shed light on the preference types that we should focus on when providing support for preference-wise testing.

As we explain later in this section, to answer the three questions involves manual examination of both the source code and the executions of the subject apps. Therefore, we selected open-source Android apps that are also available on Android application markets like Google Play and F-Droid as our subjects. Particularly, we first gather all apps from a list of open-source Android apps hosted on Github [46]. The list of apps was rather popular and has been used in quite a number of previous studies on Android applications [34, 50, 55]. Apps on the list were organized into 16 categories. We excluded apps in categories `Android TV` and `Android Wear`, since our study involves running the apps, while apps from those two categories can only be installed on specific types of devices. The remaining 14 categories contained in total 244 apps, among which 200 were available for download in Google Play and/or F-Droid (as of July 2019). We therefore used the 200 apps as the subjects for this study. Table 1 lists the total number of apps included in each category (#Total). Note that, since none of the 200 subject apps falled in category `Business`, only 13 categories are listed in the table.

To answer question Q1, we first installed and manually navigated through each app to determine whether it contains any screens devoted to settings. Then, for apps that do have settings, we check their source code to find out whether their settings are implemented using preference classes from the APL. Manual examination is necessary here since non-APL preferences can be very different in their implementations and appearances and therefore are challenging to identify in an automated fashion.

Table 1 summarizes the results from the examination. Among the 200 apps, 129 had preferences and 115 implemented those settings based on the APL, which suggests preferences are common in Android apps and the most dominant way to implement them is by using the APL. Having said that,

Table 1. Prevalence of preferences in the 200 Android apps. For each CATEGORY, the TOTAL number of apps, the numbers of apps WITHOUT and WITH PREFERENCES, and the breakdown of the latter into the numbers of apps with APL and NON-APL PREFERENCES. The percentage for a category gives the ratio of apps with APL preferences to the total number of apps in the category (APL/TOTAL).

		PREFERENCE			
CATEGORY	TOTAL	WITHOUT	WITH	APL	NON-APL
Communication	16	5	11	10 (63%)	1
Education	5	1	4	3 (60%)	1
Finance	6	2	4	4 (67%)	0
Game	14	8	6	3 (21%)	3
Health Fitness	2	0	2	2 (100%)	0
Life Style	6	1	5	3 (50%)	2
Multi-Media	27	11	16	15 (56%)	1
News and Magazines	20	6	14	14 (70%)	0
Personalization	7	2	5	4 (57%)	1
Productivity	15	6	9	8 (53%)	1
Social Network	18	7	11	10 (56%)	1
Tools	55	22	33	32 (58%)	1
Travel and Local	9	2	7	7 (78%)	0
Overall	200	71	129	115 (58%)	14

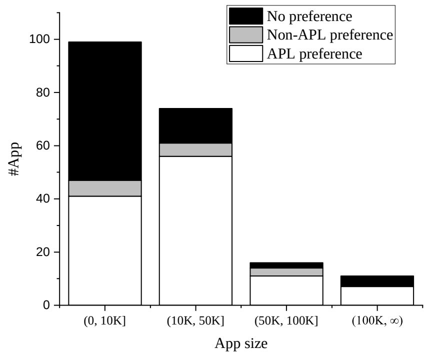


Fig. 2. Relation between app size and numbers of apps with no preferences, with non-APL preferences, and with APL preferences. App size is measured in number of lines of code contained.

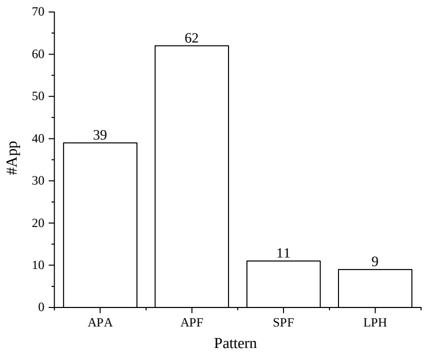


Fig. 3. The number of apps that use each pattern to instantiate settings screens.

the number of apps without preferences was larger than we expected, and the main reason is that almost half of the apps we examined in this study were relatively small in size and did not need any settings to customize their functionalities. Figure 2 shows how the numbers of apps with no preferences, with non-APL preferences, and with APL preferences vary across apps with different sizes. It is clear from the figure that most apps with no preferences were relatively small in size. To be more specific, 73% (=52/71) of the apps with no preferences had no more than 10,000 lines of code, while 73% (=74/101) of the apps with larger size contained APL preferences.

Table 2. Preference-related bugs in 10 apps with the most stars. For each APP, the TOTAL number of reported bugs, the total number of PREFERENCE-RELATED bugs, and the breakdown of that into the numbers of INCORRECT-preference-DEF and INCORRECT-preference-USE bugs. Since no bug combining incorrect preference definition and use was encountered in the study, the corresponding column is omitted from the table.

APP	TOTAL	PREFERENCE-RELATED	INCORRECT-DEF	INCORRECT-USE
AmazeFileManager	256	46 (18.0%)	11	35
AntennaPod	385	76 (19.7%)	16	60
Douya	19	2 (10.5%)	1	1
K9Mail	516	99 (19.2%)	15	84
Launcher3	27	3 (11.1%)	0	3
NewPipe	243	49 (20.2%)	10	39
OwnCloud	428	71 (16.6%)	11	60
Signal	104	20 (19.2%)	1	19
Uhabits	55	5 (9.1%)	2	3
zxing	55	12 (21.8%)	0	12
Overall	2088	383 (18.3%)	67	316

We have also manually inspected the implementations of the apps with APL preferences to find out how they instantiate settings screens based on preference hierarchies. Figure 3 gives, for each of the patterns we identified in Section 3, the number of apps that followed the pattern to instantiate at least one of its settings screens. Note that the numbers for the four patterns add up to 121 because 6 apps implemented multiple patterns in constructing their settings screens. The figure shows that, while patterns APA and APF are the most often adopted among the four, a significant number of apps follow patterns SPF and LPF in constructing their settings screens. It is therefore important that PREFEST should support all the patterns for it to be widely applicable.

To answer question Q2, we examined code repositories of the apps with APL preferences on GitHub to find out the percentage of defects reported on those apps that were related to preferences. Among the 115 apps with APL preferences, 16 apps were excluded from our examination because they did not list any issues in their repositories, 20 were excluded because they did not use tags like `bug` or `crash` to label issues as defects, and 27 were excluded because each of them had less than 10 reported issues labeled as defects. We sorted the remaining 52 apps in decreasing order of their star numbers, selected apps ranked in the top 10 positions, and manually checked each defect reported for those apps in two steps to determine whether it is related to preferences. First, in case a defect is corrected by a specific commit, the defect is considered an *incorrect-preference-def* bug if the commit modifies an `onPreferenceChange` callback method, which is to be invoked when the value of a preference is changed, and the defect is considered an *incorrect-preference-use* bug if the commit modifies code that is guarded by a condition with data dependence to a preference value. Both incorrect-preference-def and incorrect-preference-use bugs are preference-related. Second, when no such commit could be identified for a defect, we mark the defect as preference-related if its description explicitly mentions that certain preferences should be properly set in order to trigger the defect. In such a case, the defect is considered an incorrect-preference-def bug if, according to the description, the associated failure occurs immediately after changing the preference value. Otherwise, the defect is considered an incorrect-preference-use bug.

Table 2 reports on the examination results. In total, 2088 bugs were reported on GitHub for the 10 apps. Among those bugs, 383, or 18.3%, were related to preferences, with 67 being incorrect-preference-def bugs and 316 being incorrect-preference-use bugs. While apps with relatively fewer reported bugs like *Douya*, *Launcher3* and *Uhabits* had slightly lower percentage of preference-related bugs, all apps with more than 100 reported bugs had over 15% of their bugs related to preferences. Overall, such results suggest a significant percentage of bugs reported on Android apps are indeed related to preferences.

To answer question Q3, we examined settings screens from the 115 apps with APL preferences and counted how many times each type of APL preference was used. Table 3 lists 6 preference types that were the most frequently used in those apps. The top four preference types are standard preference types provided by the APL, and they together account for 90.4% of all the preferences used in the 115 apps. The bottom two preference types are supported by third party libraries, which may be the reason for the significantly smaller numbers of their occurrences in the studied apps.

Overall, the results of this empirical study clearly show that preferences are widely used in Android apps and a significant portion of bugs in those apps are indeed related to preferences, which underlines the importance of, and the necessity for, effective preference-wise testing. Since most settings in Android apps are of the six preference types listed in Table 3, Prefest focuses on supporting the effective testing of those preference types in its current implementation.

5 PREFERENCE-WISE TESTING VIA TEST CASE AMPLIFICATION

Figure 4 depicts an overview of Prefest. Prefest takes the APK file of an App Under Test (AUT) and a set of test cases for the AUT as the input, and it amplifies each test case with a group of configurations under which the test case will execute differently. More concretely, first Prefest identifies preferences that a test case depends on by symbolically analyzing the statements along the execution trace of the test case (Section 5.1), then it discovers how the preferences can be accessed at the GUI level via a combination of static and dynamic analysis of the AUT (Section 5.2), and in the end it constructs appropriate configurations for each input test case in an iterative fashion to cover more preference related branches of the AUT (Section 5.3). The final results produced by Prefest include a group of amplified test cases and their execution outcomes.

The rest of this section describes in detail how Prefest achieves effective preference-wise testing by amplifying test cases with appropriate configurations, and the description makes use of the following notations. Let P be the app under test, $T = \{t_1, t_2, \ldots t_n\}$ $(n > 0)$ be the set of input

Table 3. Six preference types that were the most frequently used in the 115 apps with APL preferences. For each TYPE, the number of valid values a preference of that type typically can take (#VALUE), the TOTAL and AVERAGE numbers of times preferences of that type were used in those apps, and the PERCENTAGE of all preferences used in those apps that are of the type. A `CheckBoxPreference` or `SwitchPreference` may only take `Boolean` values (i.e., `true` or `false`), the sets of valid values for `ListPreference`s and `SeekBarPreference`s are often small in size (i.e., containing fewer than a few hundred elements), while the sets of valid values for `EditTextPreference`s and `ColorPreference`s typically are much larger (i.e., containing more than a few thousand elements).

TYPE	#VALUE	TOTAL	AVERAGE	PERCENTAGE
CheckBoxPreference	2	851	7.4	37.2%
ListPreference	small	557	4.8	24.3%
SwitchPreference	2	453	3.9	19.8%
EditTextPreference	large	207	1.8	9.1%
ColorPreference	large	33	0.3	1.6%
SeekBarPreference	small	18	0.2	0.8%
Overall		2124	19.9	93.0%

test cases for P, and Δ be the set of all preferences defined in P. We model each preference δ ($\delta \in \Delta$) using a 5-tuple $\langle type, key, title, entryValues, locators\rangle$, where *type*, *key*, and *title* are the type, key and title of the preference, respectively, *entryValues* is the sorted set of possible entry values the preference may have, while *locators* is the set of identified locators for the preference (more about preference locators in Section 5.2). Let Θ be the union of all preferences' entry values, i.e., $\Theta = \cup_{\delta \in \Delta} \delta.entryValues$. A configuration $\phi : \Delta \nrightarrow \Theta$ assigns valid values to preferences, i.e., $\forall \delta \in dom(\phi) : \phi(\delta) \in \delta.entryValues$, where $dom(\phi)$ is the domain of ϕ. A configuration ϕ is *full* if $dom(\phi) = \Delta$ and it is *partial* if $dom(\phi) \subset \Delta$. Given two configurations ϕ_1 and ϕ_2, ϕ_1 is *compatible* with ϕ_2 if all preferences defined in ϕ_2 are assigned with the same values by ϕ_1, i.e., $\forall \delta \in dom(\phi_2) : \delta \in dom(\phi_1) \wedge \phi_1(\delta) = \phi_2(\delta)$. Each amplified test case is a pair $\langle t, \phi\rangle$, where $t \in T$ is a test case and ϕ is a full configuration.

5.1 Test-Relevant Preference Identification

According to our experience, a test case usually only interacts with a few preferences. It is therefore understandable that not all changes to a configuration will cause a specific test case to execute differently under that configuration. To make sure our amplification of the input test cases is effective and efficient, PREFEST first identifies preferences that are relevant to each test case $t \in T$: A preference $\delta \in \Delta$ is considered *relevant* to an amplified test case $\langle t, \phi\rangle$ if at least one branch condition executed by t under ϕ is data-dependent on the value of δ. Let $R_{\langle t,\phi\rangle}$ ($R_{\langle t,\phi\rangle} \subseteq \Delta$) be the set of preferences relevant to $\langle t, \phi\rangle$. The values of preferences in $R_{\langle t,\phi\rangle}$ can be essential in steering t's execution under ϕ to take certain branches, while the importance of those preferences has been largely neglected by existing approaches for Android testing.

PREFEST applies a customized, lightweight symbolic analysis to identify the relevant preferences. On the one hand, symbolic analysis is needed here to track which and how preference values are used in P, since those values are usually accessed via method invocations on the `SharedPreference` singleton object and an invocation may access the values of distinct preferences in various executions if the key used in that invocation evaluates to different results. On the other hand, off-the-shelf symbolic execution tools cannot be directly applied here, partly because of the scalability issues they suffer from [24], and partly because of the event-driven nature as well as the underlying application development framework (ADF) [40] of the apps.

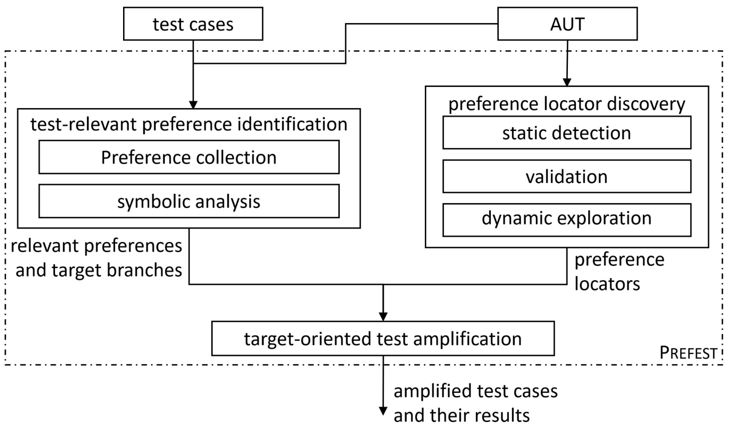

test cases
AUT
test-relevant preference identification
Preference collection
symbolic analysis
preference locator discovery
static detection
validation
dynamic exploration
relevant preferences and target branches
preference locators
target-oriented test amplification
PREFEST
amplified test cases and their results

Fig. 4. Overview of PREFEST.

$$\begin{aligned} \mathit{stmt} &::= \texttt{skip} \mid \texttt{stop} \mid \mathit{simps} \mid \mathit{ifs} \mid \mathit{prefs} & \mathit{simps} &::= \mathit{rv} = e \\ \mathit{ifs} &::= \texttt{if}(e)\ \ l_1\ \ \texttt{else}\ \ l_2 & \mathit{prefs} &::= \mathit{sv} = \texttt{get}(e) \mid \texttt{set}(e, e') \\ e &::= \mathit{const} \mid \mathit{rv} \mid \mathit{sv} \mid \mathit{op}(\bar{e}) \mid \mathit{md}(\bar{e}) \end{aligned}$$

$$e \in \texttt{Expr}\quad \mathit{const} \in \texttt{Const}\quad \mathit{rv} \in \texttt{RegVar}\quad \mathit{sv} \in \texttt{SymVar}\quad \bar{e} \in \texttt{Exprs}\quad \mathit{op} \in \texttt{Operator}$$
$$\mathit{md} \in \texttt{LibMethod}\quad \mathit{simps} \in \texttt{SimpleStmt}\quad l_1, l_2 \in \texttt{Label}\quad \mathit{ifs} \in \texttt{IfStmt}\quad \mathit{prefs} \in \texttt{PrefStmt}$$
$$\texttt{Var} = \texttt{RegVar} \cup \texttt{SymVar}$$

Fig. 5. Syntax of the core language. A statement is either a skip statement, a stop statement, a simple statement, an if statement, or a preference access statement. A simple statement assigns the value of an expression e to a regular variable rv; An if statement of form $\texttt{if}(e)\ l_1\ \texttt{else}\ l_2$ transfers the execution to l_1 if e has value true and l_2 if e has value false; A statement of form $sv = \texttt{get}(e)$ gets the value of a preference by key e and assigns the value to a symbolic variable sv; A statement of form $\texttt{set}(e, e')$ sets the value of a preference by key e with the value of e'.

Given the APK file of P, Prefest first extracts the preference resource files from the APK with the help of the *jadx* decompiler[1], and then it parses the resource files to retrieve the collection Δ of preferences defined in P. For each preference, Prefest initializes its *type*, *key*, *title*, and *entryValues* based on attributes from the corresponding resource file. Next, Prefest instruments app P to log the traces of its executions. For each test case $t \in T$, Prefest runs t on the instrumented app under the default configuration ϕ_0 and gathers not only the statements executed but also the library APIs invoked during t's execution. Based on the gathered information, Prefest unfolds the loops and inlines the invocations to non-library methods in P to construct a program $P_{\langle t,\phi_0\rangle}$ that would produce the identical execution as P if executed under ϕ_0 with the same inputs as used in t. The following symbolic analysis is applied to $P_{\langle t,\phi_0\rangle}$. Figure 5 gives the syntax of the core language of $P_{\langle t,\phi_0\rangle}$ and we will use the language to present the algorithm for the analysis. Note that switch statements in P are translated into nested if statements in $P_{\langle t,\phi_0\rangle}$ in a natural way, and that this core language contains no loops and only invocations to methods from libraries, since loops have been unfolded and invocations to non-library methods have been inlined when constructing $P_{\langle t,\phi_0\rangle}$.

We denote program $P_{\langle t,\phi_0\rangle}$ as a pair $\langle S, N\rangle$, where $S : \mathit{Label} \longrightarrow \mathit{Statement}$ maps each label to the corresponding statement, $N : \mathit{Label} \longrightarrow \mathit{Label}$ maps each label to the label of the next statement to execute. Here, *Statement* is the set of statements contained in $P_{\langle t,\phi_0\rangle}$, each statement is associated with a unique label, or id, and *Label* is the set of labels associated with those statements. $l_0 \in \mathit{Label}$ is the label of the first statement in $P_{\langle t,\phi_0\rangle}$. Let $M_{\langle t,\phi_0\rangle} : \mathit{Label} \longrightarrow \mathit{Location}$ be a map that projects each label l of $P_{\langle t,\phi_0\rangle}$ to a location in the source code of P where the statement at $S(l)$ originally appears. From the execution trace of $\langle t, \phi_0\rangle$, we can easily derive a branching history $BH_{\langle t,\phi_0\rangle} : \mathit{Label}_b \nrightarrow \mathit{Boolean}$ that maps the label of each branch condition executed by $\langle t, \phi_0\rangle$ to its evaluation result during that execution. Here, $\mathit{Label}_b \subset \mathit{Label}$ is the set of labels for the if statements in $P_{\langle t,\phi_0\rangle}$.

A state $\langle l, \sigma, \phi, EH\rangle$ of program $P_{\langle t,\phi_0\rangle}$ contains the label l of the next statement to execute, the current environment $\sigma : \mathit{Var} \longrightarrow \mathit{Expr}$ that maps each variable to the expression encoding its value, the active full configuration $\phi : \mathit{Key} \longrightarrow \mathit{Expr}$ that maps the key of each preference to its value, and the branch evaluation history $EH : \mathit{Label}_b \longrightarrow \mathit{Expr}$ that maps the label of each executed branch condition to the symbolic value of the condition expression. The inference rules in Figure 6 present the operational semantics of the simple statements, if statements, and preference access statements, where the notation "$\sigma, \phi \vdash e \Rightarrow e'$" indicates that expression e *symbolically* evaluates to e' under environment σ and configuration ϕ. Take rule *simp* as an example. The rule stipulates

[1]https://github.com/skylot/jadx

$$simp: \frac{S(l) = rv = e \qquad \sigma, \phi \vdash e \Rightarrow e'}{\langle l, \sigma, \phi, EH\rangle =\!\![\langle S, N\rangle]\!\!\Rightarrow \langle N(l), \sigma[rv \mapsto e'], \phi, EH\rangle}$$

$$if_{true}: \frac{S(l) = \texttt{if}(e)\ l_1\ \texttt{else}\ l_2 \qquad \sigma, \phi \vdash e \Rightarrow e' \qquad BH(l) = \texttt{true}}{\langle l, \sigma, \phi, EH\rangle =\!\![\langle S, N\rangle]\!\!\Rightarrow \langle l_1, \sigma, \phi, EH[l \mapsto e']\rangle}$$

$$if_{false}: \frac{S(l) = \texttt{if}(e)\ l_1\ \texttt{else}\ l_2 \qquad \sigma, \phi \vdash e \Rightarrow e' \qquad BH(l) = \texttt{false}}{\langle l, \sigma, \phi, EH\rangle =\!\![\langle S, N\rangle]\!\!\Rightarrow \langle l_2, \sigma, \phi, EH[l \mapsto e']\rangle}$$

$$pref_{get}: \frac{S(l) = sv = \texttt{get}(e) \qquad \sigma, \phi \vdash e \Rightarrow k \qquad \phi(k) = e'}{\langle l, \sigma, \phi, EH\rangle =\!\![\langle S, N\rangle]\!\!\Rightarrow \langle N(l), \sigma[sv \rightarrow e'], \phi, EH\rangle}$$

$$pref_{set}: \frac{S(l) = \texttt{set}(e, e') \qquad \sigma, \phi \vdash e \Rightarrow k \qquad k \in Key \qquad \sigma, \phi \vdash e' \Rightarrow e''}{\langle l, \sigma, \phi, EH\rangle =\!\![\langle N(l), \sigma, \phi[k \rightarrow e''], EH\rangle}$$

Fig. 6. Operational semantics of the simple, if, and preference access statements in the core language presented as a set of inference rules.

```
    ...
l1: $r1 = "widget_"
    ...
l2: $r2 = $r1 + "update_location_pref_key"
    ...
l3: $r3 = SharedPreferences.getDefaultSharedPreferences()
l4: $z0 = $r3.getBoolean($r2,0)
    ...
l5: $z1 = !$z0
l6: if($z1 == 0)
    ...
```

Listing 3. Excerpt from a test execution trace of app *good-weather*.

that, from a current state $\langle l, \sigma, \phi, EH\rangle$, if the next statement to execute is a simple statement $rv = e$ and expression e evaluates to e' under the current environment σ and configuration ϕ, executing the statement will transit the program to a new state $\langle N(l), \sigma[rv \mapsto e'], \phi, EH\rangle$, where the next statement to execute is at label $N(l)$ and regular variable rv has value e'. Note that, at the beginning of the symbolic execution, ϕ maps the key of each preference $\delta \in \Delta$ to a unique symbolic value μ_δ, i.e., $\forall\delta : \delta \in \Delta \longrightarrow \phi(\delta.key) = \mu_\delta$.

At the end of the symbolic analysis of $P_{\langle t,\phi_0\rangle}$'s execution, the final branch evaluation history, denoted as $EH_{\langle t,\phi_0\rangle}$, maps the label of each executed branch to the symbolic value of that branch's condition expression. From $EH_{\langle t,\phi_0\rangle}$, Prefest can easily discover all the symbolic values, and therefore the corresponding relevant preferences, that were used in evaluating the branch conditions during $\langle t, \phi_0\rangle$'s execution.

Take app `good-weather` in Section 2 as an example. Listing 3 shows part of the execution trace produced by a test $t1$ on the app under configuration $\phi 1$: Statements at labels $l1$ and $l2$ construct the key of preference `Update location` through string concatenation; Statement at $l3$ obtains the singleton object of `SharedPreferences`; Statement at $l4$ invokes a library method `getBoolean` on the singleton object with variable $\$r2$ as the key and assigns the obtained preference value to $\$z0$; Statement at $l5$ assigns the negation of $\$z0$ to $\$z1$, which is then tested in the branch condition at

Table 4. Instance identification for settings screen instantiation patterns. For each of the four patterns, the CRITERION to be used for identifying instances of the pattern, the id of the criterion (C-ID), as well as the associated preference resource file (PRF) and ANCHOR activity if an instance of the pattern has been successfully identified.

PATTERN	C-ID	CRITERION	PRF	ANCHOR
APA	C-APA	$cls(m_1) \in A \land m_1 = oc(cls(m_1)) \land isAddP(m_2)$	$arg(m_2)$	$cls(m_1)$
APF	C-APF	$cls(m_1) \in A \land m_1 = oc(cls(m_1)) \land m_2 \in init(cls(m_2))$ $\land cls(m_1') = cls(m_2) \land cls(m_1') \in F \land m_1' = oc(cls(m_1')) \land isAddP(m_2')$	$arg(m_2')$	$cls(m_1)$
SPF	C-SPF	$cls(m_1) \in A \land m_1 = oc(cls(m_1)) \land m_2 \in init(cls(m_2))$ $\land cls(m_1') = cls(m_2) \land cls(m_1') \in F \land m_1' = ocP(cls(m_1')) \land isSetP(m_2')$	$arg(m_2')$	$cls(m_1)$
LHA	C-LHA	$cls(m_1) \in A \land m_1 = oc(cls(m_1)) \land isLoadH(m_2) \land cls(m_1') \in F$ $\land cls(m_1') \in fReferenced(m_2) \land m_1' = oc(cls(m_1')) \land isAddP(m_2')$	$arg(m_2')$	$cls(m_1)$

$l6$. By symbolically executing the program, Prefest is able to find out the symbolic value of the branch condition at $l6$ is $(!\mu_{ul}) == 0$, where μ_{ul} is the symbolic value of preference `Update location`. Therefore, Prefest will consider the preference as *relevant* to test $\langle t1, \phi 1\rangle$.

5.2 Preference Locator Discovery

Prefest determines whether a test case $t \in T$ should be amplified with a specific configuration ϕ ($\phi \neq \phi_0$) based on t's execution result under ϕ, therefore it is critical for Prefest to be able to set the preference values of P accordingly before executing t. Given that programmatically setting preferences, e.g., by invoking methods on the `SharedPreferences` singleton object, risks breaking the integrity of P's state, Prefest always modifies preference values at the GUI level, so that all configurations it generates in test case amplification are actually feasible from a user's point of view.

Prefest uses a set of *locators* to abstract information about how each preference of P can be accessed at the GUI level. A locator for a preference is a pair $\langle \alpha, \xi \rangle$, where α is the anchor activity of the preference's containing hierarchy and ξ is a sequence of texts from the app GUI that a user needs to tap for navigating the app from α to the screen where the preference's corresponding widget is displayed. Prefest discovers locators for preferences in two steps, namely static locator discovery and dynamic locator discovery.

5.2.1 Static Locator Discovery. In static locator discovery, Prefest first analyzes P's preference resource files and the invocation relation between P's methods to identify potential instances of settings screen instantiation patterns, then constructs an initial set of candidate locators for each defined preference, and in the end checks the validity of the candidate locators, i.e., whether they can indeed help locate the preferences on P's GUI, by following their guidance to access the preferences.

Pattern Instance Identification. As explained in Section 3, programmers often follow four patterns when instantiating settings screens based on preference hierarchies. To identify the anchor activities of the preference hierarchies, Prefest examines the invocations in P to specific methods and looks for potential instances of those patterns based on a group of criteria.

Table 4 lists the criteria Prefest applies to identify instances of settings screen instantiation patterns in P. The definition of criteria makes use the following notations. C is the set of all classes in P, M is the set of all methods[2], A is the set of activity classes, and F is the set of fragment classes; Function $init : C \rightarrow 2^M$ maps each class $c \in C$ to its non-empty set of constructors; Function

[2]Prefest treats constructors as member methods with no return types.

Table 5. Important APIs for settings screen instantiation. For each group of APIs, a short description, the list of APIs in that group, and a predicate that is defined on all APL APIs but returns `true` if and only if when applied to an API from the group.

DESCRIPTION	APIs	PREDICATE
To add preferences	`PreferenceActivity.addPreferencesFromResource` `PreferenceFragment.addPreferencesFromResource`	*isAddP*
To set preferences	`PreferenceFragmentCompat.setPrefernecesFromResource`	*isSetP*
To load preference headers	`PreferenceActivity.loadHeadersFromResource`	*isLoadH*

$oc : A \cup F \longrightarrow M$ maps an activity or a fragment to its unique `onCreate` lifecycle method; Function $ocp : A \cup F \longrightarrow M$ maps an activity or a fragment to its unique `onCreatePreferences` lifecycle method; Funciton $cls : M \longrightarrow C$ maps each method to its defining class; When an instance of a pattern has been identified, $arg\,(m)$ denotes the preference resource file that is used as the parameter to invoke method m in the instance, and $fReferenced\,(m)$ denotes the list of `Fragments` referenced by $arg\,(m)$. Note that, since certain APIs from the APL play important roles in implementing the four patterns, we have also introduced three predicates, namely $isAddP$, $isSetP$, and $isLoadH$, to facilitate the easy identification of those APIs. Table 5 provides more information about the APIs and the corresponding predicates to identify them.

To identify potential instances of the four patterns listed in Table 4, PREFEST first gathers 6 sets S_A, S_B, S_C, S_D, S_E, and S_F of method pairs with invocation relation ($S_A, S_B, S_C, S_D, S_E, S_F \subseteq \tau^+$). Here, $\tau \subseteq M \times M$ is the invocation relation between P's methods, i.e., given two methods $m_1, m_2 \in M$, $\langle m_1, m_2 \rangle \in \tau$ if and only m_1 invokes m_2 in its definition, and τ^+ is the transitive closure of τ, i.e., $\tau^+ = \cup_{i=1}^{\infty} \tau^i$. In each pair $\langle m_1, m_2 \rangle \in S_A$, m_1 is the `onCreate` method of an activity class, and m_2 is method `addPreferencesFromResource`; In each pair $\langle m_1, m_2 \rangle \in S_B$, m_1 is the `onCreate` method of an activity class, and m_2 is the constructor of a fragment class; In each pair $\langle m_1, m_2 \rangle \in S_C$, m_1 is the `onCreate` method of a fragment class, and m_2 is method `addPreferencesFromResource`; In each pair $\langle m_1, m_2 \rangle \in S_D$, m_1 is the `onCreatePreferences` method of a fragment class, and m_2 is method `setPrefernecesFromResource`; In each pair $\langle m_1, m_2 \rangle \in S_E$, m_1 is the `onCreate` method of an activity class, and m_2 is method `loadHeadersFromResource`; In each pair $\langle m_1, m_2 \rangle \in S_F$, m_1 is the `onCreate` method of a fragment class, and m_2 is method `addPreferencesFromResource`. Then, PREFEST examines the pairs in S_A and combinations of pairs from $S_B \times S_C$, $S_B \times S_D$, and $S_E \times S_F$ to identify instances of patterns APA, APF, SPF, and LHA, respectively. A pair of methods $\rho = \langle m_1, m_2 \rangle$ ($\rho \in \tau^+$) is considered implementing pattern APA if criterion C-APA is satisfied, while two pairs of methods $\rho = \langle m_1, m_2 \rangle$ and $\rho' = \langle m_1', m_2' \rangle$ ($\rho, \rho' \in \tau^+$) are considered implementing patterns APF, SPF, and LHA, if criteria C-APF, C-SPF, and C-LHA are satisfied, respectively. For example, given ρ and ρ', if a fragment is instantiated inside the `onCreate` method of an `Activity` class according to ρ and API `PreferenceFragment.addPreferencesFromResource` is invoked by the `onCreate` method of that `Fragment` class according to ρ', the methods involved in ρ and ρ' satisfy criterion C-APF and implement pattern APF.

Candidate Locator Construction. From the identified instances of settings screen instantiation patterns, PREFEST can easily derive the anchor activities associated with the preference resource files, as indicated in Table 4. To construct candidate locators for a preference, PREFEST still needs to find out, when the associated anchor activity is active, what navigation text(s) need to be tapped for the preference's related widget to be displayed.

The pseudo code shown in Algorithm 1 describes how PREFEST finds out such information step by step. Taking an anchor activity *act*, a preference element *elem* from a resource file, and a list

Algorithm 1: Construction of candidate preference locators based on pattern instances.

```
Input: The list instances of identified instances of settings screen instantiation patterns;
Output: A table locators that maps each preference key to the set of locators for that preference;
1  locators ← { } ;
2  foreach instance ∈ instances:
3      rootElem ← GETROOTELEMENT(GETRESOURCEFILE(instance));
4      GETLOCATORS (GETANCHORACTIVITY(instance), rootElem, [ ], locators);
5  Procedure GETLOCATORS(Activity act, DOMElement elem, List preTexts, Map locators):
6      if elem is of type PreferenceScreen or preference-headers:
7          foreach elem′ ∈ GETCHILDELEMENTS(elem):
8              GETLOCATORS (act, elem′, preTexts, locators);
9      else if elem is of a primitive preference type:
10         $locators[elem.key] \leftarrow locators[elem.key] \cup \{\langle act, preTexts\rangle\}$ ;
11     else if elem is of type header:
12         preTexts′ ← APPEND(preTexts, elem.title);
13         elem′ ← GETROOTELEMENT(GETRESOURCEFILEREFERENCED(elem));
14         GETLOCATORS (act, elem′, preTexts′, locators) ;
```

preTexts of preceding navigation texts to tap as the input, procedure getLocators collects the navigation information for preferences defined in either the child elements of *elem* (including *elem* itself) or the resource files referenced by those child elements and makes the information easily accessible by preference keys via the output parameter *locators*: If *elem* is of type `PreferenceScreen` or `preference-headers` (Line 6), the procedure is recursively applied to each of *elem*'s child elements (Lines 7 and 8). Otherwise, if *elem* is of primitive types like `SwitchPreference` and `ListPreference` (Line 9), a candidate locator is constructed and associated to the key of the preference (Line 10). If *elem* is of type `header` (Line 11), *elem*'s text is appended to *preTexts* (Line 12), and then the procedure recursively processes the elements from the preference resource file referenced by *elem* (Lines 13 and 14). To collect the candidate locators for all preferences, Prefest iterates through the list of identified pattern instances (Line 2) and uses the associated anchor activity, the root element of the corresponding resource file, an empty list of preceding navigation texts, and an empty map as the arguments to invoke procedure getLocators (Lines 3 and 4).

Candidate Locator Validation. Since both the identification of pattern instances and the construction of candidate preference locators as described above ignore the data- and control-flow in the app code, a candidate locator $\langle \alpha_0, \xi_0 \rangle$ constructed in this way for a preference δ may be invalid in the sense that ξ_0 does not actually help Prefest steer P from α_0 to δ's containing settings screen. For instance, if δ's residing preference hierarchy is only linked to α_0 during α_0's creation *under specific conditions*, but the actual instance of α_0 Prefest launches for preference setting purposes never satisfies those conditions during preference-wise testing, candidate locators identified by Algorithm 1 with anchor activity α_0 for δ will not be really helpful since the desired preference hierarchy is not even linked to the activity. To prune out such invalid candidate locators, Prefest follows the guidance of each of those locators to check whether it can help the tool access the corresponding preference as expected. Prefest discards all the invalid locators and retains at most one valid locator for each preference.

Algorithm 2 shows how Prefest dynamically checks the validity of the candidate preference locators. After some initialization (Line 1), Prefest first groups preference keys by their potential locators (Lines 2 through 4). Next, for each locator, Prefest takes the keys of all preferences that may

Algorithm 2: Validation of candidate preference locators discovered via static detection.

Input: A map *locators* from each preference key to a set of candidate locators for that preference;
Output: A map *validLocators* from each preference key to a *valid* locator for that preference;

```
1  validLocators ← { }; loc2Pref ← { };
2  foreach key ∈ locators.keys():
3      foreach locator ∈ locators[key]:
4          loc2Pref[locator] ← loc2Pref[locator] ∪ {key} ;
5  foreach locator =⟨anchor, texts⟩ ∈ loc2Pref.keys():
6      keys ← loc2Pref[locator];
7      srcActivity ← launchActivity(anchor) ;
8      destScreen ← tapTexts(srcActivity, texts) ;
9      matchedKeys ← getMatchedKeys(destScreen, keys);
10     if destScreen != null ∧ |matchedKeys| ≥ $N_t$:
11         foreach key' ∈ matchedKeys:
12             if ¬validLocators.contains(key'):
13                 validLocators[key'] ← locator ;
14 return validLocators;
```

be accessible via the locator (Line 6), launches the anchor activity (Line 7), follows the navigation texts of the locator (Line 8), and collects the keys of the preferences whose titles exactly match with text elements shown on the destination screen (Line 9). If the navigation was successful and at least N_m preferences (N_m is empirically set to 3 by default) can be matched, the destination screen is considered a settings screen and the current locator is deemed valid for all the matched preferences (Lines 10 through 13).

5.2.2 *Dynamic Locator Discovery.*

In case all the statically constructed candidate locators for a preference fail to validate, Prefest tries to derive a valid locator for the preference by dynamically exploring *P*'s GUI.

Algorithm 3 shows how this is done in Prefest. First, Prefest gathers the anchor activities for the preferences to locate and their subclasses (Line 1), which will be used as the starting points of the exploration. The subclasses are also included here because they may share instance initialization code with their super-classes and can therefore be used to load settings screens. Next, Prefest launches each gathered activity and invokes procedure explore to start the exploration of the app's GUI from the activity (Lines 2 through 4). Procedure explore takes four arguments: a starting activity *act*, a sequence *preText* of navigation texts that need to be tapped to transit the app from the starting activity to the current screen, a set *prefToLocate* of preferences to locate, and a map *validLocators* from each preference key to a valid locator or *null* (Line 5). The procedure returns immediately if there are no preferences in *prefToLocate* to be located (Lines 6 and 7). When there are more preferences to find locator for, Prefest first extracts the texts shown on the current screen (Line 8). Then, Prefest matches the texts against the titles of preferences and, if the current screen turns out to be a settings screen, finds a preference resource file that is most likely linked to the current screen (Line 9). If such preference resource file is found (Line 10), we have constructed valid locators for preferences displayed on the current screen: For each preference to locate that is found on the screen (Line 11 and 12), arguments *act* and *preTexts* are used to construct a locator for the preference (Line 13), and matched preferences are removed from *prefToLocate* (Line 14). In view that programmers sometimes implement preference headers based on ordinary navigable

Algorithm 3: Preference locator discovery by dynamically exploring the app's GUI.

```
Input: A map $locators$ from preference keys to the sets of potential locators;
Input: The set $prefToLocate$ of preferences that do not have any valid locator;
Input: A map $validLocators$ from each preference key to a valid locator or $null$;

1  $actToExplore \leftarrow \cup_{p \in prefToLocate} locators[p].activity.subclasses()$;
2  foreach $act \in actToExplore$:
3      $currentAct \leftarrow$ launchActivity($act$);
4      explore ($currentAct$, [ ], $prefToLocate$, $validLocators$);
5  Procedure explore(Activity $act$, List $preTexts$, Set $prefToLocate$, Map $validLocators$):
6      if $prefToLocate = \emptyset$:
7          return;
8      $candidateTitles \leftarrow$ getTextViews();
9      $bestMatch \leftarrow$ getBestMatchingPRF($candidateTitles$);
10     if $bestMatch \neq null$:
11         $matchedPrefs \leftarrow$ getMatchedPrefs($candidateTitles$, $bestMatch$);
12         foreach $pref \in prefToLocate \cap matchedPrefs$:
13             $validLocators[pref.\text{key}] \leftarrow \langle act, preTexts \rangle$;
14         $prefToLocate \leftarrow prefToLocate \setminus matchedPrefs$;
15     foreach $navigationText \in$ getNavigationTexts():
16         tap($navigationText$);
17         $preTexts' \leftarrow$ append($preTexts$, $navigationText$);
18         explore ($act$, $preTexts'$, $prefToLocate$);
19         back();
20     return
```

texts, Prefest also collects the navigable texts on the current screen (Line 15), follows each of those texts (Lines 16 and 17), and explores the screens arrived at recursively (Line 18).

5.2.3 Generation of Amplified Test Cases. With the valid locators for preferences, Prefest is now able to generate an amplified test case $\langle t, \phi \rangle$ by prefixing to t a sequence of test actions that set the preferences to desired values as specified in ϕ.

Take app `good-weather` and its test case mentioned in Section 2 that changes the location as an example. The app implements pattern LHA to instantiate its settings screen: A preference headers resource file is loaded by the `onCreate` method of `org.asdtm.goodweather.SettingsActivity`. The activity displays a list of navigation texts that can be tapped to switch to different settings screens, and particularly, tapping text "`Widget settings`" will cause preference `Update Location` to be displayed on the screen. Prefest is able to detect a valid locator for the preference, where the anchor activity is `org.asdtm.goodweather.SettingsActivity` and the list of navigation texts includes only "`Widget settings`", and it generates test actions as shown in Listing 4 as a prefix to the original test case. With the prefixed test actions, preference `Update Location` will be set to `true` before the original test case starts. In the generated test actions, the anchor activity for the preference hierarchy is first launched via command `adb shell am start` of the Android Debugging Bridge [15]. Then, the navigation texts from the locator are tapped in order to open the settings screen for the preference. Next, if the current value of the preference is `false`, the preference is tapped to change its value to `true`. Similar test actions can be generated to set the values of other preferences, if necessary.

```
# setting preferences
os.popen("adb shell am start org.asdtm.goodweather.SettingsActivity");
getElement("text(\"Widget settings\")").tap();
if (getElememt("text(\"Update location\")").checkEnable() != true)
      getElememt("text(\"Update location\")").tap();
back();
```

Listing 4. Test actions constructed by Prefest to set preference `Update location`.

5.3 Target-Oriented Test Amplification

While the preferences that are relevant to a test case may be just a small portion of all preferences defined in an app, exhaustively trying out all possible value combinations of those relevant preferences, e.g., by following a N-wise combinatorial strategy (N>1), is often still way too expensive and uneconomical, in terms of the testing time, the amount of code covered, and the number of bugs detected. In view of that, test case amplification in Prefest aims to exercise more preference-dependent behaviors of apps, rather than to blindly exhaust all possible preference value combinations.

Prefest models app behaviors in terms of the set of code branches they cover, and it uses a triple $\langle loc, expr, val\rangle$ to abstract each code branch, where *loc* is a location in the app's code, *expr* is an expression, and *var* is a value. A branch $b = \langle loc, expr, val\rangle$ is *covered* by an execution if the execution reaches *loc* and *expr* evaluates to *val* at *loc* at least once during the execution; *b* is *bypassed* by an execution if the execution reaches *loc* but *expr* never evaluates to *val* at *loc* during the execution; and *b* is *missed* by an execution if the execution never reaches *loc*. *b* is called a *target branch*, or just a *target* for brevity, if *expr* is preference-dependent, i.e., *expr* is data-dependent on certain preferences.

In the rest of this section, we first define two types of branches, namely *control* and *parameter* branches, that Prefest is concerned with, then describe how Prefest analyzes the executions of P's amplified test cases to collect those branches, and in the end explain how Prefest generates new amplified test cases for P in iterations to cover more preference-related branches. Recall from Section 5.1 that all input test cases are always first amplified with the default configuration by Prefest.

5.3.1 Control and Parameter Branches. Branches that originate from constructs like conditionals and loops within P are called *control* branches. For each if statement with condition e_1 at location loc_1, Prefest constructs two control branches $\langle loc_1, e_1, true\rangle$ and $\langle loc_1, e_1, false\rangle$; For each switch statement with expression e_2 and k case clauses at location loc_2, Prefest constructs k control branches $\langle loc_2, e_2, v_i\rangle$, where v_i is the constant used in the i-th case clause ($1 \le i \le k$); For each loop statement with condition expression e_3 at location loc_3, Prefest constructs two control branches $\langle loc_3, e_3, true\rangle$ and $\langle loc_3, e_3, false\rangle$.

Recall that, for each amplified test case $\langle t, \phi\rangle$ and the corresponding program $P_{\langle t,\phi\rangle} = \langle S', N'\rangle$, Prefest gathers via symbolic analysis a map $EH_{\langle t,\phi\rangle}$ from the labels of executed if conditions to the symbolic values of those conditions and a map $BH_{\langle t,\phi\rangle}$ from the labels of executed if conditions to their concrete Boolean values in the execution (see Section 5.1). The set $\Gamma_{\langle t,\phi\rangle}$ of control branches covered by $\langle t, \phi\rangle$ can therefore be calculated as $\Gamma_{\langle t,\phi\rangle} = \{\langle loc, expr, val\rangle | \exists l \in dom(S') : M_{\langle t,\phi\rangle}(l) = loc \wedge BH_{\langle t,\phi\rangle}(l) = val\}$, and the set $\Sigma_{\langle t,\phi\rangle}$ of control branches bypassed by $\langle t, \phi\rangle$ can be calculated as $\Sigma_{\langle t,\phi\rangle} = \{\langle loc, expr, val\rangle | \exists l \in dom(S') : (M_{\langle t,\phi\rangle}(l) = loc) \wedge \forall l \in dom(S') : (M_{\langle t,\phi\rangle}(l) = loc \implies BH_{\langle t,\phi\rangle}(l) \neq val)\}$.

If the actual parameters used in an invocation to a library method m at location loc are data-dependent on a set $\Delta' = \{\delta_1, \delta_2, \ldots, \delta_k\}$ of preferences ($k \ge 1 \wedge \Delta' \subseteq \Delta$), the values of those preferences before the invocation may affect which control branches will be covered during the execution of m. Prefest, however, does not have detailed information about such influences, since the symbolic execution it applies treats all library methods as black-boxes. To cover as many target control branches in m as possible, Prefest conservatively assumes distinct value combinations for preferences in Δ' at the invocation location will always cause m to cover additional target control branches. Correspondingly, let V_i be the set of valid entry values for δ_i $(1 \leq i \leq k)$, Prefest constructs a set $\Lambda_{loc} = \{\langle loc, \delta_1 = v_1 \wedge \ldots \wedge \delta_k = v_k, true\rangle | v_1 \in V_1 \wedge \ldots \wedge v_k \in V_k\}$ of *parameter* branches to capture the configurations under which the library method invocation should be executed for it to be thoroughly tested. Note that all parameter branches are also target branches by definition.

5.3.2 Generation of Candidate Amplifications to Cover Bypassed Targets. Prefest monitors and records all targets that are covered or bypassed by the execution of each amplified test case. To produce a new configuration under which an input test case t will cover a previously bypassed target, Prefest follows an idea similar to that adopted in symbolic-execution-based test generation. More concretely, Prefest first constructs a constraint to encode the conditions that the configuration should satisfy to drive t to cover the bypassed target and then derives the desirable values for the preferences based on solutions to the constraint.

Given an amplified test case $\langle t, \phi\rangle$ and a control target $ct = \langle loc, expr, val\rangle$ bypassed by $\langle t, \phi\rangle$, there exists by definition another control target $ct' = \langle loc, expr, val'\rangle$ at the same location that was covered by the amplified test case. In other words, there exists a label l in $P_{\langle t,\phi\rangle}$ such that the statement at l was executed by $\langle t, \phi\rangle$, label l maps to location loc, and $expr$, i.e., $EH_{\langle t,\phi\rangle}(l)$, evaluated to val' at l during the execution of $\langle t, \phi\rangle$. To cover ct, Prefest tries to find a different configuration ϕ' such that the following two conditions are satisfied. First, the execution of $\langle t, \phi'\rangle$ should follow the same path as that of $\langle t, \phi\rangle$ until reaching label l; Second, $EH_{\langle t,\phi'\rangle}(l')$ should evaluate to val, i.e., $EH_{\langle t,\phi'\rangle}(l') == val$, where l' is the counterpart label of l in $P_{\langle t,\phi'\rangle}$. Both constructing a constraint to encode the first condition and finding a solution to satisfy the conjunction of the two conditions, however, can be extremely challenging or even infeasible in practice, due to limitations in existing symbolic execution and constraint solving techniques. Prefest therefore always weakens the constraint that it actually solves in finding the desirable configurations by omitting the first condition while only retaining the constraint $EH_{\langle t,\phi'\rangle}(l') == val$. To derive desirable configurations from the weakened constraint, Prefest enumerates all valid assignments to the involved preferences and evaluates the constraint under these assignments. All configurations that can make the constraint hold are considered as candidate amplifications associated with target ct.

Similarly, to construct and solve a constraint that faithfully encodes the condition under which a bypassed parameter target $pt = \langle loc, expr, val\rangle$ will be covered by test case t is also highly challenging or even impractical. Therefore, Prefest requires instead that the constraint $expr = val$ to be satisfied at the beginning of t, and it generates all value combinations of the preferences referenced in $expr$ as candidate amplifications associated with target pt.

Three things about the generation of candidate amplifications are worthy of special attention here. First, due to the simplifications applied in the generation process, the produced candidate amplifications may not be able to help the test cases cover their associated targets. Prefest therefore runs the input test cases under the candidate amplifications to dynamically check whether they work as expected. This process is referred to as *amplification validation*. Second, candidate amplifications essentially define partial configurations. When validating an amplification with partial configuration ϕ'', Prefest sets the values of all preferences specified by ϕ'' accordingly, while reusing the existing values for the other preferences defined in the app. Since different reused preference values may lead to distinct actual executions of the app, to make sure we can reproduce the observed executions after amplification validation, Prefest always records the values of all app preferences at the beginning of each amplified test. Third, all assertions in the original test cases are disabled in the amplified test cases because the assertions become obsolete in those test cases. As explained above, Prefest makes sure that each amplified test case covers some targets that were not covered by the original test cases. Given that the amplified test cases execute under different configurations and exercise distinct behaviors, keeping the old assertions most likely will do more harm than good. Therefore, we decided to discard those assertions. We leave a systematic evaluation of the new configurations' impact on the existing assertions and the generation of new assertions for the amplified test cases for future work, since the former involves investing a considerable amount of time and human effort to understand the apps, the tests, and the assertions, while the latter is a long-standing challenge in automated test case generation.

5.3.3 Test Case Amplification in Iterations. Prefest aims to amplify the input test cases for P with a group of configurations that can help the test cases cover more target branches. To keep the number of produced result configurations small, Prefest adopts a greedy strategy so that amplifications that are more likely to help cover more target branches get generated earlier. Algorithm 4 shows how Prefest achieves that in iterations.

The algorithm uses three tables *configs*, *allByPassed*, and *allCovered* as both the input and the output parameters, where *configs* maps each test case to the set of configurations to be used to amplify the test case, *allBypassed* maps each amplified test case to the set of targets bypassed by the test case and the constraints associated with those targets, while *allCovered* maps each amplified test case to the set of targets it covers. When the algorithm starts, the input parameters contain information gathered from amplifying each test case with the default configuration. Afterwards, Prefest generates more amplifications in an iterative fashion. In each iteration, it first collects a sorted set *targetBranches* of targets that are bypassed by existing amplified tests (Line 3), with targets that are bypassed by more existing amplified test cases appearing earlier in set *targetBranches*. Next, a nested while loop that implements a greedy algorithm is employed so that new configurations that are more likely to help cover more bypassed targets are generated earlier (Lines 5 through 23). The outer loop terminates if the inner loop cannot amplify the test cases to cover any previously bypassed target (Lines 4, 19, and 23).

Each iteration of the inner while loop first finds a set *curTargets* of targets that are bypassed by a group of amplified tests with compatible constraints and stores the found tests and their constraints in *curTestsAndConstraints* (Lines 6 through 12). From the group of tests found in this way, Prefest then selects the one that has the least number of covered targets (Line 13), solves its associated constraint and constructs a suitable candidate amplification based on the solution (Line 14), and executes the test with the constructed amplification (Line 15). Note that, during the execution of the amplified test case, Prefest records the target branches newly covered and bypassed. If the execution covers *some* previously bypassed targets (Line 16), the covered ones are removed from *targetBranches* (Line 17), while *allCovered*, *allBypassed*, and *configs* are updated accordingly (Lines 18). Otherwise, all branches in *curTargets* are removed from *targetBranches* and added to *allFailedTargetBranches* (Lines 21 and 22). The reason for such design is that, some target branches may not be reachable by manipulating just preference values, e.g., because their branch conditions rely on specific user inputs or states of external environment; By excluding such targets, Prefest avoids spending a long time on the unreachable targets.

In our previous work [35], test case amplification to cover the previously bypassed targets was done in one pass, instead of in iterations, and a retry mechanism was installed in order to reattempt the targets that the tool failed to cover in the only pass. In the latest implementation of Prefest, all targets that the tool failed to cover during an iteration are collected into set *allFailedTargetBranches*

Algorithm 4: How Prefest amplifies test cases to cover more target branches in iterations.

Input: A table *allBypassed* that maps each amplified test case to the associated set of $\langle target, constraint \rangle$ pairs.
Input: A table *allCovered* that maps each amplified test case to the set of branches it covers.
Input: A table *configs* that maps each test case to the set of configurations to be used to amplify the test case.

```
1  allFailedTargetBranches ← ∅;
2  while true:
3      targetBranches ← getBypassedByAll(configs, allBypassed, allCovered);
4      hasProgress ← false;
5      while ¬targetBranches.isEmpty():
6          curTargets←{targetBranches[0]};
7          curTestsAndConstraints ← getTestsAndConstraints(targetBranches[0]);
8          foreach 1 ≤ i <targetBranches.size():
9              tmpTestsAndConstraints ← getTestsAndConstraints(targetBranches[i]);
10             if isCompatible(curTestsAndConstraints, tmpTestsAndConstraints):
11                 curTargets ← curTargets ∪ {targetBranches[i]};
12                 curTestsAndConstraints.merge(tmpTestsAndConstraints);
13         test, constraint ← getWithLeastCovered(curTestsAndConstraints);
14         newConfig ← solve(test, constraint);
15         covered', bypassed' ← execute(test, newConfig);
16         if covered' ∩ targetBranches ≠ ∅:
17             targetBranches.remove(covered');
18             update(test, newConfig, covered', bypassed', allCovered, allBypassed, configs);
19             hasProgress ← true;
20         else:
21             targetBranches.remove(curTargets);
22             allFailedTargetBranches.add(curTargets);
23     if ¬hasProgress: break;
24 foreach failedTargetBranch ∈ allFailedTargetBranches \ allCovered:
25     tmpTestsAndConstraints ← getTestsAndConstraints(failedTargetBranch);
26     test, constraint ← getWithLeastCovered(tmpTestsAndConstraints);
27     newConfig ← solve(test, constraint);
28     covered', bypassed' ← execute(test, newConfig);
29     if failedTargetBranch∈covered':
30         update(test, newConfig, covered', bypassed', allCovered, allBypassed, configs);
```

(Line 22) and the same retry mechanism is applied to reattempt those targets after the iteration has terminated (Lines 24 through 30). Particularly, for each target to reattempt (Line 24), Prefest gathers the tests that have bypassed the target and the constraints under which the tests may be able to cover the target (Line 25), selects the test that has the least number of covered targets (Line 26), constructs a new amplification with which the test might cover the current target (Line 27), and executes the test with that amplification (Line 28). If the current target is actually covered by the execution, the full configuration used to run the test is reported (Line 29 and 30). Compared with amplifying the test cases in iterations, doing that in one pass has the drawback that targets at locations not covered by any initial test executions will be completely ignored by the amplification process.

5.4 System Preference Analysis

The description above mainly focuses on app preferences that users can exploit to customize app behaviors and features. The Android platform also provides a set of preferences that enables such customization at the system level. By interpreting the related accessing API methods, Prefest also supports the handling of six system preferences that are often used, namely `WiFi`, `bluetooth`, `mobile data`, `GPS locating`, `network locating` and `music playing`. Conceptually, Prefest treats these system preferences as shared by all the Android apps and handles them in the same way as it handles app preferences. Therefore, we do not differentiate system and app preferences when describing the experiments we conducted to evaluate Prefest or reporting the experimental results in Section 6.

5.5 Implementation

We implemented our approach into a tool, also named Prefest. Taking as the input an Android app and a group of test cases, Prefest amplifies the input test cases with necessary configurations to exercise more target branches in the app. At the moment, Prefest accepts input test cases written in the Espresso[3] or Appium[4] format, be they manually written by developers or automatically generated by testing tools. The Espresso testing framework, provided as part of the official Android development toolkit, defines APIs for writing UI tests that simulate user interactions within Android apps, while Appium is a widely-used open source test automation framework, which allows users to test native Android apps without the SDK or rebuilding their apps. Although steps like test-relevant preference identification in preference-wise testing with Prefest can be integrated into an automated test generation tool to slightly enhance the efficiency of test case amplification, the integration will cause Prefest to be tightly bound with the other tools and limit its applicability. We therefore design Prefest to take existing test cases as the input. Test cases generated by tools like *AndroidRipper* [3], A^3E [7] and *Stoat* [54] can be easily translated into the Espresso and/or Appium format and then fed to Prefest.

In its current implementation, Prefest supports preferences of types `SwitchPreference`, `CheckBoxPreference`, `ListPreference`, `EditTextPreference` and `SeekbarPreference`. Compared with our previous work [35], the support for `SeekbarPreference` was added since `SeekbarPreference`s are widely used in Android apps (see Section 4). We leave adding the support for other types of preferences into Prefest for future work.

At the moment, amplification test actions generated by Prefest open anchor activities via the Android Debugging Bridge command, but activities with restricted permissions are not directly launchable after Android 7. There are two possible ways to enable Prefest to work even in the face of the restriction. First, if Prefest is used by the developers of an app, they may build a special version of the app where the "exported" attributes of the anchor activities are set to `true` to lift the restriction. Second, information about the steps needed for opening each anchor activity could be provided as extra input to Prefest. While providing such extra information increases the costs of using Prefest, the additional costs are most likely moderate and acceptable to the users.

[3]https://developer.android.com/training/testing/espresso
[4]https://http://appium.io/

6 EVALUATION

To gain first-hand knowledge about how Prefest supports preference-wise testing in practice, we conducted comprehensive experiments where Prefest is applied to amplify tests for a wide range of Android apps. Our experiments aim to answer the following research questions:

RQ1: How effective is Prefest? The main purpose of Prefest is to automatically amplify existing tests with new configurations so that they exercise more preference-related behaviors of Android apps. In RQ1, we assess to what extent amplifications produced by Prefest can help existing tests cover more code and detect more bugs that are related to preferences;

RQ2: How efficient is Prefest? In RQ2, we study the costs in time for test amplification with Prefest;

RQ3: How does important design decisions in Prefest affect its effectiveness and efficiency? In RQ3, we examine how and to what extent test-relevant preference identification and target-oriented test amplification affect Prefest's effectiveness and efficiency.

RQ4: How well does Prefest work in amplifying test cases manually prepared by programmers? We use automatically generated test cases as seeds for test amplification in answering RQ1, RQ2, and RQ3 so as to guard against possible bias in selection of apps and their test cases. In RQ4, we study how well Prefest works on test cases that are manually prepared by programmers.

6.1 Subject Apps and Tests

We prepared subject apps in two phases, with the first phase focusing on selecting apps for answering research questions RQ1, RQ2, and RQ3, and the second phase focusing on selecting apps for answering research question RQ4.

More concretely, we constructed in the first phase an initial pool that contains 340 apps from both previous research [43, 49, 52] and a popular list of open-source Android apps on GitHub [46]. Among those apps, 66 were excluded because they are duplicates or missing, 26 were excluded because they are too old to compile, 145 were excluded because they contain fewer than five preferences, and 19 were excluded because they are not compatible with Android API level 19, and 44 were excluded because Stoat achieved instruction coverage lower than 20% on those apps. Here, the requirement for compatibility with Android API level 19 was critical in our experiments, since we employed the Stoat tool to automatically generate GUI tests for Android apps, and Stoat produces the best test generation results at Android API level 19. We decided to utilize Stoat to prepare the subject tests for our experiments because it was a state-of-the-art automated Android test generation tool and it was utilized in various previous research studies [20, 23, 45, 57] to prepare GUI tests for Android apps. We then applied Stoat to generate tests for the remaining 84 apps. Each run of *Stoat* on a particular app took two hours, with one hour being spent on GUI exploring and the other on Markov Chain Monte Carlo (MCMC) sampling; Each test suite generated by Stoat contained at most 30 tests and each test has at most 30 events in one sampling iteration. Based on the test generation results, 13 apps were excluded since Stoat crashed for over 30 times during the 1 hour GUI exploration, 31 apps were excluded since the tests generated for each of those apps covered less than 20% of the app's code. In this way, we were left with 40 apps at the end of the first phase. Among those 40 apps, 28 were available in the Google Play store [17] and 8 of them were popular apps each with over 1 million downloads. Only 5 apps among the 40, however, had more than 3 programmer-written GUI test cases.

To gather more subjects to be used in addressing RQ4, we looked for additional apps with more than a couple programmer-written GUI test cases on GitHub in the second phase. Particularly, we first searched GitHub for projects with keywords "android application" and "android app" and then

Table 6. Subject apps and their test cases automatically generated by Stoat. For each app, its id, version, the numbers of lines of code (#i), branches (#b), and defined preferences (#p) in the app, the number of test cases generated by Stoat for the app (#tc), and the following information about the generated test cases: The instruction (%i) and branch (%b) coverage achieved, the number of relevant preferences (#rp) and the ratio of that number to #p (%rp), the number of collected target branches (#tb), the number of covered target branches (#tbc) and the ratio of that number to #tb (%tbc), and the time in minutes needed to run the tests (t). Note that #rp may be larger than #p, leading to %rp values greater than 100%, if the tests access also system preferences.

ID	APP	VER	#I	#B	#P	Stoat								
						#TC	%I	%B	#RP	%RP	#TB	#TBC	%TBC	T
A01	a2dpvolume	2.13.0.4	18324	1443	15	180	40.0%	17.4%	7	46.7%	48	27	56.3%	73.4
A02	Adaway	4.3.0	16189	1162	19	150	41.1%	28.9%	9	47.4%	18	14	77.8%	71.3
A03	Alwayson	0.9.5.2	15379	1335	29	240	44.5%	30.7%	24	82.8%	72	57	79.2%	120.7
A04	AmazeFileManager	3.3.0-rc1	80767	6735	31	210	20.6%	15.0%	19	61.3%	88	45	51.1%	137.8
A05	amme	2018.11.15.987	93689	6449	34	180	25.3%	17.9%	9	26.5%	39	18	46.2%	105.2
A06	Anki-Android	2.9-alpha-54	128890	10155	96	210	33.1%	22.7%	19	19.8%	72	36	50.0%	144.8
A07	AntennaPod	1.7.1	38421	3431	34	180	33.0%	23.5%	14	41.2%	44	26	59.1%	73.4
A08	APhotoManager	0.7.2.181027	47512	4889	25	150	39.0%	26.8%	19	76.0%	154	80	51.9%	71.1
A09	avare	8.1.2	138689	8610	83	120	30.2%	12.5%	46	55.4%	132	77	58.3%	70.3
A10	CalendarNotification	5.0.5	51317	3319	43	240	46.4%	34.4%	21	48.8%	52	27	51.9%	95.7
A11	commons	2.9	42439	2490	5	270	25.8%	15.0%	2	40.0%	10	8	80.0%	101.5
A12	connectbot	1.9.5-35	70864	3756	26	390	23.8%	22.5%	15	57.7%	48	26	54.2%	127.9
A13	FanFictionReader	1.58a	27910	1946	7	240	31.8%	21.9%	5	71.4%	26	14	53.8%	94.5
A14	Fillup	1.7.2	18141	3932	6	210	21.1%	16.5%	4	66.7%	32	8	25.0%	63.2
A15	forecast	1.6.2	8331	532	15	300	57.9%	40.8%	16	106.7%	74	39	52.7%	137.7
A16	good-weather	4.4	11195	678	11	360	59.9%	34.4%	8	72.7%	22	11	50.0%	137.2
A17	hn-android	5.0	14790	1298	5	240	47.8%	34.3%	5	100.0%	37	18	48.6%	80.5
A18	KISS	3.7.2	20003	1823	36	360	47.4%	34.3%	23	69.4%	83	48	43.2%	150.7
A19	materialistic	3.2	30085	2587	37	210	44.3%	26.1%	17	45.9%	66	43	65.2%	113.1
A20	movianremote	1.1.0	874	40	8	180	77.6%	55.0%	0	0.0%	0	0	0.0%	119
A21	mupen64plus	3.0.87-beta	58716	5080	165	330	28.7%	17.4%	30	18.2%	102	60	58.8%	197.9
A22	MyExpenses	r410	150889	13245	36	390	19.7%	10.3%	18	50.0%	49	28	57.1%	223.3
A23	nanoConverter	0.7.87	8508	527	6	210	32.8%	31.9%	7	116.7%	68	49	72.1%	121.6
A24	nextcloud	3.12.0	184155	15348	6	300	18.5%	12.1%	0	0.0%	0	0	0.0%	109.2
A25	nextcloud-news	0.9.9.36	36241	2112	18	420	23.9%	13.1%	6	33.3%	23	12	52.2%	100.7
A26	NotePad	1.0.2	7722	753	6	321	52.0%	39.8%	6	100.0%	133	82	61.7%	104.4
A27	Omni-Notes	5.4.4	34993	2895	23	330	25.5%	21.7%	10	43.5%	38	19	50.0%	88.5
A28	OpenBikeSharing	1.10.0	5529	363	5	180	58.8%	47.9%	4	80.0%	31	24	77.4%	91.5
A29	openhab	2.12.18	64089	6175	20	390	27.5%	18.2%	10	50.0%	32	16	50.0%	168.8
A30	opensudoku	2.5.2	17606	1328	14	120	44.6%	32.5%	10	71.4%	36	23	63.9%	65.3
A31	Radiobeacon	0.8.18	36252	2171	20	180	37.2%	19.9%	11	55.0%	44	31	70.5%	117.4
A32	RedReader	1.9.9	59012	4984	62	240	32.8%	24.9%	27	43.5%	83	35	42.2%	77.2
A33	reference-browser	1.0.2047	18771	896	7	390	27.1%	5.2%	2	28.6%	4	2	50.0%	94.5
A34	runnerup	1.90.1	75863	5878	67	150	20.2%	13.7%	10	14.9%	26	14	53.8%	92.7
A35	Signal-Android	4.33.0	237215	16644	40	450	30.6%	14.3%	17	42.5%	52	28	53.8%	245.9
A36	smsdroid	1.7.7	12709	957	33	360	43.0%	30.0%	14	42.4%	26	15	57.7%	139.5
A37	SuntimesWidget	0.13.4	148810	11221	44	180	24.6%	15.2%	21	47.7%	83	43	51.8%	81.7
A38	TapAndTurn	2.8.0	3043	270	5	300	53.7%	29.6%	1	20.0%	2	1	50.0%	116.6
A39	Timber	1.6	55262	4307	11	240	25.4%	15.1%	7	63.6%	30	16	53.3%	89.5
A40	TintBrowser	1.8.1	33074	7558	22	210	27.0%	16.2%	9	40.9%	32	19	59.4%	65.1
A41	uhabit	1.7.9	20107	1637	11	300	54.8%	28.5%	2	18.2%	6	3	50.0%	100.7
A42	vanilla	1.0.80	48501	4801	36	270	45.5%	35.2%	29	80.6%	98	48	49.0%	109.7
A43	websms	4.9.5	10839	1030	35	390	33.0%	26.9%	22	62.9%	47	37	78.7%	211.5
A44	WhereYouGo	0.9.3	27066	2283	47	150	38.6%	26.7%	16	34.0%	40	25	62.5%	71.3
A45	WikiPedia	2.7.237	105509	7334	53	180	43.1%	27.3%	23	43.4%	62	36	58.1%	122.7
Overall		–	2333635	186407	1357	11301	30.1%	19.0%	594	43.9%	2264	1288	56.9%	5102.9

gathered all the matched projects with over 50 stars. This produced a list of 1857 apps. Among those apps, 127 were excluded because they were duplicates or unavailable for download, 1697 were excluded because they each had fewer than 3 programmer-written GUI test cases, 12 were excluded because they contained fewer than five preferences, 6 were excluded because over 80% of their test cases were out of date and did not match the apps' source code, and 5 were excluded because they cannot be compiled due to problems like missing license key or missing Google Play service key. Since only ten apps were left after the pruning, we slightly relaxed the selection criteria, i.e., we no longer required compatibility with Android API level 19, and retained all the ten apps. Although we started with 1857 apps in total, we were left with only 10 apps at the end of the second phase mainly because very few Android apps on GitHub have more than a couple of GUI test cases. This phenomena, however, was in line with the observations made by Pecorelli et al. [47].

Since the two sets of apps gathered in the two phases had 5 apps, namely *AntennaPod*, *connectbot*, *KISS*, *Omni-Notes* and *SuntimesWidget*, in common, we collected in total 45 subject apps in the end, and we also applied the same process as described above to generate test cases for the 5 new apps found in the second phase using Stoat. Table 6 lists the 45 apps. For each app, the table provides basic information about both the app and the tests generated by Stoat for it. The size of the apps varies between 874 and over 237K instructions, or between 40 and over 16K branches[5], and the number of preferences defined in those apps ranges between 5 and 165. There is therefore a considerable amount of diversity with the subjects, making our experiments representative of Prefest's behaviors on a wide range of apps. The 11,301 initial test cases generated by Stoat take in total 5,102.9 minutes to run. The overall instruction and branch coverage achieved by the tests was 30.1% and 19.0%, respectively. More importantly, while the apps have defined 1,357 preferences in total, only 594, or 43.9%, of them were actually relevant to the generated tests, which confirms our observation explained in Section 1 that each test case typically interacts with only a small number of preferences defined in the app. When executed under the default preference settings, the generated tests revealed in total 2264 target branches and covered 1288, or 56.9%, of those target branches.

For each subject app with at least three programmer-written GUI test cases, Table 7 lists the basic information about the app and its associated programmer-written tests. Overall, programmers have written in total 360 tests for the apps, and most characteristics of the programmer-written tests were comparable to those of the automatically generated tests. The size of these apps varies between around 18.7K and over 184.1K instructions, or between 896 and over 15.3K branches, and their numbers of defined preferences range between 6 and 44; While the apps have defined 250 preferences in total, only 120, or 48%, of them were relevant to the tests; When executed under the default preference settings, the generated tests revealed in total 418 target branches and covered 243, or 58.1%, of those target branches. However, the tests cover only 25.5% and 15.5% of the app instructions and branches, respectively. While test cases with lower coverage may be common for open-source Android apps, we expect the test cases for commercial apps to have much better quality and achieve much higher coverage. Using these apps and their programmer-written test cases as the subject therefore constitutes a major threat to the external validity of our findings with respect to RQ4. We discuss further this threat in Section 6.4.

To understand to what extent programmers test preference-related code in their apps, we manually examined the tests that explicitly mention "preference" in their names—we refer to such tests as preference-oriented tests. We make the following two observations. First, 119, or 33.1%, of the programmer-written tests were actually preference-oriented, and all of the ten apps,

[5]The JaCoCo library (https://www.eclemma.org/jacoco/) was employed in this work to measure app size as well as instruction and branch coverage of tests.

Table 7. Subject apps with at least three programmer-written GUI test cases. For each APP, the basic information about both the app and its associated programmer-written test cases, as was reported in Table 6, as well as the number of preference-oriented tests (#TC') and the number of preferences relevant to those preference-oriented test cases (#RP').

ID	APP	VER	#I	#B	#P	Programmer-written										
						#TC	%I	%B	#RP	%RP	#TB	#TBC	%TBC	T	#TC'	#RP'
A07	AntennaPod	1.7.1	38421	3431	34	54	24.6%	16.2%	15	44.1%	53	36	67.9%	13.2	29	10
A12	connectbot	1.9.5-35	70864	3756	26	10	22.1%	18.2%	13	50.0%	38	20	52.6%	0.9	0	0
A18	KISS	3.7.2	20003	1823	36	16	32.0%	21.3%	21	58.3%	64	33	51.6%	0.3	2	21
A22	MyExpenses	r410	150889	13245	36	81	29.5%	16.7%	20	55.6%	62	34	54.8%	3.8	14	7
A24	nextcloud	3.12.0	184155	15348	6	36	18.8%	10.4%	0	0.0%	0	0	0.0%	2.3	0	0
A25	nextcloud-news	0.9.9.36	36241	2112	18	17	31.0%	22.1%	11	61.1%	31	16	51.6%	1.5	5	7
A27	Omni-Notes	5.4.4	34993	2895	23	15	25.1%	19.5%	9	39.1%	36	18	50.0%	1.0	3	4
A29	openhab	2.12.18	64089	6175	20	7	24.7%	15.2%	7	35.0%	22	13	59.1%	0.7	2	3
A33	reference-browser	1.0.2047	18771	896	7	21	39.8%	15.8%	2	28.6%	4	2	50.0%	2.8	6	2
A37	SuntimesWidget	0.13.4	148810	11221	44	103	27.9%	16.7%	22	50.0%	108	71	65.7%	5.0	58	15
Overall		-	767236	60902	250	360	25.5%	15.5%	120	48.0%	418	243	58.1%	31.5	119	69

except *connectbot* and *nextcloud*, had at least one preference-oriented test, which suggests that, when programmers do write test cases for their apps, they tend to take preferences into account. Second, most of the preference-oriented tests are superficial in the sense that they seldom check how preferences affect app behaviors, which suggests that, even if programmers are aware of the importance of preference-wise testing, the tests they write are insufficient for detecting bugs related to preferences. In particular, among the 119 preference-oriented test cases, 13 (2 from *KISS*, 2 from *MyExpense*, 3 from *Omni-Note* and 6 from *reference-browser*) only navigated to, but never interacted with, the settings screens, and 77 (26 from *AntennaPod* and 51 from *SuntimesWidget*) simply checked read-/write-access to the preference values, without exercising other preference-related behaviors directly. Such results emphasize that Android developers are in dire need of help to conduct effective preference-wise testing.

6.2 Experimental Setup

To answer RQ1 and RQ2, we applied Prefest to amplify the tests automatically generated by Stoat for the 45 subject apps.

To answer RQ3, we first modified Prefest to produce three variant test amplification tools as the following:

(1) Prefest-cr works in the same way as Prefest except that it performs 2-way combinatorial testing over preferences relevant to each existing amplified test, instead of conducting target-oriented test amplification; That is, given the initial amplified test $\langle t, \phi_0 \rangle$, Prefest-cr generates a group Φ of configurations for t such that $\forall \delta_1, \delta_2 \in R_{\langle t, \phi_0 \rangle} : \forall v_1 \in \delta_1.entryValue, v_2 \in \delta_2.entryValue : \exists \phi' \in \Phi : \phi'(\delta_1) = v_1 \wedge \phi'(\delta_2) = v_2$.

(2) Prefest-ca also performs 2-way combinatorial testing in amplifying tests, but it generates a group of configurations for a test such that each pair of all preference values is included at least once; That is, given the amplified test $\langle t, \phi_0 \rangle$, Prefest-ca generates a group Φ' of configurations for t such that $\forall \delta_1, \delta_2 \in \Delta : \forall v_1 \in \delta_1.entryValue, v_2 \in \delta_2.entryValue : \exists \phi' \in \Phi' : \phi'(\delta_1) = v_1 \wedge \phi'(\delta_2) = v_2$.

(3) Prefest-nd is much less sophisticated and amplifies each input test case with a single configuration where each preference is randomly set to a non-default value.

We then compared the test amplification results produced by Prefest and the three variants on 13 subjects, including 8 apps that we investigated already in our previous work [35] and 5 extra apps where Prefest was able to detect bugs. Given that the total number of 2-way combinations of preference values can be astronomical, especially when some preferences that may take a relatively large amount of possible values are present, in our previous work [35] we applied Prefest-ca and Prefest-cr on only 8 subject apps, namely, *a2dpvolume* (A01), *Alwayson* (A03), *good-weather* (A16), *Notepad* (A23), *opensudoku* (A26), *Radiobeacon* (A27), *SuntimesWidget* (A32), and *Wikipedia* (A40), and considered at most two possible values for each preference when building the 2-way combinations, with one being the default value and the other being randomly selected from the remaining entry values. In this work, we also applied these tools to the subject apps where Prefest can detect bugs and compared the tools in terms of their capability to detect bugs.

To answer RQ4, we applied Prefest to amplify the tests written by programmers for the 10 subject apps listed in Table 7.

All experiments were conducted on a desktop machine with 8GB RAM and 2.0GHz quad-core processor running Windows 10, and the Android emulator to run the test cases of apps was configured with 2GB RAM and the X86 ABI image.

6.3 Experimental Results

This section reports on the results from experiments.

6.3.1 RQ1: Effectiveness. Table 8 lists, for each subject app, the basic information about the amplification results produced by Prefest, and the box plot in Figure 7 shows the distribution of improvements Prefest achieved in terms of various metrics. In total, Prefest generated 919 amplified test cases for the apps, averaging to 20.4 amplified test cases per app.

The amplified test cases helped raise the overall instruction and branch coverage by 9.3% and 15.3%, respectively. Considering that Stoat is a state-of-the-art test generation tool for Android apps, the improvement in code coverage achieved by Prefest is significant. In particular, improvements of over 10% in instruction and branch coverage were observed on 21 and 31 apps, respectively, and improvements of less than 3% in both instruction and branch coverage were only observed on 4 apps, namely apps *commons* (A11), *movianremote* (A20), *nextcloud* (A24) and *uhabit* (A41). *movianremote* accesses the values of its preferences via inter-component communication, instead of by calling methods on the shared `SharedPreferences` singleton object, but Prefest does not handle such special mechanism in its current implementation; No target branches were revealed by the input test cases on app *nextcloud*; As for apps *commons* and *uhabit*, the improvements were small partly because their input test cases have only a few relevant preferences and partly because the amplified test cases for those apps share most behaviors with the original test cases. It is also worth mentioning that, for apps *Anki-Android* (A06), *avare* (A09), *Signal-Android* (A35) and *SuntimesWidget* (A37), although the improvements of 14.8%, 12.5%, 9.2%, 11.7% in instruction coverage and 17.6%, 20.2%, 17.5% and 21.2% in branch coverage may not seem huge, the absolute numbers of additional instructions and branches that are covered by the amplified test cases are actually substantial, considering that each of these apps has more than 100K instructions and 10K branches.

Although the number of relevant preferences did not change after amplification on 24 apps, that number did increase by more than 10% on 14 apps, achieving an overall improvement of 9.8%. Such significant increase indicates that the amplifications enabled the test cases to check the influence of quite some additional preferences on app behaviors; Meanwhile, the amplified test cases raised the percentage of covered target branches on all but two apps, achieving an overall improvement of 49.9%. The huge raise suggests that the amplifications enabled the test cases to exercise the different choices at various target branches much more thoroughly. Generally speaking, we believe

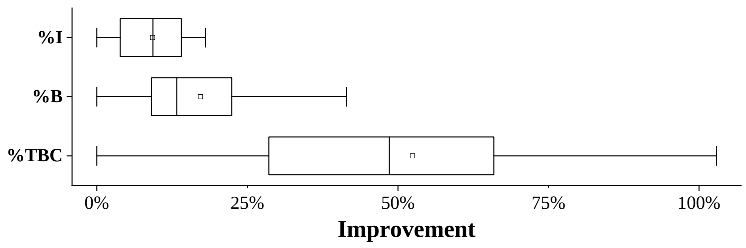


Fig. 7. Distribution of the improvements achieved by Prefest in terms of instruction (%i), branch (%b), and target branch (%tbc) coverage.

Table 8. Test amplification results produced by PREFEST on automatically generated test cases. For each app, its ID, the number of amplified test cases generated (#TC), the information about the whole suite of test cases after amplification (including %I, %B, #RP, #TB, and %TBC, as reported in Table 6 for tests generated by STOAT), and the information about the amplification process with PREFEST: The number of iterations PREFEST went through in amplifying the test cases (#ITE), the amplification time in minutes (T), and the breakdown of that to the time spent on test-relevant preference discovery (T_D), preference locator discovery (T_L), and target-oriented test amplification (T_A). Values in the brackets are calculated as $(m_a - m_b)/m_b$, where m_b and m_a are the measurements before and after test amplification, and they indicate how much PREFEST has improved the measurements.

ID	#TC	%I	%B	#P_R	#TB	%TBC	#ITE	T	T_D	T_L	T_A
A01	10 (5.6%)	41.6 (4.0%)	20.7 (19.0%)	8 (14.3%)	60 (25.0%)	93.3 (65.9%)	3	87.0	73.9	2.8	10.3
A02	4 (2.7%)	41.8 (1.7%)	30.4 (5.2%)	9 (0.0%)	18 (0.0%)	100.0 (28.6%)	1	77.7	71.7	3.4	2.6
A03	19 (7.9%)	48 (7.9%)	34.5 (12.4%)	24 (0.0%)	74 (2.8%)	93.2 (17.8%)	2	152.3	121.4	7.8	23.1
A04	29 (13.8%)	23.7 (15.0%)	17.7 (18.0%)	21 (10.5%)	94 (6.8%)	83.0 (62.3%)	2	202.6	141.8	5.1	55.7
A05	21 (11.7%)	27.6 (9.1%)	19.8 (10.6%)	10 (11.1%)	64 (64.1%)	89.1 (93.0%)	3	142.8	105.9	10.5	26.4
A06	34 (16.2%)	38 (14.8%)	26.7 (17.6%)	28 (47.4%)	94 (30.6%)	84.0 (68.1%)	3	228.5	145.7	19.1	63.7
A07	20 (11.1%)	35.1 (6.4%)	26.4 (12.3%)	15 (7.1%)	47 (6.8%)	74.5 (26.0%)	2	102.4	74.7	5.9	21.8
A08	37 (30.8%)	43.4 (11.3%)	30.2 (12.8%)	46 (0.0%)	146(10.6%)	93.2 (59.7%)	2	169.0	71.3	10.6	87.1
A09	14 (5.8%)	34.0 (12.5%)	15.0 (20.2%)	24 (14.3%)	65 (25.0%)	89.2 (71.9%)	2	134.9	96.2	8.9	29.8
A10	26 (17.3%)	48.8 (5.0%)	38.6 (9.1%)	19 (0.0%)	164(6.5%)	92.1 (77.2%)	3	117.1	71.7	3.1	42.3
A11	3 (1.1%)	26 (0.8%)	15.3 (2.0%)	2 (0.0%)	10 (0.0%)	100.0 (25.0%)	1	104.2	102.5	0.8	0.9
A12	13 (3.3%)	23.9 (0.6%)	23.2 (3.2%)	15 (0.0%)	48 (0.0%)	87.5 (61.5%)	1	147.8	129.1	0.6	18.1
A13	18 (7.5%)	37.4 (17.6%)	27.4 (25.1%)	5 (0.0%)	34 (30.8%)	70.6 (31.1%)	2	111.4	95.3	1.4	14.7
A14	17 (8.1%)	24.3 (15.2%)	20.2 (22.4%)	4 (0.0%)	33 (3.1%)	90.9 (263.6%)	2	86.1	63.6	0.5	22
A15	52 (17.3%)	64.9 (12.1%)	57.7 (41.4%)	17 (6.3%)	110(48.6%)	80.0 (51.8%)	3	203.6	138.7	2	62.9
A16	10 (2.8%)	68.6 (14.5%)	48.4 (40.7%)	8 (0.0%)	22 (0.0%)	81.8 (63.6%)	1	152.6	137.7	2.1	12.8
A17	14 (5.8%)	54.5 (14.0%)	37.9 (10.5%)	5 (0.0%)	45 (21.6%)	82.2 (69.0%)	2	101.3	82.0	1	18.3
A18	31 (8.6%)	54.7 (15.4%)	43.1 (25.7%)	29 (26.1%)	111(33.7%)	91.9% (58.9%)	2	218.4	154.8	5.7	57.9
A19	32 (15.2%)	52.3 (18.1%)	34.3 (31.4%)	22 (29.4%)	84 (27.3%)	81.0 (24.3%)	3	180.3	113.8	7.4	59.1
A20	0 (0.0%)	77.6 (0.0%)	55 (0.0%)	0 (0.0%)	0 (0.0%)	0.0 (0.0%)	1	119.3	119.3	0	0
A21	40 (12.1%)	29.3 (2.1%)	18.7 (7.5%)	31 (3.3%)	128(25.5%)	86.7 (47.4%)	3	311.7	200.0	8.9	102.8
A22	18 (4.6%)	21.8 (10.5%)	11.7 (13.3%)	18 (0.0%)	53 (8.2%)	75.5 (32.1%)	2	267.9	224.2	7.2	36.2
A23	19 (9.0%)	36.7 (11.9%)	39.1 (22.6%)	7 (0.0%)	70 (2.9%)	92.9 (28.9%)	2	148.4	121.9	6.2	20.3
A24	0 (0.0%)	18.5 (0.0%)	12.1 (0.0%)	0 (0.0%)	0 (0.0%)	0.0 (0.0%)	0	113.2	111.2	0	2.0
A25	12 (2.9%)	26.5 (10.9%)	15.1 (15.4%)	7 (16.7%)	28 (21.7%)	67.9 (30.1%)	2	113.8	101.2	2.0	10.5
A26	61 (29.0%)	55.6 (6.9%)	49.3 (23.9%)	6 (0.0%)	159(19.5%)	76.7 (24.5%)	3	163.9	104.9	0.9	58.1
A27	22 (6.7%)	26.8 (5.1%)	22.7 (4.7%)	12 (20.0%)	41 (7.9%)	58.5 (17.1%)	2	111.0	89.3	4.8	16.9
A28	22 (12.2%)	60.2 (2.4%)	50.4 (5.2%)	4 (0.0%)	31 (0.0%)	77.4 (0.0%)	1	113.8	92.0	0.8	21
A29	17 (4.4%)	32.2 (17.3%)	22.1 (21.6%)	11 (10.0%)	48 (50.0%)	77.1 (54.2%)	2	200.4	170.6	2.7	27.1
A30	17 (14.2%)	47.5 (6.5%)	37.4 (15.1%)	11 (10.0%)	52 (44.4%)	92.3 (44.5%)	2	91.4	66.0	2.6	22.8
A31	21 (11.7%)	40.9 (9.9%)	21.9 (10.1%)	12 (9.1%)	50 (13.6%)	82.0 (16.4%)	2	143.7	118.0	2.2	23.5
A32	38 (15.8%)	36.9 (12.5%)	31.4 (26.1%)	27 (0.0%)	83 (0.0%)	85.5 (102.9%)	2	134.1	79.1	13.1	41.9
A33	5 (1.3%)	30.7 (13.3%)	9.8 (88.8%)	2 (0.0%)	12 (200.0%)	83.3 (66.7%)	2	100.1	95.5	1.1	3.6
A34	11 (7.3%)	21 (4.0%)	14.9 (8.8%)	10 (0.0%)	26 (0.0%)	88.5 (64.3%)	1	115.8	93.5	11.1	11.2
A35	27 (6.0%)	33.4 (9.2%)	16.8 (17.5%)	19 (11.8%)	62 (19.2%)	75.8 (40.8%)	2	299.5	256.5	7.3	35.7
A36	12 (3.3%)	44.2 (2.8%)	33.5 (11.7%)	24 (71.4%)	70 (169.2%)	95.7 (65.9%)	2	166.5	139.9	6.7	19.9
A37	36 (20.0%)	27.4 (11.7%)	18.4 (21.2%)	21 (0.0%)	95 (14.5%)	73.7 (42.2%)	2	136.1	83.6	3.0	49.5
A38	3 (1.0%)	55.7 (3.7%)	32.2 (8.8%)	1 (0.0%)	2 (0.0%)	100.0 (100.0%)	1	120.3	117.0	2.7	0.6
A39	5 (2.1%)	26.4 (3.9%)	16.7 (10.6%)	7 (0.0%)	30 (0.0%)	100.0 (87.5%)	1	98.6	90.8	2.6	5.2
A40	21 (10.0%)	35.2 (30.4%)	21.3 (31.5%)	14 (55.6%)	51 (59.4%)	82.4 (38.7%)	3	90.6	65.4	4.5	20.7
A41	6 (2.0%)	55.7 (1.6%)	29.1 (2.1%)	3 (50.0%)	8 (33.3%)	75.0 (50.0%)	2	108.0	102.0	2.6	3.4
A42	43 (15.9%)	52.8 (16.0%)	41.9 (19.0%)	29 (0.0%)	113(15.3%)	89.4 (82.5%)	2	197.1	110.7	6.7	79.7
A43	14 (3.6%)	38.7 (17.3%)	35.1 (30.4%)	24 (9.1%)	53 (12.8%)	96.2 (22.2%)	2	240.6	211.9	4.3	24.4
A44	18 (12.0%)	40.6 (5.2%)	29.8 (11.6%)	16 (0.0%)	46 (15.0%)	82.6 (32.2%)	2	92.7	71.8	5.2	15.7
A45	27 (15.0%)	45.9 (6.5%)	29.4 (7.7%)	25 (8.7%)	80 (29.0%)	86.3 (48.5%)	3	187.0	124.8	5.6	56.6
Overall	919 (8.1%)	32.9 (9.3%)	21.9 (15.3%)	652 (9.8%)	2714 (19.9%)	85.3 (49.9%)	-	6713.8	5153.3	222	1338.4

Table 9. Bugs detected by Prefest in the subject apps. For each bug, the ID (id) and name (app) of the app it belongs to, the type of exception triggered by the bug (exception), whether it has been reported before (new), and its corresponding issue URL on GitHub.

ID	APP	EXCEPTION	NEW	URL
A04	AmazeFileManager	BadParcelableException	F	github.com/TeamAmaze/AmazeFi-leManager/issues/1400
A15	forecast	IllegalArgumentException	T	github.com/martykan/forecastie/issues/395
A16	GoodWeather	IllegalArgumentException	T	github.com/qqq3/good-weather/issues/54
A18	KISS	NumberFormatException	T	github.com/Neamar/KISS/issues/1136
A31	Radiobeacon	IntentReceiverLeaked	T	github.com/openbmap/radiocells-scanner-android/issues/223
A40	TintBrowser	NullPointerException	T	github.com/Anasthase/TintBrowser/issues/120
A42	vanilla	IllegalArgumentException	T	github.com/vanilla-music/vanilla/issues/898

Prefest was effective in helping the generated test cases exercise new behaviors that are dependent on preferences.

Regarding the number of iterations Prefest went through for amplifying the test cases, while one iteration was already enough on 9 apps, additional iterations helped Prefest produce more amplified test cases for the other 34 apps, suggesting that the amplifications do make the apps exercise more of their behaviors and many of those behaviors cover target branches that were missed by the input test cases.

Amplified test cases produced by Prefest helped detect 7 bugs, as shown in Table 9. These bugs are all preference related and only cause problems when certain functions of the apps were tested under specific configurations. None of the bugs were detected by Stoat, since Stoat missed the specific values of those relevant preferences or even the settings screens completely. The bug in app `Radiobeacon` caused data leaks, while the others caused crashes. We were able to reproduce all these bugs. Compared with our previous research [35], we found two more bugs in apps `TintBrowser` and `forecast`: The former bug is relevant to a preference type not supported by the previous version of Prefest, while the latter one was found after the first iteration of test amplification. Overall, six of the bugs, i.e., all except the one in `AmazeFileManager`, were reported for the first time. We have submitted these bugs together with the steps to reproduce the failures on GitHub. So far, bugs in apps `KISS`, `vanilla` and `forecast` have been confirmed and fixed by developers of the corresponding apps. Especially, the bug in `vanilla` was an old one introduced over one year ago, and the developers were happy to know the root cause of the bug and be able to fix it. We, however, have not received any response regarding the other three bugs, possibly because the three projects are no longer actively maintained. Nevertheless, given that they caused crashes or data leaks, we are confident they are real bugs.

Compared with our previous work [35], the current implementation of Prefest also handles preferences of type `SeekBarPreference`. Among the 45 subject apps, six apps, namely *alwayson*, *Anki-Android*, *materialistic*, *mupen64plus*, *TintBrowser*, and *vanilla*, contained `SeekBarPreferences`. In our experiments on the six apps, the additional capability of Prefest led to an overall increase of 5.8% and 5% in instruction and branch coverage, respectively. The increase was not huge, partly because the total number of `SeekBarPreferences` relevant to the test cases was small, and partly because the values of `SeekBarPreferences` are often used to invoke API methods, hence covering those parameter target branches does not always lead to any improvement on instruction or branch coverage. Nevertheless, such results demonstrate that Prefest can be extended to support the effective testing of more preference types.

Prefest was effective in amplifying test cases to exercise more behaviors and detect bugs that are dependent on preferences.

6.3.2 RQ2: Efficiency. Table 8 also lists for each app the time in minutes Prefest took to amplified the test cases and the breakdown of that to the time spent on various steps.

Overall, Prefest took in total 6713.8 minutes to generate 919 amplified test cases for the 45 subject apps, averaging to 7.3 minutes per amplification and 149.2 minutes per app. Among the three steps that Prefest took to amplify the test cases, test-relevant preference discovery was by far the most time-consuming. This was mainly because Prefest needs to run each input test case during the step so as to discover its relevant preferences, and running those test cases already needs 5102.9 minutes (see Table 6). Target-oriented test amplification accounted also for a significant portion of Prefest's total running time, since test cases with relevant preferences are executed repeatedly to evaluate the impact of various configurations in that step. Prefest spent relatively little time on preference locator discovery, because static analysis was able to construct valid locators for most

Table 10. Comparison between the amplification results produced by Prefest and its three variants. For each amplification approach, the *extra* instruction ($\%\mathrm{I}_{\Delta}$) and branch ($\%\mathrm{B}_{\Delta}$) coverage achieved, the time in minutes taken to produce the results (T), and the number of amplified test cases generated (#TC).

APP	Prefest				Prefest-cr				Prefest-ca				Prefest-nd			
	$\%\mathrm{I}_{\Delta}$	$\%\mathrm{B}_{\Delta}$	T	#TC	$\%\mathrm{I}_{\Delta}$	$\%\mathrm{B}_{\Delta}$	T	#TC	$\%\mathrm{I}_{\Delta}$	$\%\mathrm{B}_{\Delta}$	T	#TC	$\%\mathrm{I}_{\Delta}$	$\%\mathrm{B}_{\Delta}$	T	#TC
a2dpvolume	1.6%	2.9%	87.0	10	1.7%	3.5%	1549.7	1080	2.4%	3.7%	3214.8	1800	1.4%	2.6%	359.8	180
Alwayson	3.5%	3.8%	152.4	19	3.1%	4.7%	1494.2	1049	2.7%	4.2%	8157.8	2880	1.4%	2.5%	761.8	240
AmazeFileManager	3.1%	2.7%	202.6	29	4.2%	3.6%	3674.5	1968	2.9%	2.7%	14598.6	2520	1.5%	1.5%	1535.3	210
forecast	6.9%	16.9%	203.6	52	9.3%	21.2%	4273.0	2700	11.1%	16.7%	4583.2	3000	4.7%	10.1%	515.6	300
good-weather	8.8%	14.0%	153.6	10	10.2%	15.1%	1905.9	2035	10.5%	16.8%	4297.1	3600	7.4%	11.7%	455.1	360
KISS	7.3%	8.8%	218.4	31	9.1%	11.1%	7545.9	3590	11.0%	13.0%	18780.2	4680	7.2%	8.4%	1390.8	360
Notepad	3.6%	9.5%	164.0	61	3.2%	6.3%	1462.9	1661	3.6%	5.8%	2622.9	2889	2.3%	4.2%	292.9	321
opensudoku	2.9%	4.9%	91.3	17	2.7%	5.3%	237.5	129	3.1%	5.2%	2484.6	1200	1.8%	3.1%	233.6	120
Radiobeacon	3.7%	2.0%	143.6	21	2.6%	1.6%	475.2	294	2.9%	2.4%	3373.2	1800	2.0%	1.6%	320.2	180
SuntimesWidget	2.9%	3.2%	134.1	36	5.1%	4.7%	2998.9	1697	5.1%	5.4%	6412.5	2520	2.5%	2.9%	500.3	180
TintBrowser	8.2%	5.1%	90.6	21	8.4%	4.9%	870.2	737	8.0%	4.8%	6074.2	2520	5.9%	3.6%	500.3	210
vanilla	7.3%	6.7%	197.1	43	10.3%	10.5%	4506.4	2481	12.4%	12.4%	14549.5	3780	6.2%	5.4%	1021.6	270
Wikipedia	2.8%	2.1%	187.1	27	6.2%	5.2%	2659.5	1429	5.7%	5.2%	5733.6	2160	2.2%	1.0%	432.6	180
Overall	4.0%	4.2%	2024.7	377	5.7%	5.7%	33690.8	20850	5.8%	5.9%	94834.8	35349	3.0%	3.1%	8271.9	3111

preferences and therefore the much more expensive process of dynamic exploration was seldom needed.

The total test amplification time with Prefest was only 1.3 times of the running time of the input test cases. It is therefore clear that the two times are at the same order of magnitude. While such time costs make Prefest inappropriate to be used in an interactive way, Prefest's performance was compatible with many other usage scenarios. For example, the tool can be invoked on the test suite of an app while the developer is on a break or off for the day, and it can also be integrated into processes like continuous integration where test case selection and prioritization techniques are routinely employed to help reduce the number of tests that actually need to be executed.

Prefest was efficient in amplifying existing test cases. Its running time was at the same order of magnitude as that of the input test cases.

6.3.3 RQ3: Design Decisions. Table 10 compares test case amplification results produced by Prefest, Prefest-cr, Prefest-ca, and Prefest-nd. For space reasons, the number of bugs detected by each tool is not included in the table: Prefest-ca and Prefest-cr were both able to detect all the 7 bugs Prefest detected, but not more. Prefest-nd was able to detect three of the 7 bugs, i.e., the ones in apps *forecast*, *good-weather* and *a2dpvolume*, because each of those three bugs can be easily triggered by setting one Boolean typed preference to its non-default value.

Understandably, Prefest-ca produced the largest number of amplified test cases and achieved the greatest improvement in instruction and branch coverage with those test cases, since it utilizes a more comprehensive set of configurations to amplify the test cases; Prefest-cr produced fewer amplified test cases and achieved slightly smaller improvement in instruction and branch coverage than Prefest-ca, since it considers fewer configurations for amplification. Given that Prefest-cr managed to cover almost the same percentage of extra instructions and branches as Prefest-ca, but using only 59.0% (=20850/35349) of the amplified test cases, it is obviously more desirable to focus on relevant preferences in preference-wise test case amplification when the allocated time is limited. Similarly, Prefest managed to achieve 70.2% (=4.0%/5.7%) and 73.7% (=4.2%/5.7%) of the improvements brought by Prefest-cr in terms of extra instruction and branch coverage achieved, but using only 1.8% (=377/20850) of the amplified test cases.

Table 11. Test amplification results produced by PREFEST on programmer-written test cases. For each app, its ID, the number of amplified test cases generated (#TC), the information about the whole suite of test cases after amplification (including %I, %B, #RP, #TB, and %TBC, as reported in Table 6 for tests generated by STOAT), and the information about the amplification process with PREFEST: The number of iterations PREFEST went through in amplifying the test cases (#ITE), the amplification time in minutes (T), and the breakdown of that to the time spent on test-relevant preference discovery (T_D), preference locator discovery (T_L), and target-oriented test amplification (T_A). Values in the brackets indicate how much test amplification has improved the measurements of the test suites for the apps. Particularly, each percentage is calculated as $(m_a - m_b)/m_b$, where m_b and m_a are the measurements before and after test amplification, respectively.

ID	#TC	%I	%B	$\#P_R$	#TB	%TBC	#ITE	T	T_D	T_L	T_A
A07	21	25.1% (2.0%)	16.6% (2.5%)	15 (0.0%)	56 (5.7%)	82.1% (20.9%)	2	33.0	13.7	5.9	13.4
A12	11	22.3% (0.9%)	18.9% (3.8%)	13 (0.0%)	38 (0.0%)	92.1% (75.0%)	1	9.1	1.2	0.6	7.3
A18	17	38.7% (20.9%)	27.3% (28.2%)	22 (4.8%)	85 (32.8%)	89.4% (73.4%)	2	15.0	0.4	3.4	11.2
A22	29	30.9% (4.7%)	18.2% (9.0%)	20 (0.0%)	68 (9.7%)	79.4% (44.8%)	2	37.8	4.5	7.2	26.1
A24	0	18.8% (0.0%)	10.4% (0.0%)	0 (-)	0 (-)	0.0% (-)	1	3.1	3.1	0	0
A25	17	38.9% (25.5%)	27.4% (24.0%)	11 (0.0%)	37 (19.4%)	78.4% (51.9%)	3	13.2	1.8	2.0	9.4
A27	21	26.5% (5.6%)	20.6% (5.6%)	11 (22.2%)	39 (8.3%)	64.1% (28.2%)	2	15.9	1.4	4.8	9.7
A29	7	29.8% (20.6%)	18.7% (23.0%)	9 (28.6%)	31 (40.9%)	83.9% (41.9%)	2	10.4	1.2	2.7	6.6
A33	5	40.7% (2.3%)	17.7% (12.0%)	2 (0.0%)	10 (150.0%)	90.0% (80.0%)	2	6.2	3.5	1.1	1.7
A37	37	29.1% (4.3%)	18.2% (9.0%)	23 (4.5%)	122 (13.0%)	81.1% (23.4%)	2	49.6	5.5	3.0	41.0
Overall	165	27.1% (6.3%)	17.0% (9.7%)	126 (5.0%)	486 (16.3%)	74.1% (47.1%)	19	193.1	36.2	30.7	126.2

In terms of the total running time required by the approaches, PREFEST-CR was highly expensive, taking 23.7 days, or around 16.7 times of PREFEST's running time, to finish amplifying the test cases for the 13 apps; PREFEST-CA was even more time-consuming, taking 64.9 days to finish running on the apps; By focusing on the relevant preferences and the target branches when amplifying test cases, PREFEST only took 2024.7 minutes to run on the apps, which is merely 6.0% and 2.2% of the running time with PREFEST-CR and PREFEST-CA, respectively.

Compared with PREFEST, PREFEST-ND implements a rather straightforward strategy, and it produced a slightly smaller number of amplified test cases in the end. Those amplified test cases covered a significantly smaller part of the code, possibly because many of the amplifications PREFEST-ND produced did not help the tests exercise new behaviors. The total running time of PREFEST-ND was 3.0 times longer than that of PREFEST's. The main reason is that PREFEST-ND always amplifies test cases with full configurations, which are much more time-consuming to prepare than partial configurations often produced by PREFEST, while the target-oriented test amplification step involves repeatedly preparing those configurations and running the test cases.

Such results clearly suggest that the test case amplification strategy implemented in PREFEST is overall more cost-effective than the other alternatives.

PREFEST stroke a good balance between the effectiveness and efficiency in test case amplification.

6.3.4 RQ4: Programmer-written tests. Table 11 gives, for each app listed in Table 7, the basic information about the amplification results produced by PREFEST.

Overall, PREFEST generated 165 amplified test cases for the apps, averaging to 16.5 amplified test cases per app, and the amplified test cases helped raise the instruction and branch coverage by 6.3% and 9.7%, respectively. While the number of relevant preferences was not changed after amplification on 6 apps, that number did increase on 4 apps, achieving an overall improvement of 5.0%. More importantly, the amplified test cases raised the total number of target branches by 16.3% and the overall percentage of covered target branches by 47.1%. PREFEST achieved the greatest improvement in terms of instruction and branch coverage on three apps, namely apps *KISS* (A18), *nextcloud-news* (A25), and *openhab* (A29), mainly because these apps make heavy use of preferences, while little was done in their programmer-written tests to check the correctness of their preference-related behaviors; Prefest also managed to amplify the test cases for apps *AntennaPod* (A07) and *SuntimesWidget* (A37) and increase their coverage, because preference-related behaviors in those two apps were poorly tested, even though over 50% of the apps' test cases were preference-oriented. Prefest achieved such results in multiple iterations on all but two apps. Prefest failed to produce any amplified test cases for only one app, namely *nextcloud* (A24), because the app's programmer-written tests were not relevant to any preferences. In such a case, applying an automated tool, e.g., Stoat, to generate a collection of initial tests that make some use of preferences could help bootstrap the test amplification process with Prefest. Generally speaking, the improvements brought by Prefest on the programmer-written tests are comparable with those on the Stoat-generated test cases, suggesting that Prefest is also effective in helping programmer-written test cases exercise new behaviors that are dependent on preferences.

It took Prefest in total 193.1 minutes to produce the amplified test cases, averaging to 19.3 minutes for each app and 1.2 minutes per amplified test case. The overall test amplification time is 6.1 times longer than that of running the programmer-written test cases. The relatively larger difference between the two times is mainly because the total amount of time required to execute the programmer-written test cases is much shorter than that for running the automatically generated test cases. Since the two times are still at the same order of magnitude, we consider the amplification of programmer-written tests with Prefest to be efficient.

Prefest was effective in amplifying programmer-written test cases to exercise more behaviors that are dependent on preferences, and its running time was at the same order of magnitude as that of the input test cases.

6.4 Threats to Validity

In this section, we discuss possible threats to the validity of our study and show how we mitigate them.

6.4.1 Construct validity. Threats to construct validity are mainly concerned with whether the measurements used in the experiment reflect real-world situations.

In this work, when evaluating the effectiveness of Prefest in test case amplification, we adopted popular metrics like extra instructions/branches covered and new faults detected by the amplified test cases. However, we had to rely on a weak oracle in deciding whether amplified test cases executed successfully, even if the original test cases were equipped with programmer-written assertions. That is, we considered all executions of amplified test cases that do not cause crashes or information leaks as passing. We believe it is reasonable to exclude the old assertions from the amplified test cases since those test cases are essentially new test cases. Although we ensure in this way that all failed executions expose real bugs in the subject apps, we may miss other faults that silently produce incorrect results. In the future, on the one hand we will systematically evaluate the impact of new configurations on the existing assertions, on the other hand we will investigate the generation of new assertions for the amplified test cases.

6.4.2 Internal validity. Threats to internal validity are mainly concerned with the uncontrolled factors that may have also contributed to the experimental results.

As with other test amplification tools, the effectiveness of Prefest greatly depends on the quality of the input test cases. Therefore, one major threat to internal validity has to do with how the test cases to be amplified in our experiments were prepared. To reduce the bias in test case preparation as much as possible, we used either test cases automatically generated by the Stoat tool or test cases written by programmers. Experimental results suggest that Prefest is effective and efficient in amplifying both types of tests. Another threat to internal validity is the possible faults in the implementation of Prefest and the scripts we wrote to run the experiments. To address the threat, we carefully reviewed our code and experimental scripts to ensure their correctness before conducting the experiments.

6.4.3 External validity. Threats to external validity are mainly concerned with whether the findings in our experiment are generalizable to other situations.

A major threat to external validity is that our evaluation results may not generalize to other Android apps. To mitigate this risk, we carefully selected 45 subject apps in various sizes and with different levels of complexity from a large pool of real-world Android apps. Despite the great effort we put into the preparation of subjects, the 45 apps may not be good representatives of Android apps in general, since most of the subject apps were open-source ones, and the quality of the programmer-written test cases for the apps was not high in the sense that they cover only less than 30% of the app code. In view of that, we plan to gather more Android apps with higher-quality tests and more comprehensively evaluate the effectiveness and efficiency of Prefest on those apps in the future.

7 RELATED WORK

This section reviews researches on automated Android testing and combinatorial testing that are closely related to the work described in this paper.

7.1 Automated Android Testing

Nowadays, frameworks and tools that automate the *execution* of Android tests, e.g., Robotium [48], monkeyrunner [42], and Appium [6], are widely used in industry already. To further reduce the costs for testing, techniques like fuzz testing [2, 18, 36], model-based testing [3, 7, 20, 28, 54], search-based testing [23, 37, 38], and machine-learning-based testing [30, 33, 45, 53] have been developed to automate also the *generation* of new tests for Android apps, and symbolic analysis has been applied in many of these Android test generation techniques. Anand et al. [4] propose an approach to automatically and systematically generate events to exercise mobile apps. In their approach, events are symbolically tracked from their originating points to locations where they are ultimately handled and the gathered constraints are solved to produce new, concrete events. Mirzaei et al. [40] present *SIGDroid*, which combines model-based testing with symbolic execution to systematically generate test inputs for Android apps. Gao et al. [19] present the *SynthesiSE* symbolic execution approach for Android apps where models for Android framework are automatically deduced, rather than manually prepared. Prefest also applies symbolic analysis to gather constraints on preference values and solves those constraints to discover new configurations for testing Android apps. Symbolic analysis employed in Prefest, however, is less expensive and more likely to scale than in other Android test generation techniques for two reasons. First, its scope is more restricted and the constraints that it needs to solve are typically simpler, since Prefest focuses on just the use of preferences along the executions of given test cases. Second, given that most constraints Prefest needs to solve concern only a single preference and that the numbers of possible values those preferences may take are often small, correct solutions to those constraints can be easily determined by examining all the possible combinations. In contrast, a generic solver has to be invoked to find solutions to the constraints in techniques like *SIGDroid* and *ACTEve*, which often is expensive and risks not finding any correct solutions within the given time limit.

Another line of work in this area aims to facilitate the testing of Android apps in different settings and/or contexts. Kowalczyk et al. [27] observe that Android apps may exhibit distinct behaviors when run on different devices and operating systems with different internal settings, and they argue Android testing should be aware of the apps' internal and external configurations. Liang et al. [31, 32] focus on contextual parameters like device heterogeneity, wireless network speed, and unpredictable sensor input, and propose the *contextual fuzzing* technique that systematically explores a range of mobile contexts. Ki et al. [25] propose a framework called *Mimic* that tests UI compatibility of Android applications with different mobile devices and Android versions. Ceccato et al. [9] propose an in-vivo testing framework for mobile apps, where tests are executed in the field to check whether the apps behave correctly on different devices and/or operating systems as well as under different settings. The main rationale behind the work is that, despite the huge configuration space, the configurations that matter and should be well tested are those used in practice. Compared with these works, Prefest focuses just on the impact of preferences on app behaviors and supports effective and efficient preference-wise testing.

7.2 Combinatorial Testing

Combinatorial Testing has been an active field of researches in the last twenty years [44]. One of the major trends in this area has been towards minimizing the size of test sets w.r.t. a given combinatorial criteria. Along that line, techniques based on greedy and heuristic algorithms [11, 12, 29, 58], genetic algorithms [39, 51], or artificial intelligence [1] have been proposed. Some combinatorial optimization techniques were also adopted in Android testing recently, of which two are closely related to the work in this paper: Mirzaei et al. [41] propose *TrimDroid*, an approach that statically extracts dependencies among widgets to reduce the number of combinations in GUI testing; Sadeghi et al. [49] design the *PATDroid* technique for testing Android permission configurations that performs hybrid program analysis and reduces the amount of permission combinations to be tested by excluding irrelevant permissions. Compared with *TrimDroid*, Prefest analyzes not only the AUT but also the existing test cases when deriving new configurations. Compared with *PATDroid*, Prefest targets at preferences, which are more difficult to analyze as their values can be passed along executions. Besides, in view that features and behaviors influenced by different preferences are often independent, we implement the target mode, instead of pairwise combinatorial testing, in Prefest to strike a better balance between effectiveness and efficiency in preference-wise testing.

7.3 Test Amplification

Test amplification has been applied to augment existing tests on various platforms and for different purposes. Since Prefest amplifies GUI test cases for Android apps to cover more preference-related behaviors, we briefly review in this section test amplification techniques that aim to improve code coverage and those that target mobile applications. Interested readers may refer to [13] for a comprehensive summary of such techniques.

Tillmann and Schulte [56] describe a technique to amplify unit tests to exercise more behaviors of the program under test. The technique replaces concrete values in tests with variables, gathers path conditions over those variables via symbolic execution, and generates new values for the variables to drive the test executions to cover new paths. New test cases are often needed to cover the changed code when a program has evolved. Xu [59] proposes a directed test suite augmentation algorithm, named *DTSA*, that combines genetic algorithm and concolic testing to derive new tests from a given test suite for such a purpose. Compared with existing techniques, *DTSA* generates test cases for all paths that may reach a specific branch, which increases the chance of success. Bloem et al. [8] present a technique that combines symbolic execution and model checking to test suite augmentation. Instead of exploring new execution paths and constructing path conditions in a forward way, the technique employs a backward heuristic search on the control-flow graph to gather paths from code already covered by the input test suite to code not yet covered. All these techniques employ symbolic execution to gather conditions on values that may drive existing tests along new execution paths. As explained in the introduction, off-the-shelf symbolic execution techniques, however, are not sufficient to enable the type of test amplification that Prefest aims to accomplish, and we have to devise a new technique to construct the constraints on preference values. To make sure the preference values used in the amplified test cases do not break the integrity of app states, Prefest also generates test actions at the GUI level to configure the preferences properly.

Research has also been done to amplify test cases for mobile applications. Assis et al. [14] propose a technique called *x-PATeSCO* to generate new tests for apps on one platform based on existing tests for the same apps on another platform. Test generation in *x-PATeSCO* is based on four test patterns manually summarized from common use scenarios. Zhang et al. [60, 61] introduce a cost-effective technique that amplifies existing tests via dynamic code instrumentation to validate exception handling code in Android applications. Neither of the two techniques take into account the possible impact of preferences on app behaviors. Compared with them, Prefest amplifies existing tests with actions that explicitly set preference values at the GUI level to exercise apps under more configurations.

8 CONCLUSION

In this paper, we present the Prefest approach to effective testing of Android apps with preferences. Given an Android app and a set of test cases for the app, Prefest amplifies the test cases with a small number of configurations to enable them to exercise more behaviors and detect more bugs that are dependent on preferences. In the experimental evaluation conducted on Android apps with automatically generated and programmer-written test cases, amplified test cases produced by Prefest covered significantly more behaviors of the apps and detected real bugs.

REFERENCES

[1] Bestoun S Ahmed and Kamal Z Zamli. 2010. PSTG: a T-way strategy adopting particle swarm optimization. In *Proceedings of the 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*. IEEE Computer Society, 1–5.

[2] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M Memon. 2015. Exploiting the saturation effect in automatic random testing of Android applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, 33–43.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 59.

[5] Joseph Annuzzi, Lauren Darcey, and Shane Conder. 2014. *Introduction to Android application development: Android essentials*. Pearson Education.

[6] AppiumConf. 2019. Appium : Automation for Apps. http://appium.io/. [online, accessed 01-Mar-2021].

[7] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Acm Sigplan Notices*, Vol. 48. ACM, 641–660.

[8] Roderick Bloem, Robert Koenighofer, Franz Röck, and Michael Tautschnig. 2014. Automating test-suite augmentation. In *2014 14th International Conference on Quality Software*. IEEE, 67–72.

[9] Mariano Ceccato, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orrù, and Paolo Tonella. 2019. Toward In-Vivo testing of mobile applications. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 137–143.

[10] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.

[11] David M Cohen, Siddhartha R Dalal, Michael L Fredman, and Gardner C Patton. 1997. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.

[12] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2008. Constructing interaction test suites for Highly-Configurable systems in the presence of constraints: a greedy approach. *IEEE Transactions on Software Engineering* 34, 5 (2008), 633–650.

[13] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A snowballing literature study on test amplification. *Journal of Systems and Software* 157 (2019), 110398.

[14] Thiago Botti de Assis, André Augusto Menegassi, and Andre Takeshi Endo. 2019. Amplifying Tests for Cross-Platform Apps through Test Patterns. In *SEKE*. 55–74.

[15] Google Developers. 2021. Android Debug Bridge. https://developer.android.com/studio/command-line/adb. [online, accessed 01-Mar-2021].

[16] Google Developers. 2021. Documentation of Settings for Android Developers. https://developer.android.com/guide/topics/ui/settings. [online, accessed 01-Mar-2021].

[17] Google Developers. 2021. Google Play. https://play.google.com. [online, accessed 01-Mar-2021].

[18] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. 1–12.

[19] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 419–429.

[20] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*. 269–280.

[21] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 204–217.

[22] Sayed Hashimi, Satya Komatineni, and Dave MacLean. 2011. *Pro Android 3*. Apress.

[23] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. 2019. Search-based energy testing of Android. In *Proceedings of the 41st International Conference on Software Engineering*. 1119–1130.

[24] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 67–77.

[25] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2019. Mimic: UI compatibility testing system for Android apps. In *Proceedings of the 41st International Conference on Software Engineering*. 246–256.

[26] Satya Komatineni, Dave MacLean, and Sayed Y Hashimi. 2012. *Pro Android 4*. Vol. 1. Springer.

[27] Emily Kowalczyk, Myra B Cohen, and Atif M Memon. 2018. Configurations in Android testing: they matter. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*. 1–6.

[28] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for Android applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 115–127.

[29] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: a general strategy for T-Way software testing. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. 549–556.

[30] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 1070–1073.

[31] Chieh-Jan Mike Liang, Nic Lane, Niels Brouwers, Li Zhang, Börje Karlsson, Ranveer Chandra, and Feng Zhao. 2013. *Contextual fuzzing: automated mobile app testing under dynamic device and environment conditions*. Technical Report. Technical Report MSR-TR-2013-100.

[32] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*. 519–530.

[33] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 42–53.

[34] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2020. Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1078–1089. https://doi.org/10.1145/3324884.3416623

[35] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. 2019. Preference-Wise testing for Android applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 268–278.

[36] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[37] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.

[38] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.

[39] CC Michael, GE McGraw, MA Schatz, and CC Walton. 1997. Genetic algorithms for dynamic test data generation. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*. IEEE Computer Society, 307.

[40] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. SIG-Droid: Automated system input generation for Android applications. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 461–471.

[41] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of Android applications. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 559–570.

[42] monkeyrunner. 2019. monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/. [online, accessed 01-Mar-2021].

[43] Stas Negara, Naeem Esfahani, and Raymond PL Buse. 2019. Practical Android test recording with espresso test recorder. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. 193–202.

[44] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11.

[45] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164.

[46] pcqpcq. 2019. open-source-android-apps. https://github.com/pcqpcq/open-source-android-apps/. [online, accessed 01-Mar-2021].

[47] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2020. Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps. In *Proceedings of the 28th International Conference on Program Comprehension*. 296–307.

[48] RobotiumTech. 2019. Android UI Testing Robotium. https://github.com/RobotiumTech/robotium. [online, accessed 01-Mar-2021].

[49] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDdroid: permission-aware GUI testing of Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 220–232.

[50] Gian Luca Scoccia, Anthony Peruma, Virginia Pujols, Ben Christians, and Daniel E. Krutz. 2019. An Empirical History of Permission Requests and Mistakes in Open Source Android Apps. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, 597–601. https://doi.org/10.1109/MSR.2019.00090

[51] Toshiaki Shiba, Tatsuhiro Tsuchiya, and Tohru Kikuno. 2004. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference-Volume 01*. 72–77.

[52] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: beyond GUI testing for Android applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 27–37.

[53] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 12–22.

[54] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[55] Shin Hwei Tan and Ziqiang Li. 2020. Collaborative Bug Finding for Android Apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1335–1347. https://doi.org/10.1145/3377811.3380349

[56] Nikolai Tillmann and Wolfram Schulte. 2006. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE software* 23, 4 (2006), 38–47.

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

[58] Ziyuan Wang, Baowen Xu, and Changhai Nie. 2008. Greedy heuristic algorithms to generate variable strength combinatorial test suite. In *Proceedings of the 2008 The Eighth International Conference on Quality Software*. IEEE Computer Society, 155–160.

[59] Zhihong Xu and Gregg Rothermel. 2009. Directed test suite augmentation. In *2009 16th Asia-Pacific Software Engineering Conference*. IEEE, 406–413.

[60] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 595–605.

[61] Pingyu Zhang and Sebastian Elbaum. 2014. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–28.