Preference-Wise Testing for Android Applications

Yifei Lu
Minxue Pan*
Juan Zhai
lyf@smail.nju.edu.cn
mxp@nju.edu.cn
zhaijuan@nju.edu.cn
State Key Laboratory for Novel Software Technology,
Software Institute, Nanjing University
Nanjing, China

Tian Zhang*
Xuandong Li*
ztluck@nju.edu.cn
lxd@nju.edu.cn
State Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology,
Nanjing University
Nanjing, China

ABSTRACT

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, need to be thoroughly tested. Unfortunately, the specific preferences used in test cases are typically not explicitly specified, forcing testers to manually set options or blindly try different option combinations. To effectively test the impacts of different preference options, this paper presents Prefest, as a preference-wise enhanced automatic testing approach, for Android apps. Given a set of test cases, Prefest can locate the preferences that may affect the test cases with a static and dynamic combined analysis on the app under test, and execute these test cases only under necessary option combinations. The evaluation shows that Prefest can improve 6.8% code coverage and 12.3% branch coverage and find five more real bugs compared to testing with the original test cases. The test cost is reduced by 99% for both the number of test cases and the testing time, compared to testing under pairwise combination of options.

CCS CONCEPTS

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

KEYWORDS

Android apps, Android testing, preference-wise testing

ACM Reference Format:
Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. 2019. Preference-Wise Testing for Android Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Corresponding author.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The last decade has witnessed a rapid growth in Android apps, drawing attention from both academia and industry. To cope with the ever-changing market demands, Android app developers have to work in fast development cycles, causing a growing need for cost-effective testing approaches. Automatic generation of test inputs [3, 7, 8, 15, 19] aiming at the fully automatic testing of Android apps, as an example, has been prosperous since then.

For mobile apps on all platforms, it is often the case that there are some setting options designed to allow users to change app features and behaviors, and in Android, it is the *preference* [12]. By using preference, users can switch among different GUI styles, change the behaviors of certain functions, and enable or disable services, etc. While preference offers users the ability of customization, unfortunately for developers, the resulting diverse GUI displays and app behaviors require more testing under different preference options. Indeed, an app may work well in one setting of preference options, while crash in another. To properly test an app's behavior under different preference options, which we call *the preference-wise testing*, can be challenging. The specific preferences used in one test case is typically not explicitly specified, and existing tools have not considered the impacts of preferences on app behaviors during testing. Black-box testing captures app status from GUIs. Since changing preference options usually causes just slight or even no changes in GUIs, preferences are mostly ignored. As for while-box testing, since a key-value mechanism is used for preference access, where the keys are typically dynamically generated, techniques such as symbolic execution are required for the accurate prediction of keys. However, symbolic execution is predominantly known to be suffering from scalability issues [16], which is even worse for Android apps due to the event-driven nature and the application development framework (ADF) [23]. Therefore, despite of the recent progresses in mobile testing, testers are still forced to manually set preference options or try different option combinations for the same test case, if they want to perform preference-wise testing.

In this paper, we propose the problem of preference-wise testing for Android apps and present the Prefest approach. Prefest is built on two key observations. Our first observation is that a test case typically interacts with just a few preferences defined in the app. So, for each test case, Prefest analyzes the preferences that may impact the app behavior, which we call *test case relevant preferences*, and executes test cases only under relevant preference option combinations. Specifically, given an Android app, Prefest first leverages

a static analysis to identify the preference structure that contains all the preferences defined in the app. Then in a dynamic analysis, it executes the test cases and logs the execution flows to pinpoint the relevant preferences to each test case. Finally, it re-executes test cases only under relevant preference option combinations to reach previously uncovered code.

To further reduce the test cost, we exploit our second observation that Android apps often share app states globally using the key-value mechanism. So, under one preference option combination, a piece of code executed in different test cases often produces the same app behavior, and therefore, does not require re-executions. We equip Prefest with a reduction strategy named *Target Mode*, which splits the app code into blocks and performs a further analysis of the relevance between the preferences and the code blocks. For one code block, referred as *target*, Prefest will execute it only if it has not been executed by previous test cases.

Prefest can enhance the performance of existing automated testing tools. In addition, it can be a useful complement to manual testing. In practice, developers and testers are often not the same group of people. Identifying relevant preferences for test cases and testing apps under adequate preference options can be a costly or even tough job for testers. This is where Prefest can be handy, since it is all automated and the manual effort can be saved.

The main contributions can be summarized as:

(1) A novel problem of *preference-wise testing* and also a fully automated solution Prefest, to improve the efficacy of existing testing approaches by considering the effects of preferences;
(2) Multiple techniques employed in analyzing the impacts of preferences to Android testing, including the loading patterns for preference identification, the analysis for relevant preferences acquisition, and the Target Mode for test cost reduction;
(3) A prototype also named Prefest and an empirical study on 30 real-world apps, showing that Prefest achieves 6.8% and 12.3% improvement in code and branch coverages, respectively, and detects five more real bugs.

The paper is organized as follows. Sec. 2 introduces the background and motivation of our work. Sec. 3 provides the overview and the details of the Prefest approach. Sec. 4 presents the experimental evaluation. Related work is discussed in Sec. 5 and conclusion is drawn in Sec. 6.

2 BACKGROUND & MOTIVATION

2.1 Background

In Android, GUI pages containing preferences are called *setting screens*. To use setting screens in an app, a programmer needs to define: (1) resource files (in XML format) to describe the preferences in each setting screen; (2) invocations of preference-related APIs in source code to specify the loading location of each setting screen; and (3) the accesses of preferences in source code.

Listing 1 shows a simplified resource file for a setting screen. The top-level tag *PreferenceScreen* defines the container for a setting screen. Each contained element represents a preference of different types, such as *ListPreference* and *CheckBoxPreference* in Listing 1.

To perform preference-wise testing, we need to obtain the essential details for each preference, including: (1) *key*: the unique name to refer to the preference in source code; (2) *title*: the text displayed in the setting screen; (3) *defaultValue*: the initial value of the preference; and (4) *entryValues*: the possible options can be set to the preference. As Listing 1 shows, these details are coded in the resource files, which can be retrieved by static analysis.

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

Listing 1: Sample resource file for preferences

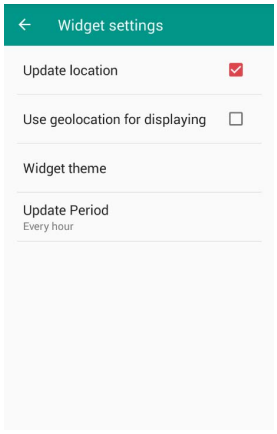
For an app to load a setting screen defined in the resource file, the most common way is to call the API method *addPreferencesFromResource* with the resource file as its parameter, upon the creation of an *Activity* or a *Fragment*, i.e., within their lifecycle methods *onCreate*. A special setting screen named *PreferenceHeaderScreen*, which shows a list of navigation texts to switch among different setting screens, is officially recommended to load with another API method *loadHeadersFromResource* (see Sec. 3.2).

The accesses of preferences values are particularly complex. Android provides the *SharedPreferences* mechanism for activities and applications to manage preference data in the form of key-value pairs of primitive data types in the Android file system. The precise values of keys are critical to analyzing which preferences are relevant to a test case. However, they are difficult to acquire through static analysis since very often they are generated dynamically. To address this problem, we employ a dynamic approach to analyze which preferences are loaded and used for the given test cases. More details will be discussed in Sec. 3.3.

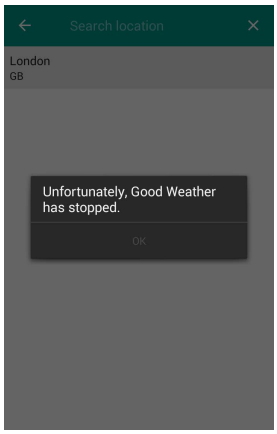
2.2 Motivation

In this section, we use a simple app, called *GoodWeather*, to show how preferences affect app behaviors. GoodWeather is an app that allows users to select the location by GPS or text search and displays the weather condition for the selected location. It also has a feature called widget that decks out the phone screen with the up-to-date weather condition. Users are offered with customization options manifested in preferences, as shown in Figure 1a.

Some of the preferences can change the widget's functions, for example, *update location* can determine whether or not to start a service to runtime synchronize the location in the widget with the one set in the app. Others can be used to customize the styles of look, such as *widget theme*. The setting of such preferences can affect either the app behavior or the GUI display, and in some cases, cause bugs. For example, by default, *update location* is set to *disabled*, under which users are able to change the location. However, if *update location* is enabled, when users try to change the location, a crash would occur, as shown in Figure 1b. Clearly, to reveal this



(a) The setting screen for widget settings



(b) A crash when selecting a location

Figure 1: A preference-related crash in GoodWeather

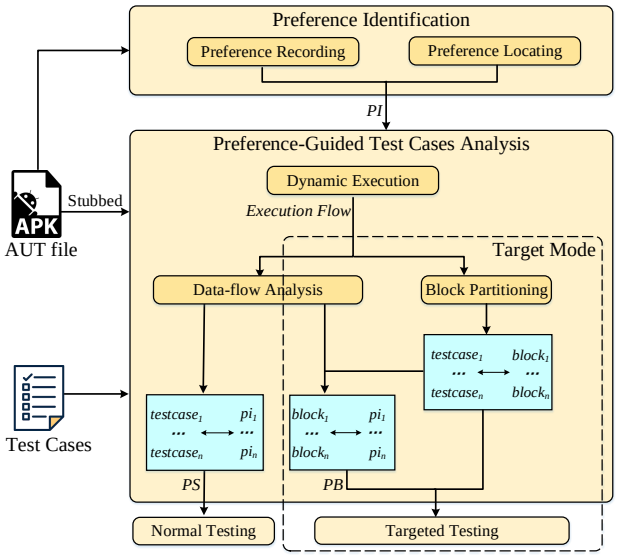


Figure 2: Overview of Prefest

bug, testers need to set this specific preference option first, and then change the location in the app. However, there is no explicit connection between a preference setting for the widget and a failure in the main app, and thus this bug is very likely to be untested.

From the GoodWeather example, it is obvious that a systematic and thorough preference-wise testing is needed to improve app quality. However, preference-wise testing can be challenging, since the impacts of preferences are tangled with app functions. As illustrated by the example, only enabling preference *update location* or selecting the current location will not trigger the crash. Very often testing tools or even human testers have no knowledge about what preferences would affect the functions under test. Therefore, to intentionally reveal instead of randomly triggering the preference related bugs, they may have to perform exhaustive combinations of test cases and preference settings, which can lead to an explosion in testing space. Hence, there is an urgent need for the study of cost-effective preference-wise testing approach.

3 PREFERENCE-WISE TESTING

3.1 Approach Overview

Figure 2 depicts the overview of Prefest. Given an APK file of the *App Under Test (AUT)* and a set of test cases for the *AUT*, Prefest identifies the relevant preferences that may affect the app behavior and runs the test cases under relevant preference option combinations to have a more thorough test of the *AUT*. The test cases can be written manually, or generated from automated testing approaches like *AndroidRipper* [3], A^3E [7] or *Stoat* [33]. Prefest consists of two major analyses: *Preference Identification*, which identifies and locates all the preferences (denoted as *PI*) defined in *AUT*; and *Preference-Guided Test Case Analysis*, which reveals the relevance between preferences and test cases through a data-flow analysis, and only tries the combinations of relevant preference options for each test case (denoted as *PS*). An additional analysis mode, called *Target Mode* is also proposed, in which Prefest splits the code into code blocks, and identifies the untested blocks and their relevant preferences (denoted as *PB*). It only executes the test cases that can reach untested code blocks, and therefore, is more efficient.

3.2 Preference Identification

To conduct preference-wise testing, it is necessary to first identify the collection of preferences defined in the *AUT*. Prefest achieves so by reversing preference resource files from the *AUT* with *jadx* [31], and recording preferences by their *key*, *title*, *type* and *entryValues*. Currently, it supports four types of preferences, which are *SwitchPreference*, *CheckBoxPreference* , *ListPreference* and *EditTextPreference*. The decision is based on the investigation of 115 apps containing preferences from a popular open-source Android app list on GitHub [27]. It shows, on average, each app contains 20 preferences, of which 18 (90%) preferences are of the aforementioned four types. The other 10% are of the other types or customized preferences by developers, which we plan to support in the future.

Then, Prefest uses *Soot* [17] to statically analyze the source code for the Activities and Fragments in which the preferences are located. It first collects all the direct method calls, denoted as $m_{caller} \rightarrow m_{callee}$, in *AUT*. The method callbacks are not considered here, since methods responsible for loading setting screens are mostly directly called during the initialization of the Activities or Fragments. Each method m is assigned an attribute *declass* representing its declaration class. A *call trace* ρ, defined as $\rho = m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_{t-1} \rightarrow m_t$, represents that through methods $m_2, m_3, \ldots, m_{t-1}$, m_1 eventually invokes m_t, and T is the set of all call traces. Prefest conducts analysis on the call traces to identify where the setting screens are loaded. By studying ways of implementing the setting screen loading, we summarized three loading patterns from the Android official documents, as shown in Table 1.

Pattern LPA represents that the loading of a setting screen is performed by an Activity, where the loading API *addPreferencesFromResource* is eventually called by the *onCreate* method of an

Table 1: Patterns for loading setting screens

Pattern	Definition
LPA	$\exists \rho = m_{oc} \rightarrow ... \rightarrow m_{add} \in P$, $m_{oc}.declass \in Activities$
LPF	$\exists \rho, \rho' \in P, \rho = m_{oc} \rightarrow ... \rightarrow m_{add}$, $\rho' = m'_{lifecycle} \rightarrow ... \rightarrow m'_{init}$, $m_{oc}.declass = m'_{init}.declass \in Fragments \wedge$ $m'_{lifecycle}.declass \in Activities$
LPH	$\exists \rho, \rho' \in P, \rho = m_{oc} \rightarrow ... \rightarrow m_{add}$, $\rho' = m'_{oc} \rightarrow ... \rightarrow m'_{load}$, $m_{oc}.declass \in fragments\_referred(m'_{load}) \wedge$ $m'_{oc}.declass \in Activities$

m_{oc}: lifecycle method *onCreate*;
$m_{lifecycle}$: any lifecycle method;
m_{add}: API method *addPreferencesFromResource*;
m_{load}: API method *loadHeadersFromResource*;
m_{init}: the constructor of a Class.

Activity through ρ. The setting screen is shown when the activity is launched. Pattern LPF represents that the loading of a setting screen is performed by a Fragment, which itself is initialized by an Activity. Loading API *addPreferencesFromResource* is eventually called by method *onCreate* declared in a Fragment, and an Activity instantiates this Fragment in one of its lifecycle methods through ρ'. For pattern LPF, the setting screen is shown when the activity is launched, initializing the fragment to load the setting screen. Pattern LPH represents that a preference header, responsible for loading multiple setting screens, is loaded by an Activity. Through call trace ρ, loading API *addPreferencesFromResource* is eventually called by method *onCreate* declared in a Fragment. Different from pattern LPF, the Fragment is not initialized explicitly, but instead, referred to in a preference header resource file. When an Activity eventually calls method *loadHeadersFromResource* in its *onCreate* method through ρ' and loads the preference header, all fragments referred to in its resource file, represented by $fragments\_referred(m'_{load})$, are initialized by the Android system. For pattern LPH, when the activity is launched, a preference header is shown, containing a list of selections for users to switch among different setting screens.

To analyze which pattern is adopted, Prefest starts from each m_{add} and m_{load}, and performs a backwards search for any match of the pattern LPA, LPF or LPH. After the analysis, it obtains necessary information for each preference, denoted as $pi = \langle key, title, type, entryValues, location \rangle$. We define *PI* as the set of all the *pi*. With *PI*, Prefest is able to set any concerned preference option combinations automatically with off-the-shelf Android GUI test frameworks.

3.3 Preference-Guided Test Case Analysis

To reduce the number of preference option combinations for test cases, we need to analyze for each test case which preferences are relevant. We define the relevant preferences to a test case are those whose values are acquired, passed and used in branch conditions during the execution of the test case, since preferences used in branch conditions can dynamically modify the function behaviors. These branches, ignored by existing approaches, are usually blind spots in Android testing.

However, it is difficult to conduct a precise analysis statically, as Android apps are not stand-alone applications but plugins into the Android framework [6]. Even worse, the SharedPreferences mechanism used in preferences' acquisition makes that the same line of code may point to a different preference, since the key of the preference can be changed. Techniques such as symbolic execution is required, however, they suffer from scalability issues due to the event-driven nature and the application development framework of Android.

We propose a dynamic analysis to address this problem. For the AUT, Prefest instruments loggers with Soot at the beginning and the end of each method, and also at each branching point. For efficiency, loggers are not instrumented in Android SDK and the other external libraries. We simply record the invocations of API methods in these libraries. When running a test case, the logs are automatically collected, from which an execution flow comprised of a linear sequence of statements is generated. Then Prefest analyzes the execution flow statement by statement and collects variable manipulations and branch conditions.

$$
\begin{aligned}
(expression)\ e &::= n \mid pi \mid v \mid op(e) \mid mi(e) \in E \\
(variable)\ v &::= \{v_1, v_2, ..., v_m\} \in V \\
(condition\ label)\ l &\in Label \\
(statements)\ s &::= v = e \mid if\ e^l\ s_i\ else\ s_j \mid \\
&\qquad switch\ e^l\ case\ n_i : s_i;\ case\ n_j : s_j; \ldots \\
(execution\ flow)\ f &::= s_1; s_2; s_3; \ldots; s_n
\end{aligned}
$$

The syntax of an execution flow is shown above. Here, V and E represent the sets of variables and expressions, respectively. Each $e \in E$ can be a constant n (including constants of Boolean, Integer, Float, String), a variable v, or an expression constructed with a java operator *op*, a method invocation *mi* or a symbolic variable *pi* representing a preference. Recall that all necessary information for manipulating a preference is in *pi* (Sec. 3.2), so it is natural to use *pi* as the symbolic representation for preferences.

In an execution flow, loops are unfolded during the dynamic execution. Branch conditions of conditional statements are all labelled. Additionally, for invocations of methods that are instrumented, the parameter passing and method returns are also considered as assignments, and the execution of method bodies are included in the execution flow. Finally, an execution flow f is represented as a series of statements.

$$
\begin{aligned}
(symbolic\ variable\ state)\ \Gamma_v &::= [v_1 : e_1, \ldots, v_m : e_m] \\
(symbolic\ conditional\ state)\ \Gamma_c &::= [l_1 : e^{l_1}, \ldots, l_n : e^{l_n}] \\
(execution\ state)\ \omega_s &::= \langle \Gamma_v, \Gamma_c \rangle
\end{aligned}
$$

Our data-flow analysis is performed along the execution flow, statement by statement. To deal with aliasing, an Andersen's style analysis is implemented. The execution state ω_s at statement s is defined above. For ω_s, we use Γ_v and Γ_c to define the mapping relation that maps a variable v or a branch condition labeled with l to its symbolic expression e.

By applying Γ_v on the variables representing keys, expressions about keys can be obtained. In most cases, keys are represented with constants, or string operations over several constants, and therefore, Prefest can calculate the concrete values of such keys. Then it retrieves the preferences having been accessed during testing from *SharedPreferences* by interpreting the seven preference acquisition methods defined in the Android official documents, including getBoolean, getFloat, getString, getInt, getLong, getStringSet, and getAll, with the calculated concrete values of keys.

```
...
S1: $r1 = "widget_"
...
S2: $r2 = $r1 + "update_location_pref_key"
...
S3: $r3 = SharedPreferences.getDefaultSharedPreferences()
S4: $z0 = $r3.getBoolean($r2,0)
...
S5: $z1 = !$z0
S6: if($z1 == 0)
          l6
...
```

Listing 2: A slice of execution flow of GoodWeather

Take the preference *update location* of GoodWeather in Sec. 2.2 for example. Listing 2 shows a slice of the execution flow, consisting of four inconsecutive sequences of statements, which acquires and uses preference *update location*. Statements $S1$ and $S2$ generate the key of preference *update location* by string concatenation. $S3$ obtains *SharedPreferences* which stores all preferences. $S4$ invokes a preference acquisition method (*getBoolean*) with variable $\$r2$ as the key, and assigns the acquired preference option value to $\$z0$. $S5$ assigns the reverse of $\$z0$ to $\$z1$, which contributes to the branch condition in $S6$. Prefest calculates the concrete value of the key variable $\$r2$ used in $S4$, which is "widget_update_location_pref_key". It then interprets *getBoolean* in $S4$ with the value of $\$r2$, to get the specific preference *update location*, represented by the symbolic variable $pi_{update\_location}$.

With Γ_c, a relevant preference can be revealed from whether its pi is directly or can affect by assignments other variables contained in the symbolic value of any branch condition. For instance, in the Γ_c of the execution state at $S6$ of Listing 2, we have $\langle l_6, pi_{update\_location} \neq 0\rangle$ for the branch condition in $S6$. So, preference *update location* is *relevant* to this branch condition, and by setting it to different values (*true* or *false*), the execution can reach different branches.

Now Prefest can test different app behaviors by trying different option value combinations of the relevant preferences, instead of all the preferences in the app. The valid values for a preference, i.e., entryValues, are already known, as discussed in Sec. 3.2. Specifically, SwitchPreferences and CheckBoxPreferences can be set to *true* or *false*; ListPreferences can be set to a finite set of options in the form of strings, predefined by developers; for EditTextPreferences which accept user text inputs as their values, Prefest uses boundary values as its entryValues, such as null or a random string (0, 1, IntMax are tested when only number input is allowed), since it focuses on bug detection.

We define a preference option combination to be tried for a test case *testcase* as a *ps*, where $ps = \langle\{\langle pi, value\rangle\}, testcase\rangle$, and each $\langle pi, value\rangle$ represents the setting for a single preference. A *testcase* can have multiple *ps* representing different option combinations. Note that the number of *ps* for a test case depends on the combinatorial strategy of preferences. For example, a pairwise combinatorial strategy can result in a smaller number of *ps* than a full combinatorial strategy. *PS* represents all preference option combinations to be tested on all test cases in our preference-wise testing. Given a $ps \in PS$, Prefest generates a script and executes it to set preference option values, before executing the test case $ps.testcase$. In the script, for each $\langle pi, value\rangle$ of *ps*, the *pi.title* and *pi.location* help locate the preference in the screen, while *pi.type* and *value* are used to generate operations that set the correct option value for the preference. After all the relevant preference are set, the original test case *ps.testcase* is executed.

3.4 Test Cost Reduction with Target Mode

By focusing on relevant preferences, Prefest only needs to try option combinations for the relevant preferences. However, we empirically found out that *PS* can still be of a large size in some cases. For instance, in app *Suntimes*, 12 two-option (*true* and *false*) preferences are used in branch conditions upon its initialization, where option combinations can be too many. A further reduction of test cost is required, and we propose the *Target Mode*.

In Target Mode, Prefest splits the app code into blocks—straight-line code sequences with no branches in except to the entry and no branches out except at the exit. Since Prefest aims at testing the preference-related branches, we select blocks in preference-related branches as our targets. Noticing that third-party libraries can also be affected by preferences through parameter passing to demonstrate different behaviors, blocks containing invocations of third-party methods with preference-related variables as their parameters are also considered as targets. By splitting the execution flows into blocks, Prefest analyzes the relevant preferences to targets, similar to the analysis in Sec. 3.3. Like the *ps* for a test case, we define a preference option combination to be tested for a target as $pb = \langle\{\langle pi, value\rangle\}, block\rangle$.

As discussed earlier, targets only need to be executed once during testing. To accelerate the testing process, Prefest adopts the greedy strategy that the test case which can potentially execute most targets under a certain preference option combination is selected to be executed first. The key to the strategy is that we need to know which blocks can be reached by a test case under different option combinations, and which option combinations can help to reach the previously unreached blocks. By analyzing the execution flow of a test case combined with code, we can locate the branching points that the test case can reach, and all branches belong to these branching points can be reached potentially. To test the unreached blocks, we need the concrete values of variables, including preferences, to manipulate the values of the branch condition. Thanks to the symbolic representation of the branch conditions, most concrete values can be calculated. Thus, given a target block, Prefest can produce its *pb*, which is used to set the values of preferences.

Algorithm 1 shows the details of the Target Mode. It takes *PS*—the set of test cases with different preference option combinations—as input, and outputs PB_{total}—the set of the reached blocks with the option combination settings when it finishes. In the beginning is a

Algorithm 1: Target Mode of Prefest.

Input: $PS = \{\langle\{\langle pi, value\rangle\}, testcase\rangle\}$
Output: PB_{total}: the total set of pb of reached blocks

```
1  foreach ps ∈ PS do
2      PBS := getTargetBlocks(ps);
3      PB_target.put(PBS);
4  end
5  PB_target.remove(PB_origin);
6  PB_total := ∅;
7  while PS ≠ ∅ & PB_target ≠ ∅ do
8      ps_max := getMostBlocks(PS);
9      PB_reach := execute(ps_max);
10     PB_total := PB_total ∪ PB_reach;
11     PB_target.remove(PB_reach);
12     PS.remove(ps);
13 end
```

loop (lines 1-4) that iterates each $ps \in PS$ to get the blocks that can be potentially reached by $ps.testcase$, which form the set of targets PB_{target}. Then in line 5, PB_{origin}, the already reached blocks when executing the original test cases, are removed from PB_{target}.

Next, a greedy algorithm, which is also a loop, starts from line 7. It aims at reaching as many unreached blocks as possible each turn until all unreached blocks are reached or there is no test cases left to be executed. In the loop, Prefest will (1) search PS for ps_{max} that can test most unreached blocks; (2) execute ps_{max} and record all the reached blocks with option combination settings (PB_{reach}), add them to PB_{total} and remove them from PB_{target} (lines 9-11; (3) remove ps from PS (line 12).

The Target Mode uses just one option combination for a block, instead of exhausting combinations of relevant preferences. This is particular effective when there are more than one preferences in a branch condition. As the experiment in Sec. 4.4 shows, Target Mode can reduce a significant portion of the test cost, while still being effective in code coverage and bug detection.

3.5 System Preference Analysis

In this paper, we mainly focus on user preferences, which are designed for users to change app behaviors and features. Similarly, environment configurations of the Android system can also make apps behave differently. They are like the preferences on the system level. Prefest also supports the testing under different environment configurations. Currently, Prefest supports six kinds of environment configurations that are often used, which are *WiFi*, *bluetooth*, *mobile data*, *GPS locating*, *network locating* and *music playing*. By interpreting API methods for acquiring the status of the six environment configurations, similar to the interpreting the preference acquisition methods but no keys required, the environment configurations can be treated the same as user preferences.

4 EVALUATION

We implemented our approach into a tool, also name Prefest. The tool and the experimental data are available online [1].

[1] https://github.com/Prefest2018/Prefest

To evaluate Prefest, we conducted a series of experiments to answer the following questions:

RQ1 How effective is Prefest in terms of the code/ branch coverages and the bug detection ability?
RQ2 How efficient is Prefest in terms of the number of test-runs and the test time?
RQ3 How does Prefest compare against alternative approaches for preference option combinations in terms of effectiveness and efficiency?
RQ4 How does *Target Mode* perform? Specifically, does it strike a good balance between test cost and test effectiveness?

4.1 Experiment Setup

We selected *Stoat* [33], one of the state-of-the-art automated Android testing tools, to generate test cases as inputs for Prefest. The subject apps are chosen from both previous researches [29, 32] and a popular open-source Android app list on GitHub [27] with the following criteria:

(1) the app should contain at least five preferences in its setting;
(2) the app should be able to run standalone instead of as a library, and should be compatible with Android API-19, which is the recommended environment for *Stoat*;
(3) the app should achieve a code coverage of over 20% and not easily crash when tested with *Stoat*.

Eventually, 7 apps from previous researches and 22 apps from the GitHub list satisfying the criteria were selected. Together with our motivating example GoodWeather, totally 30 apps were chosen as our subjects. We also analyzed the apps' sizes by lines of ByteCode (i.e., lines of instructions, calculated by JaCoCo) and numbers of preferences. The results show that the complexity of these apps has enough diversity for ranging from 5k instructions with 5 preferences, to over 200k instructions with 96 preferences.

To answer the RQs, we first applied Prefest with Target Mode (denoted as Prefest(T)) on all 30 apps. Then, we compared Prefest(T) with another two combination approaches for preference options, which are:

NonDefault—from Sec. 3.2, we know that each preference has a default value under which the original test case is executed. In this strategy, each preference is set to a value other than its default value (random value is used if there are multiple valid values).

Pairwise—the most common type of t-way combinatorial testing [26], that is, for any two preferences among all preferences, all possible pairs of their option values are tested for each test case. For ListPreference and EditTextPreference which may have multiple values, only two values—the default one and a randomly selected one—is used to restrict the combination number in *Pairwise*.

We also compared Prefest against an implementation without the Target Mode (denoted as Prefest(N)) in the comparative study. Prefest(N) uses pairwise technique to construct the set PS to be executed. In other words, compared with *Pairwise*, Prefest(N) adapts the pairwise testing for not all preferences but only relevant ones.

The comparative study was only conducted on GoodWeather and the seven apps from previous researches, since it was extremely time consuming and virtually impossible to conduct the study on all 30 apps.

Table 2: The performance of PREFEST on 30 apps

Subject	Inst.	Pref.	*Default*		PREFEST(T)		*Default*		PREFEST(T)	
			Inst.%	Branch%	Inst.%	Branch%	Time(min)	#Run	Time(min)	#Run
GoodWeather	11192	11	60.61	35.10	68.33(+12.7%)	47.05(+34.1%)	137	360	14(<1)	14
A2dpvolume	18324	15	40.03	17.39	41.56(+3.8%)	20.31(+16.8%)	73	180	8(<1)	6
Alwayson	15379	29	44.53	30.71	46.10(+3.5%)	33.63(+9.5%)	121	240	22(<1)	15
Suntimes	64898	25	39.65	29.25	42.69(+7.7%)	32.58(+11.4%)	88	150	32(<1)	22
Opensudoku	17606	14	44.61	32.53	46.76(+4.8%)	36.60(+12.5%)	65	120	12(<1)	10
Radiobeacon	36252	20	37.18	19.90	39.60(+6.5%)	21.28(+6.9%)	118	180	17(<1)	16
NotePad	7722	6	51.97	39.84	55.19(+6.2%)	47.81(+20.0%)	104	321	48(<1)	56
WikiPedia	105509	53	43.14	27.31	45.66(+5.8%)	29.23(+7.0%)	123	180	28(4)	17
fillup	18141	6	21.13	16.46	24.32(+15.1%)	20.06(+21.9%)	123	210	17(<1)	19
TintBrowser	33074	22	26.98	16.15	29.06(+7.7%)	17.49(+8.3%)	65	210	3(<1)	3
Signal	237215	40	30.58	14.27	31.81(+4.0%)	15.04(+5.4%)	246	450	49(11)	25
Anki-Android	128890	96	33.05	22.68	34.83(+5.4%)	24.75(+9.1%)	145	210	17(<1)	16
Runnerup	75863	67	20.15	13.66	21.14(+4.9%)	14.87(+8.9%)	93	150	14(1)	12
amme	93689	34	25.29	17.91	26.58(+5.1%)	18.86(+5.3%)	105	180	10(2)	8
nanoConverter	8508	6	34.38	30.89	38.32(+11.5%)	37.76(+22.2%)	107	210	23(<1)	19
APhotoManager	47512	25	38.96	26.77	42.82(+9.9%)	29.52(+10.3%)	71	150	33(<1)	23
Timber	55262	11	25.42	15.11	25.98(+2.2%)	16.25(+7.5%)	90	240	9(1)	8
AntennaPod	38421	34	25.81	18.54	28.80(+11.6%)	22.03(+18.8%)	77	180	26(1)	24
vanilla	48501	36	45.54	35.20	48.99(+7.6%)	38.74(+10.1%)	110	270	60(1)	35
materialistic	30085	37	44.33	26.09	49.89(+12.5%)	32.12(+23.1%)	113	210	61(2)	32
RedReader	59012	62	32.83	24.94	34.83(+6.1%)	28.35(+13.7%)	67	240	24(2)	21
commons	42439	5	25.80	15.02	25.98(+0.7%)	15.22(+1.3%)	101	270	5(1)	4
hacker-news	14790	5	47.84	34.28	50.55(+5.7%)	37.13(+8.3%)	80	240	18(1)	14
KISS	20003	37	47.44	34.28	53.69(+13.2%)	41.25(+20.3%)	151	360	44(4)	24
uhabit	20107	11	54.76	28.47	55.66(+1.6%)	29.14(+2.4%)	101	300	4(1)	3
Omni-Notes	34993	23	25.09	20.38	26.30(+4.8%)	21.62(+6.1%)	124	330	12(1)	11
AmazeFileManager	80767	31	20.64	14.97	22.26(+7.9%)	16.70(+11.6%)	138	210	55(4)	35
connectbot	70864	26	23.21	23.88	23.37(+0.7%)	24.41(+2.2%)	83	120	11(1)	3
forecast	8331	15	57.94	40.79	65.23(+12.6%)	53.57(+31.3%)	137	300	26(1)	22
OpenBikeSharing	5529	5	58.84	47.93	60.70(+3.2%)	48.76(+1.7%)	92	180	18(<1)	17
Average	**48284**	**27**	**37.59**	**25.69**	**40.23(+6.8%)**	**29.07(+12.3%)**	**107**	**232**	**24(2)**	**18**

The experimental environment was a physical machine with 8GB RAM and 2.0GHz quad-core processor. The Android emulator to run tests was configured with 2GB RAM and the X86 ABI image (SDK 4.4.2, API level 19). The running of *Stoat* was on Ubuntu 14.04 configured as: 1h for GUI exploring, 1h for MCMC sampling, 30 steps as the longest steps in sampling one case, and 30 cases generated at one iteration of sampling. For comparison, we retrieved the test cases from Stoat's records of MCMC sampling and run the tests on Windows 10 under the above four strategies. For all the experiments, we use *JaCoCo* [14] to calculate the coverage of instructions and branches.

4.2 RQ1: Effectiveness on Coverage and Bugs

Table 2 lists the 30 apps, their sizes measured by number of instructions and preferences, the instruction and branch coverages achieved by the original test (*Default*) and PREFEST(T), respectively. As we can see, with a preference-wise testing, the coverages of all subjects have been improved by percentages ranging from 0.7%-15.1% for instruction coverage, and 1.3%-34.1% for branch coverage. The average improvement is 6.8% and 12.3%, for instruction and branch coverages, respectively. As an enhanced testing for an already state-of-the-art tool, this improvement is significant.

We can see that PREFEST(T) achieved large improvement in some apps: among 30 apps, instruction coverage improvement over 10% is seen in 8 apps, and branch coverage improvement over 20% is seen in 7 apps; and small improvement (less than 3% improvement in instruction and branch coverages) in 3 apps—*commons*, *connectbot* and *uhabit*. We studied these apps and their original test cases, and found out that the apps having more improvement were better tested by *Stoat*, compared with the apps with less improvement. This is reasonable, since PREFEST is a complement to existing testing approaches and relies on the execution flows to analyze the relevant preferences. Therefore, the preference-wise testing and the other testing approaches can form a mutual boost relationship in performance.

It is worth mentioning that for apps *Signal*, *Anikandroid* and *Wikipedia*, although the improvement of 4.02%, 5.39% and 5.84% (5.40%, 9.13% and 7.03%) in instruction (branch) coverage is not

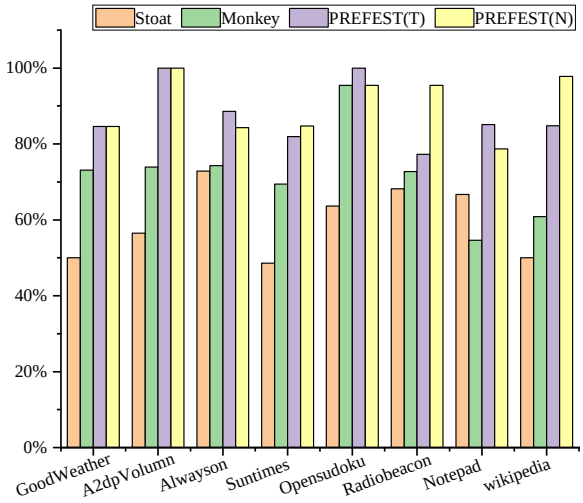


Figure 3: Preference-related branch coverage achieved by *Stoat*, *Monkey*, Prefest(T) and Prefest(N)

significant, considering that these apps have more than 100k instructions, the more tested instructions and branches, in absolute terms, can be over 2000 instructions and 100 branches.

We are particularly interested in branch coverage, since branches can cause different app behaviors with the same movements on the GUIs, and are common in complex apps. We conducted experiments to evaluate how well the branches can be tested with Prefest, and whether it is possible to use existing approaches to obtain similar or better results. We chose *Default* (*Stoat*), Prefest(N), Prefest(T), and an additional testing tool *Monkey* [13]—a clear winner among current test input generation tools [9], to conduct experiments on the example of *goodWeather* and the seven apps from existing researches. We configured *Monkey* as [9] suggested, and the test time was also set to 1 hour, the time for MCMC sampling in Stoat.

The results are shown in Figure 3. For all the branches in the apps that can be affected by preferences, Prefest(T) and Prefest(N) covered 88% and 90% branches on average. Although Prefest(T) tries less preference option combinations than Prefest(N), in some apps, it can achieve higher branch coverage, since Prefest(T) can select the exact options for ListPreferences to cover preference-related branches via the concrete value calculation, whereas for the pairwise combination strategy of Prefest(N), a random selection of the ListPreferences value is used for the non-default value. *Stoat* and *Monkey* achieved 59% and 72% branch coverages on average, respectively. Considering that all preferences have default values, even forbidding the setting of preferences, *Stoat* and *Monkey* should be able to achieve a branch coverage ranging from 30% to 50% from our observation. So, from this point of view, we can say that it is difficult to achieve a high coverage for these preference-related branches, even with the two of the most effective testing tools. However, with Prefest, the branch coverage can be easily improved to around 90%.

Prefest detected additional five bugs, as shown in Table 3. These bugs are all preference related, which can only be found by testing specific functions under specific preference settings, and are not detected by *Stoat*. The reason is that *Stoat* usually missed some specific values of specific preferences, or sometimes even missed the setting screens, due to its random nature. The first bug causes data leaks while the others cause app crashes, which were logged as error messages by Android system. All bugs have been reproduced. Only the bug in *vanilla* is two-preference relevant while the rests are one-preference relevant. Particularly, the first four bugs were revealed for the first time, and we posted issues on GitHub. The last bug has been reported by others. So far, bugs in *KISS*, *vanilla* and *AmazeFileManager* have been confirmed and fixed by developers. Especially, the bug revealed in *vanilla* was an old one introduced over one year ago, and developers were happy to know the root cause and be able to fix it. There is no response for the other two bug issues, and we noticed that these two projects are no longer maintained. Nevertheless, since they cause app crashes or data leaks, we are confident that they are real bugs.

Table 3: Bugs detected by Prefest

App	GitHub Issue URL
GoodWeather	github.com/qqq3/good-weather/issues/54
Radiobeacon	github.com/openbmap/radiocells-scanner-android/issues/223
KISS	github.com/Neamar/KISS/issues/1136
vanilla	github.com/vanilla-music/vanilla/issues/898
AmazeFileManager	github.com/TeamAmaze/AmazeFileManager/issues/1400

4.3 RQ2: Efficiency

To answer RQ2, we recorded the test time and the numbers of test-runs of *Default* and Prefest(T) on the 30 apps in Table 2. Time consumed by Prefest(T) consists of the preference analysis time and the test execution time, and Table 2 shows the total time, with the analysis time in parentheses.

Compared to 107 minutes and 232 test-runs took by *Default* on average, Prefest(T) only took 24 minutes and 18 test-runs, contributing 22.4% and 8.0% to those of *Default*. The reason is that Prefest(T) aims at only the unreached blocks, and thus, just needs to execute part of the test cases. Meanwhile, as discussed, Prefest(T) has a good performance on code coverage and bug detection, showing the value of our proposed "enhanced testing".

The results also show the efficiency of our static and dynamic combined analyses. For 29 out of all 30 apps, 23 apps took just about 1 minute to conduct the analyses, and the other 6 apps took no more than 4 minutes. Only app —*Signal*, took 11 minutes due to its large app size and long execution flow. However, the time cost is still acceptable, compared with the original test time, and more time spent on complex apps, we believe, is worthwhile.

4.4 RQ3 & RQ4: Comparative Study

To answer RQ3 and RQ4, we run experiments on the example of *GoodWeather* and the seven apps from existing researches with *Default*, Prefest(T), Prefest(N), *NonDefault* and *Pairwise*, and recorded the results in in Table 4 and Table 5.

Table 4: Comparison of the instruction and branch coverages of different strategies

Subject	*Default*		PREFEST(T)		PREFEST(N)		*NonDefault*		*Pairwise*	
	Inst.%	Branch%	Inst.%	Branch%	Inst.%	Branch%	Inst.%	Branch%	Inst.%	Branch%
GoodWeather	60.61	35.10	68.33	47.05	70.12	49.41	67.28	46.02	70.46	51.18
A2dpvolume	40.03	17.39	41.56	20.31	41.70	20.93	41.42	19.96	42.38	21.07
Alwayson	44.53	30.71	46.10	33.63	47.64	35.36	45.90	33.18	47.23	34.91
Suntimes	39.65	29.25	42.69	32.58	43.94	34.81	42.18	32.23	43.67	34.76
Opensudoku	44.61	32.53	46.76	36.60	47.29	37.80	46.42	35.62	47.67	37.65
Radiobeacon	37.18	19.90	39.60	21.28	39.80	21.51	39.24	21.46	40.06	22.29
Notepad	51.97	39.84	55.19	47.81	55.19	46.08	54.26	43.96	55.63	45.56
Wikipedia	43.14	27.31	45.66	29.23	49.25	32.49	45.25	28.33	48.81	32.51
Average Improvement%			**6.4%**	**14.8%**	**8.9%**	**19.3%**	**5.4%**	**11.9%**	**8.9%**	**19.8%**

Table 5: Comparison of test-run numbers and test time of different strategies

Subject	*Default*		PREFEST(T)		PREFEST(N)		*NonDefault*		*Pairwise*	
	#Run	Time(min)	#Run	Time(min)	#Run	Time(min)	#Run	Time(min)	#Run	Time(min)
GoodWeather	360	137	14	14	2035	1766	360	453	3600	4295
A2dpvolume	180	73	6	8	1080	1473	180	357	1800	3212
Alwayson	240	121	15	22	1049	1365	240	754	2880	8150
Suntimes	150	88	22	32	1497	3446	150	397	1800	4975
Opensudoku	120	65	10	12	129	169	120	231	1200	2482
Radiobeacon	180	118	16	17	294	355	180	318	1800	3371
Notepad	321	104	56	48	1661	1357	321	292	2889	2622
Wikipedia	180	123	17	32	1429	2529	180	427	2160	5728
Average Percentage			**9.0%**	**22.2%**	**522%**	**1557%**	**100%**	**393%**	**1037%**	**4360%**

Table 4 lists the instruction and branch coverages, and Table 5 lists the test time and the number of test runs. In general, we have *Pairwise* > PREFEST(N) > PREFEST(T) > *NonDefault* in the improvement of coverages and *Pairwise* > PREFEST(N) ≫ *NonDefault* ≫ *Default* > PREFEST(T) in the test time and the number of test-runs. Being the simplest way to conduct perfence-wise testing, *NonDefault* showed the poorest performance in improving code coverage, and took more time than PREFEST(T). This demonstrates that a more sophisticated approach for preference-wise testing is needed.

From Table 4 we can see that *Pairwise* and PREFEST(N) have the best and similar performance in improving instruction and branch coverages, which are 8.9% and over 19% improvement for instruction and branch coverages, respectively. The marginally lower branch coverage of PREFEST(N) than *Pairwise* is because PREFEST(N) missed some relevant preferences due to the short-circuit evaluation in the compiling stage. As Soot works on Java ByteCode, these short-circuit preferences were not analyzed. However, such cases are extremely rare, and thus, we can consider the effectiveness of PREFEST(N) and *Pairwise* as equivalent. PREFEST(T) comes next in effectiveness, with 6.4% and 14.8% improvement for instruction and branch coverages, respectively. A main reason for more coverage of *Pairwise* and PREFEST(N) compared with PREFEST(T) lies in that, for few blocks, their behaviors can vary under different preference option combinations. For example, some blocks, responsible for displaying GUIs, can present different preferences on setting screens, depending on the value of a certain preference, e.g., a switch deciding to display or hide a sub-menu of preferences. These cases cannot be handled by PREFEST(T), but with a more exhaustive trying of different preferences, PREFEST(N) is able to process most of them.

Nevertheless, PREFEST(T) still retained 72% and 77% improvement in instruction and branch coverages of those of PREFEST(N) and *Pairwise*. Considering its time cost, we still consider PREFEST(T) as the best approach, for its balance on effectiveness and cost. As Table 5 shows, the *Pairwise* approach was extremely time-consuming, by taking over 43 times of the original time cost. In fact, the comparative study on just these 8 apps took about 35 days, and we estimated that over four months would be needed to scale the study to all the 30 apps. The fact that 24 days were used in applying *Pairwise* on the 8 apps indicates the necessity of our PREFEST. By removing irrelevant preferences in combinations, PREFEST(N) reduces the time by about two third of the time cost of *Pairwise*, but still needed more than 15 times of the original test time. In contrast, PREFEST(T) only took about half an hour to perform the tests, accounting to just one-fifth of the original test time.

Nowadays, fast developing cycle is the key to the success of Android app development due to the fast-changing mobile markets, and developers typically can only spare a little time for testing. The Target Mode, which tries to keep a good balance between test efficiency and effectiveness, is more likely to be attractive to developers. If app quality is critical and time recourse allows, developers can still choose PREFEST(N) for its best effectiveness in coverage but much less time cost than *Pairwise*. However, as the experiments show, the effectiveness in bug detection for PREFEST(N)

and Prefest(T) is the same: all bugs found by Prefest(N) and *Pairwise* were found by Prefest(T).

4.5 Threats to Validity

4.5.1 Internal Threats. The major threat comes from that the original test cases may include some operations of setting preference options, which will change some option values set by Prefest and result in executing different code parts than planned. To mitigate this threat, Prefest takes into account the effects of some simple preference setting methods, such as *setBoolean()*, *setString()*, when calculating the values of preference options.

Another threat comes from Soot, which we use to perform the analysis. Soot works on Java ByteCode, so the short-circuit evaluation in the compilation phase would lead to the missing of relevance between preferences and test cases. An alternative analysis framework based on original source code can solve the problem, which we plan to study in the future.

The third threat comes from that the current implementation of Prefest takes Stoat's way of focusing on error messages produced by the Android system, and does not consider test assertions. If one needs assertions in the test cases, since Prefest generates new tests with different preference settings, new assertions will be needed.

4.5.2 External Threats. The main external threat is that our evaluation results may not be generalized on other Android applications. Our experiments were performed only on thirty apps, since the experiments were time-consuming. It is possible that the effectiveness may vary for other apps. However, this problem is alleviated since the complexity of the thirty apps has enough diversity for ranging from 5k to over 200k instructions, and several apps are also widely used in real-world such as Wikipedia and Signal.

As Prefest has only worked with Stoat, another threat arises from whether Prefest can work with other test input generation approaches. We mitigate this threat by implementing Prefest into an independent tool which takes test cases as the direct input. In this case, Prefest can easily cooperate with other tools, as test cases can be easily obtained from these tools' log files. Certainly, manually written test cases are also welcome to Prefest.

5 RELATED WORK

In this section, we will discuss relevant researches from Android testing and combinatorial testing.

5.1 Android Testing

Nowadays, frameworks and tools that automate the execution of tests are widely spread in industry, such as Robotium [28], monkeyrunner [25] and Appium [5]. To further improve the automation, many research approaches are proposed for the automated generation of test inputs, based on fuzzing testing techniques [2, 19], model-based testing techniques [3, 7] and search-based techniques [20, 21]. Several researches also apply symbolic execution or concolic execution to Android testing: Mirzaei et al. [23] present *SIGDroid*, which combines model-based testing with symbolic execution to systematically generate test inputs for Android apps; Anand et al. [4] illustrate the technique *ACTEve*, which treats touch on screen as user inputs and generates sequences of events automatically and systematically with concolic execution to alleviate the path explosion problem. These approaches are similar to Prefest, as Prefest also performs a dynamic analysis similar to concolic execution. However, Prefest only focuses on preferences and analyzes the execution of given test cases, so it consumes much less test cost and is less affected by app size. Also, most preferences' options are enumerable, in which case the exact option values to reach the targets can be obtained by enumeration, while for SIGDroid and ACTEve, a constraint solver is needed for specific values, which can be more time-consuming. Most importantly, the application scenarios and purposes are different: Prefest works based on existing tests to enhance their performance in terms of preferences, while both *SIGDroid* and *ACTEve* try to generate new tests for given apps.

5.2 Combinatorial Testing

Combinatorial Testing has been an active field of researches in the last twenty years [26]. One of the major trends in this area has been towards minimizing the size of test sets for a given combinatorial criteria, with greedy and heuristic algorithms [10, 11, 18, 34], genetic algorithm [22, 30], or even artificial intelligence [1].

Recent years, these combinatorial optimization techniques are also adapted in Android testing. Two studies are closely related to this paper, one is *TrimDroid* [24], an approach that statically extracts dependencies among widgets to reduce the number of combinations in GUI testing; and the other one is *PATDroid*, which performs a hybrid program analysis that excludes irrelevant permissions to reduce unnecessary permission combinations for test cases. Compared with TrimDroid, Prefest uses both static and dynamic analyses on the *AUT* and the existing test cases to perform the preference-wise testing under certain preference option combinations, while TrimDroid employs static analysis over the *AUT*, to automatically generate test cases. Compared with PATDroid, Prefest targets at preferences, which are more difficult to analyze as their values are passed through execution flows. In addition, PATDroid uses manual written test cases while Prefest uses test cases generated from automatic testing tools, which are usually of huge size, bringing more difficulty for reduction. In summary, we propose the Target Mode in Prefest that can reduce the test cost to a plausible level.

6 CONCLUSION

We present Prefest, a preference-wise enhanced testing approach for Android applications. With a static and dynamic combined analysis, Prefest gives an automated solution to test apps only under necessary preference option combinations with existing tests, in which a Target Mode is proposed for further reduction in test cost. Our experiment results show that within 1% test cost compared to tests under pairwise combinations of preferences, Prefest achieves a 6.8% and 12.3% improvement in code and branch coverages. Moreover, we also found five additional preference-related bugs in real-world apps using Prefest.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program of China (Grant No. 2017YFB1001801) and the National Natural Science Foundation of China (Nos. 61690204, 61632015).

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] AppiumConf. 2019. Appium : Automation for Apps. http://appium.io/. [online, accessed 15-Feb-2019].

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[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] 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.

[9] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?(E). (2015), 429–440.

[10] 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.

[11] 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.

[12] Google Developers. 2018. Documentation of Settings for Android Developers. https://developer.android.com/guide/topics/ui/settings. [online, accessed 01-Sep-2018].

[13] Google Developers. 2019. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. [online, accessed 15-Feb-2019].

[14] EclEmmaTeam. 2019. JaCoCo: Java Code Coverage Library. https://www.eclemma.org/jacoco/. [online, accessed 15-Feb-2019].

[15] 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.

[16] 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.

[17] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[18] 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*. IEEE Computer Society, 549–556.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] monkeyrunner. 2019. monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/. [online, accessed 15-Feb-2019].

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

[27] pcqpcq. 2019. open-source-android-apps. https://github.com/pcqpcq/open-source-android-apps/. [online, accessed 15-Feb-2019].

[28] RobotiumTech. 2019. Android UI Testing Robotium. https://github.com/RobotiumTech/robotium. [online, accessed 15-Feb-2019].

[29] 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.

[30] 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*. IEEE Computer Society, 72–77.

[31] Skylot. 2019. jadx : Dex to Java decompiler. https://github.com/skylot/jadx/. [online, accessed 15-Feb-2019].

[32] 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*. IEEE Press, 27–37.

[33] 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*. ACM, 245–256.

[34] 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.