Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 7, July 2019, pp. 2008-2018
DOI: 10.23940/ijpe.19.07.p28.20082018

Equivalent Version Sets Testing Method for Android Applications based on Code Analysis

Song Huang, Sen Yang*, Yongming Yao, and Lele Chen

Command and Control Engineering College, Army Engineering University of PLA, Nanjing, 210007, China

Abstract

The Android system is an open source mobile operating system that has been released in numerous versions. Android fragmentation is becoming more and more serious. This paper shows how different Android runtime environments affect test coverage results. To address this limitation, we run apps on all Android versions to collect coverage rate and also present an algorithm to generate equivalent test runtime-environment-set to exercise mobile apps. Our approach is to systematically test the targeted code of Android apps based on code analysis. It analyzes the decompiled code that identifies the code related to the Android SDK version and then generates the corresponding test cases. An empirical study of the practical usefulness of the technique is presented for six widely-used industrial apps. The test result shows that our equivalence test runtime-environment-set only requires less than half of all versions, which dramatically reduces the test resources. Moreover, the method coverage of these applications increased by an average of 49.3% on all versions and 46.8% on equivalent test runtime-environment-set, respectively.

Keywords: mobile testing; code analysis; android version; runtime environment; set generation

(Submitted on May 21, 2019; Revised on June 27, 2019; Accepted on July 20, 2019)

1. Introduction

Today, millions of apps occupy the daily lives of people and play a vital role in conducting business and news, personal well-being affairs, entertainment, and so on. With the rapid rise of mobile devices, people's demand for mobile applications is continuously increasing, and the number of mobile applications is growing exponentially. Nowadays, ensuring their quality is a problem for developers. The characteristics of the mobile application are different from desktop applications. First of all, mobile applications depend on the vibrant event-driven GUI events to achieve their function. Secondly, the development life cycle of mobile applications is short and fast, and the costs of manual testing are extensive and time-consuming. The effectiveness of existing test automation tools is non-ideal. Therefore, the effective testing of mobile applications is becoming more and more critical.

At present, there are two main categories of apps in the market. The first is open source apps. Non-profit organizations, such as F-droid, usually develop them. The programming difficulty of these applications is usually not too high, and their functions are relatively simple. Most of the previous testing tools are aimed at testing these open source apps. For these open source apps, their structure is not too complicated, and their code coverage is relatively easy to count. Moreover, many tools have to use the source code of these apps to test, such as ACTEVE [1]. However, because the functions of these apps are often too simple, their usage in people's daily lives is so low that most people have never heard of many of them. The other type of app is closed-source apps, such as those found in the Google Play Store, which are usually developed by industrial companies like YouTube. In order to earn more users' favor, they usually add more functions to the app. Thus, the robustness of these apps is often stronger, and users are more likely to use these apps. From a practical point of view, some testing tools perform well on open source apps, but if they cannot adequately test widely-used closed source apps, their practicability will be significantly reduced.

* Corresponding author.
E-mail address: yangsen0310@qq.com

It is mentioned in [2] that in open source apps, the existing tools [3-8] all perform well. However, when they are applied to industrial apps (like Wechat, CNN, and Word), the test coverage they achieve is too low to meet the requirements. Why is it that many of the approaches that have been proposed do not perform as well when applied to industrial apps as they do on open source apps? In probing the reason, we found that the application of industrial apps, becoming as compatible as possble with different system versions of mobile phones, will write a large amount of code to handle the API call from different Android versions. When we test the app in a single version (like Android 4.4), we congenitally miss these code; consequently, no matter how we improve the test method, the test coverage will remain uncovered on these branches. Meanwhile, [9] mentioned that configurations in Android testing do matter in testing and previous works did not consider it, but it did not point out how to effectively solve this problem.

To solve this problem, we analyze the codes dealing with different Android versions in the source code of decompiled industrial apps. Then, we give the equivalent runtime environment set of Android runtime versions, which further improves the coverage of the test and can actively locate the defects caused by the Android runtime environment. This approach is a new way to solve the problem of low test coverage caused by the runtime environment from the perspective of source code analysis.

The contributions of this work are:

(1) A novel approach to generate test runtime environment sets to exercise apps. Our approach, based on code analysis, is sufficiently useful to solve fragmentation problems of Android environment runtime.

(2) An efficient algorithm to increase the method coverage related to the runtime environment. By checking the decompiled code, we generate the equivalent runtime environment set to run a random test, thereby alleviating the low coverage problem.

(3) An empirical study of the practical usefulness of the algorithm has been presented for six widely-used industrial apps. The testing result shows that our equivalence test runtime-environment-set only needs less than half of the original versions set, which can significantly reduce test resources. Moreover, the method coverage of these applications increased by an average of 49.3% on all versions and 46.8% on the equivalent test runtime-environment-set, respectively.

2. Background: Android Versions

The Android system is a free and open source operating system based on Linux, and it is mainly used in mobile devices such as smartphones and tablets. It was developed by Google and the Open Handset Alliance. Its initial version was released on November 5, 2007. The Android operating system was initially developed by Andy Rubin and mainly used in mobile phones. Before Android was officially released, it had two internal test versions, named after famous robots: Android Beta and Clockwork Robot (Android 1.0). Since May 2009, Google has changed its naming rules to use desserts as code names for their system versions, which begin with capital letters in alphabetical order, starting with C: Cupcake (Android 1.5), Donut (Android 1.6), Clair (Android 2.0/2.1), Froyo (Android 2.2), Gingerbread (Android 2.3), Honeycomb (Android 3.0), Ice Cream Sandwich (Android 4.0), Jelly Bean (Android 4.1/4.2), KitKat (Android 4.4), Lollipop (Android 5.0), Marshmallow (Android 6.0), Nougat (Android 7.0), Oreoid (Android 8.0), Pie (Android 9.0), and Q Beta (Android 10.0). These subsequent versions are further improved and optimized based on the previous version. They fix the bugs existing in old versions, delete some functions that are no longer needed, and add many new functions that were not available in the previous version.

As can be seen from Google's latest statistical report on Android's market sharing by the end of October 2018 in Table 1, Android 7.0 and Android 7.1 accounted for 28.2% of the total market share, while Android 6.0 accounted for 21.6%, ranked second, and the gap was not significant. Android 5.x and Android 8.0 were also around 20%. Android 4.x and Android 4.x still have a part of the market share. Also, few people use versions before Android 4.0. As can be seen from Figure 1, many Android system versions exist on user devices at present.

Google has provided the corresponding Android SDK (software development kit) for each Android version for app developers to program. Because there are so many different versions of Android and the Android SDK updates quickly, many new features and APIs (application programming interface) may not be available in the lower version. Many industrial apps are devoted to facing as many user devices as possible, so the development process of these apps may be as robust as possible. To keep the downward compatibility of the new functional interface, the app usually has a minimum version configuration in the development process. For example, if the app is only compatible with the version Android 4.2 and above, it should set "minSdkVersion=17" in the APK (Android Package), which indicates that it can be downward compatible with the Android 4.2 version. This app can normally run on the Android 4.2 version, because some new features have been eliminated during

its running to guarantee that there will be no crash. Avoiding this problem requires developers to focus on compatibility and adaptation in the code.

Table 1. Market share of Android SDK

Version	Codename	API number	Distribution
2.3.3 -2.3.7	Gingerbread	10	0.2%
4.0.3 -4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.1%
4.2.x		17	1.5%
4.3		18	0.4%
4.4	KitKat	19	7.6%
5.0	Lollipop	21	3.5%
5.1		22	14.4%
6.0	Marshmallow	23	21.3%
7.0	Nougat	24	18.1%
7.1		25	10.1%
8.0	Oreo	26	14.0%
8.1		27	7.5%
9.0	Pie	28	<0.1%
10.0	Q Beta	29	<0.1%

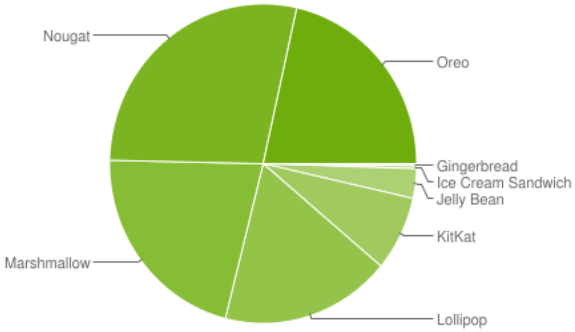

Figure 1. Market share of Android SDK

Overall, there are over 17 Android versions, and most of them still occupy a particular market share. Many mainstream industrial apps need to add a large amount of code to handle the implementation of different Android system versions in the development of the Android app, which means that they can adapt as many Android versions as possible. Therefore, when we test these widely use industrial apps, we need to consider how to cover these codes.

3. Overview of OurApproach

In this section, we will illustrate our approach using an app named My Piano Play app from the Google Play Store, which has over 5 million installations and whose main UI interface is shown in Figure 2. We first analyze the app by decompiling the apk (Section 3.1). We then describe the equivalent runtime environment set building process of this app (Section 3.2). Finally, we test this app and show how we improve the method coverage with our approach (Section 3.3).

Figure 2. Main UI interface of My Piano Play

3.1. Decompile the My Piano Play App

Many of the functions of the Android app are implemented by calling a series of API interfaces, which are written in Java. At the end of app development, the compiled byte code, resource files, and other data are compressed together, through the AAPT tool, to form an Android package (APK). APKs are downloaded to Android devices by users to install related applications. The APKs are Zip files, but the suffix name is changed to "apk". After unzipping the apk, we can see the Dex (Dalvik VM executes) files in the APK. These are not Java ME byte code but Dalvik byte code. Also, the APK has META-INF folders to store program entry information, res folders to store resource files, Android Manifest files to store global configuration information of app, and resources files to store compiled binary resource files.

Apk decompilation involves decompiling the Dalvik byte code and resource files in APK files by using APK compilation tools. The decompiled files can be reprogramed, recompiled, and packaged to achieve the purpose of personalized customization. Through the decompiling app, we can obtain the java code like Figure 3. Because app developers also consider that decompilation of code may lead to leakage of core technology, they often obfuscate the source code. The obfuscated code does not make the code unable to decompile but renames the class, method, variable, and other information in the source code and replaces them with meaningless names, such as numbers and letters. However, the system API call in source code is usually not obfuscated, and the general level of obfuscation does not affect the implementation of our algorithm.

```
private void startSoundUpdate ( )
{
  this.piano.ceremony ( ) ;
  this.soundUpdater=new SoundUpdater ( this ) ; this.soundUpdater.start ( this ) ;
  ……
  catch ( Exception localException )
  {
  GoogleServiceManager.startAdLoad ( ) ;
  }
}
```

Figure 3. Decompiled code snippet of the main class

3.2. Build the Runtime-Environment-Set

To further analyze the app source code, we can find the system API call judging the current version of the system. As shown in Figure 4, in this branch, apps can invoke the interface if the current runtime SDK is equal to or greater than API 17. If we test on a lower version, this branch will always remain uncovered no matter how we improve the testing strategy. Therefore, we need to count where these similar codes appear in the source code and test these branches directionally. At the same time, it is worth noting that we only analyze these branches in the main code class of the app source code, rather than the similar code in the third-party packages.

Android currently has 17 API versions. By counting the code related to the branch judging the system version in apps, we can divide all the API versions into separate subsets, such as {[16,17), [17,19), [19,21), [21,23), [23,29]} for My Piano Play. These sets are mutually exclusive, and then we need to select a version number from each mutually exclusive subset randomly. Based on these sets, an equivalent runtime-environment-set can be formed, such as {16,17,19,21,27}. Next, we need to test My Piano Play independently on each set, collect the final code coverage respectively, and merge them to get joint test coverage.

```
if ( Build.VERSION.SDK_INT >= 17 )
{
localWindowManager.getDefaultDisplay ( ) .getRealSize ( localPoint ) ;
this.real_width= Math.max ( localPoint.x, localPoint.y ) ;
this.real_height =Math.min ( localPoint.x, localPoint.y ) ;
int i = Math.round ( this.real_width / ( 0.4F * this.xdpi ) ) ;
this.key_len = i;
this.key_maxlen = i;
if ( this.key_len >= 8 )
  {
      break label728;
  }
this.key_len = 8;
}
```

Figure 4. Version-related code snippet

3.3. Test and Evaluation

We chose Monkey as the test tool to exercise the app, because [2] pointed out that Monkey is still one of the most efficient Android test tools in the widely-used industrial apps. Firstly, we test on all system versions of devices. We inject about 10,000 random events into each device for testing. This process can be carried out in parallel to increase the efficiency of testing.

For our My Piano Play example, using the equivalent runtime environmental set, our approach explores 18.2% method coverage, compared with 7.9% by the widely-used Android 4.4 in previous works. Moreover, we found that this app failed to run on Android 4.3, which may be a compatibility problem.

4. Generating Test Set

In this section, we describe how our method works in detail. We begin with an explanation of the decompiling process for apps (Section 4.1). We then illustrate the generation algorithm of our equivalent runtime environment version set (Section 4.2). Finally, we discuss how to collect the final code coverage (Section 4.3).

4.1. Decompile the Apk

The Android APK packaging process is shown in Figure 5. In the beginning, the Java code of the app is compiled by Java Compiler to generate class files, which then become DEX files by the DX tool. Then, the code and resource files are packaged by apkbuilder, and the jarsigner is used to sign the developer's signature. Finally, the signed APK may be aligned by zipalign. Each APK file contains the DEX file, which records the information of all class files of the entire Android project. It is an eight-byte binary stream file that can be run directly on the Android Dalvik virtual machine. The DEX file can be seen as a list of sections, where each section contains a list of items, and each item may nest a structure data. The information of the section list (type, number of sub-items, and file offset) is eventually recorded in the section called MAP_LIST. The decompiling APK needs to convert DEX files into intermediate code and then translate them into JVM instructions after optimization. Finally, these instructions are combined to form original class files. This operation can be implemented using the dex2jar package. After decompilation, we can obtain the source code corresponding to APK. Next, we need to search the code related to runtime environment judgment.

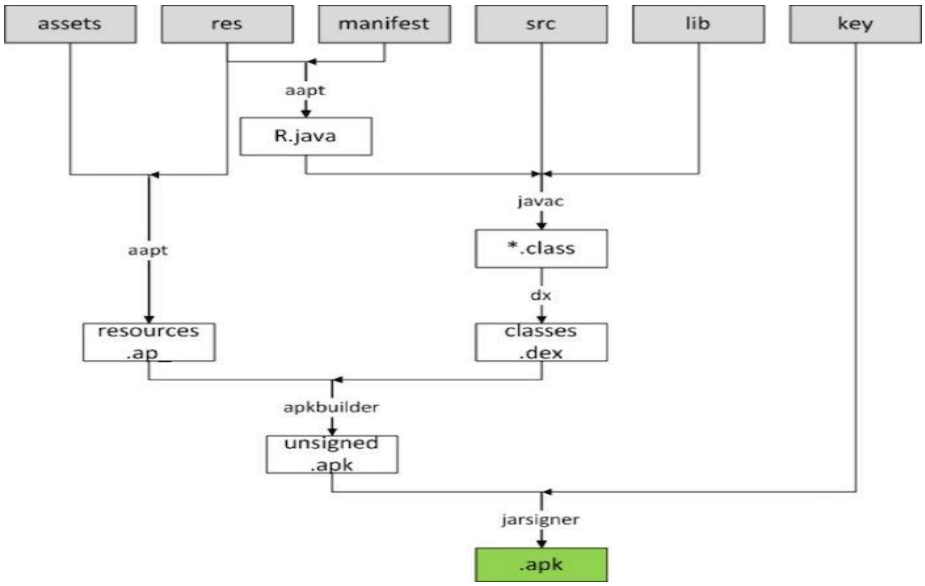

Figure 5. Package procedure

4.2. Build the Best Set of the Android Test Runtime Version

4.2.1. Set Partition of Android Version

We define M_0 as the set to be partitioned. At first, there is only one element in the initial M_0: { [minVersion, maxVersion]}, which is the set of all Android versions. After each update of the set, one element in the original set may be divided into two

elements. When we search in the source code, we find that there is a system API call "Build.VERSION.SDK_INT". This API represents that the program is judging the version of the Android system currently running. Because this call is not vital in the main class of the app, it is rarely added to operations such as obfuscation. Most of the relevant code in the app is easy to locate. After this call in the source code, the current system version is often compared with a **target version number** (like in Figure 4). If the current system version satisfies the condition of comparison, a corresponding branch can be executed. If not, another branch may be executed. This logic comparison is usually equal to (=), greater than (>), less than (<), less than or equal to (<=), greater than or equal to (>=), etc. After the comparison operator, there is often a target version number to compare with. At this time, we first need to find the set corresponding to the comparison value from the M_0 (lines 6-8). Then, we need to classify conditions, and the algorithm flow is shown in Algorithm 1:

a) We discuss that the operator is an equality operator (lines 7-8). If the current set contains only one element, the set will not be updated. On the contrary, the original set will be divided into two subsets,:one subset contains only the **target version number**, and the other subset is the complementary set of the former subset.

b) We discuss that the operator is a less than operator (lines 10-11). If the current set contains only one element, it will not be updated. Otherwise, it will be judged whether the **target version number** is at the boundary point or not. If it is at the boundary point, the set retains its original state. If not, the set is divided into two subsets from the **target version number**: the left subset is **open** at this point, and the right subset is **closed** at this point.

c) We discuss that the operator is a less than or equal operator (lines 13-14). If the current set contains only one element, it will not be updated. Otherwise, it will be judged whether the **target version number** is at the boundary point. If it is at the boundary point, the set retains its original state. If not, the set is divided into two subsets from the **target version number**: the left subset is **closed** at this point, and the right subset is **open** at this point.

d) While the greater than operator (lines 16-17) and greater than or equal operator (lines 19-20) are similar to the less than operator and less than or equal operator, we need to change the choice of subset partition.

Algorithm 1: Set partition

Function: Update (M_0,Version)
Input: Partitioning set M_0, related code Version
Output: Partitioned set M_0

```
1.  foreach ( set in M0 )
2.    leftborder=set.left ( )
3.    leftcontain=set.leftcontain ( )
        //judge left border closed or open
4.    rightborder=set.right ( )
5.    rightcontain=set.rightcontain ( )      //judge right border closed or open
6.  if ( leftborder<version.number<rightborder|| ( version.number=leftborder&&leftcontain=true )
    || ( version.number=rightborder&&rightborder=true ) )      //judge the number in or out the set
7.    if ( version.choose="=" )
8.      Equal ( set,version.number, M0 )
9.      break
10.   else if ( version.choose="<" )
11.     Less ( set,version.number, M0 )
12.     break
13.   else if ( version.choose="<=" )
14.     Lessequal ( set,version.number, M0 )
15.     break
16.   else if ( version.choose=">" )
17.     Great ( set,version.number, M0 )
18.     break
19.   else if ( version.choose=">=" )
20.     Greatequal ( set,version.number, M0 )
21.     break
22.   else
23.     Else ( set,version.number, M0 )
24.     break
```

4.2.2. Generate the Equivalent Runtime Environment Set

For an app source code, we need to inspect its main class to find the branch related to the system runtime version. The main class here refers to the Java source code under the SRC file in apps. We only look at this part of the source code to find the relevant code. In other third-party libraries, although there are relevant codes, their stability and correctness have been tested

by the publisher (generally Google), and generally no bug will appear. Focusing on the main class can further narrow the scope of our research, and we can spend more effort testing the code written by app developers to find problems more efficiently. By traversing the source code (lines 2-7), we find all the branches related to the runtime environment in the main class, which divide the initial set into several mutually exclusive subsets. Then, we select a version from each subset (lines 9-13) and generate the equivalent runtime environment set to exercise the app. The specific flow is shown in Algorithm 2.

Algorithm 2: Generate equivalent runtime environment set

Function: GetBestSet (Code)
Input: App source code
Output: Equivalent runtime environment set M

```
1.  M0 = { [min, max]}   // initialize the set
2.  foreach（line in Code）
3.    if ( line contains "…SDK_INT" )
4.       Version=line.findVersion ( )
5.       Update ( M0,Version )
6.    else
7.       continue
8.  M={}
9.  foreach ( set in M0 )
10.   if ( set has one member )
11.     M=M∪{set.select ( ) }
12.   Else
13.     M=M∪{set.randomselect ( ) }
```

4.3. Test & Coverage Collection

In the process of testing, we choose the strategy of best tool testing, which involves using the best performance automated testing tools to test each app. When we run a test, test programs run independently on different Android versions, which can reduce the time of testing and improve the efficiency of our testing. When testing each device, record files will be formed. These files record in detail which methods in apps have been covered. After testing, we pull these data files from the devices and count the methods covered in the test, and many of them may be covered more than once. We then merge the method coverage data to obtain the joint coverage as a test result on the equivalent version set.

5. Empirical Evaluation

In this section, we elaborate on the empirical evaluation of our algorithm. The evaluation aims to answer two questions:

RQ1. Does the runtime environment impact the performance of testing coverage?

RQ2. Can we improve method coverage by establishing an equivalent runtime environment set?

First, we introduce the experiment settings and the choice of APP (Section 5.1). According to Question 1, we design a comparative experiment to verify the importance of Android version when testing industrial apps (Section 5.2). According to Question 2, we design a comparative experiment to verify the effectiveness of this algorithm in the equivalent test runtime environment by comparing the code coverage of the full version set (Section 5.3).

5.1. Evaluation Setup

We use Monkey [3] for testing. The experimental platform runs on Ubuntu 14.04 system with eight cores (2.50 GHz Intel Core (TM) CPU) and 12GB RAM. We eliminate Android 2.3 and Android 4.0 because few use these systems now, and these systems are often error-prone and unstable in our study. We install 16-29 APIs of Android SDK on 13 emulators. In the experiment, we use official Android x86 emulators to test cases. Each emulator is configured with 2GiB of RAM. We use Acvtool [10] to instrument apps to collect method coverage. We select six widely-used apps in the Google market, most of which have been installed more than a million times and have been used in previous research works [2]. Table 2 shows selected industrial apps in detail. "#Version" shows the app version we downloaded. "#Install" indicates the installation numbers of the app, according to Google Play. "#Method" shows the number of methods in those apps. Many industrial apps are now multidex, which cannot be instrumented (see Section VI for details), so we failed to instrument more apps in Google Play in our study. We choose one open source app named "aLogcat" as the control group of open source apps. We run each test continuously by injecting 10,000 random events on each industrial app three times.

Table 2. Overview of selected industrial apps

APP Name	Version	Category	Install	Method
My Baby Piano	9.18.2	Parenting	5m+	22051
Marvel Comics	3.10.9	Comics	5m+	50385
Filters For Selfie	1.3.0	Beauty	1m+	17145
Mirror	1.0	Beauty	1m+	11343
Speedo-meter	1.3.2	Auto&Vehicles	1m+	11065
Andbible	2.11.1.246	Book&Tool	0.1m+	19312
aLogcat	1.0.3	Tool	<0.1m	1799

5.2. Study 1: *Does Runtime Environment Matter*?

Table 3 shows the statistic of the method coverage rate of each industrial app collected on each Android API under study. "X" means that the corresponding test failed to run on this API. Table cells with grey backgrounds indicate the highest method coverage over APIs. All coverage percentage numbers retain one decimal digit.

RQ-1 (Runtime environment). As can be seen from Table 3, firstly, method coverage varies dramatically over different APIs, which means that the runtime environment does impact the test performance of automatic test tools. Secondly, Android 4.4 (API 19) could be the ideal version for testing apps because all apps run successfully on it. However, there are many other choices to test the app. The variation of method coverage across 17 APIs is shown in Figure 6. Meanwhile, 18 crashes occurred when we ran this experiment. Moreover, many of those exceptions were applied only in specific runtime versions.

Table 3. Statistics of method coverage rate on selected apps by Monkey under study

API / APP	16	17	18	19	21	22	23	24	25	26	27	28	29
My Baby Piano	14.8	15.3	X	7.9	12.6	8.1	8.0	8.0	8.2	8.5	8.9	8.0	8.1
Marvel Comics	14.9	16.3	11.9	16.4	X	X	13.8	14.6	13.6	15.2	13.9	14.2	14.8
Filters for Selfie	28.7	19.4	X	17.0	25.2	20.9	15.3	14.9	13.8	14.7	17.3	15.7	13.4
Mirror	5.2	4.2	X	5.4	X	5.7	X	5.0	5.4	5.3	5.6	5.6	4.5
Speedometer	9.1	10.0	X	11.1	9.6	9.5	9.0	9.9	9.5	8.9	8.9	8.9	9.0
And bible	X	X	X	26.4	24.8	3.3	X	31.4	19.0	24.8	X	20.0	20.5
aLogcat	8.3	8.1	5.4	8.0	7.4	7.2	7.4	7.6	7.1	5.7	8.1	6.6	6.4

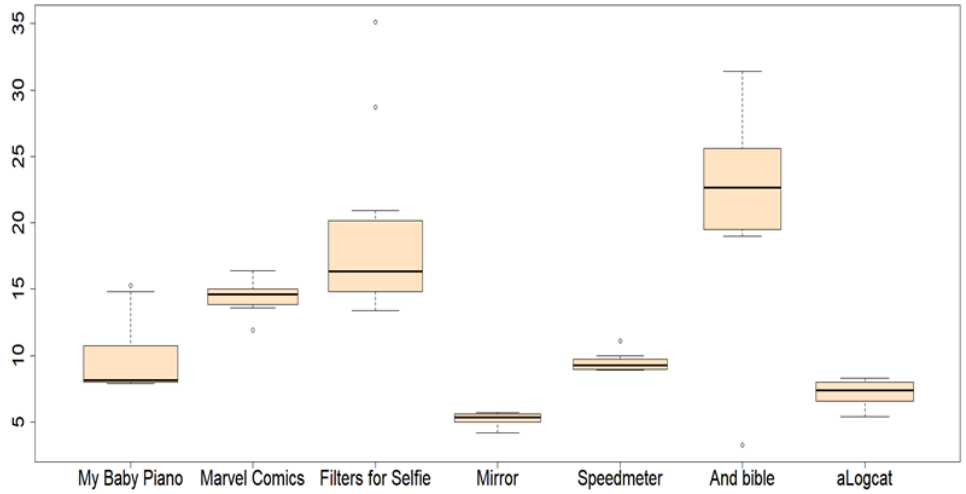

Figure 6. Variation of method coverage

Table 4 shows how the Android SDK impacts the performance of testing APP. "# Coverage on Single Version" shows the original strategy to exercise apps on a single Android version (like Android 19). "# Full Coverage" merges the method coverage of all runtime environments. "#Growth Rate" shows the growth rate of method coverage over coverage on a single version. As shown in Table 4, method coverage increased by 11.1%-130.4% based on the original strategy. Moreover, on average, our algorithm resulted in a 49.3% increase in method coverage. To be clear, we eliminate the app 'aLogcat' because this app is selected to show that the runtimes environment problem is not notable in open source apps.

Table 4. Results of method coverage rate

APP name	Coverage on single version	Full coverage	Growth rate
My Baby Piano	7.9	18.2	130.4%
Marvel Comics	16.4	18.3	11.6%
Filters For Selfie	17.0	31.9	87.6%
Mirror	5.4	6.0	11.1%
Speedometer	11.1	12.9	16.2%
And bible	26.4	36.7	39.0%
aLogcat	8.0	8.6	7.5%

5.3. Study 2: *Algorithm Effectiveness*

We run our algorithm on each selected app. Table 5 shows the subsets after set partition generated by Algorithm 1 and the equivalent runtime environmental set generated by Algorithm 2 (italic numerals represent test failures on this API version). "# Coverage on Single Version" shows the original strategy to exercise apps on a single Android version (like Android 19). "#Joint Coverage" merges the method coverage on the equivalent runtime environment set. "#Growth Rate" shows the growth rate of method coverage over original coverage.

Table 5. Results of method coverage rate on equivalent runtime environmental set

APP name	Subsets after set partition	Equivalent set	Coverage on single version	Joint coverage	Growth rate
My Baby Piano	{[16,17), [17,19), [19,21), [21,23), [23,29]}	{16,17,19,21,27}	7.9	18.2	130.4%
Marvel Comics	{[16,17), [17,19), [19,21), [21,23), [23,26), [26,28), [28,29]}	{16,17,19,***21***,24,26,29}	16.4	18.0	9.8%
Filters For Selfie	{[16,17), [17,18), [18,19), [19,21), [21,22), [22,23), [23,29]}	{16,17,***18***,19,21,22,27}	17.0	31.6	85.9%
Mirror	{[16,17), [17,18), [18,29]}	{16,17,22}	5.4	5.8	7.4%
Speedometer	{[16,21), [21,23), [23,29]}	{19,21,24}	11.1	12.2	9.9%
And bible	{[16,19), [19,24), [24,29]}	{19,21,24}	26.4	36.3	37.5%
aLogcat	{[16,29]}	{16}	8.0	8.3	3.8%

RQ-2 (effectiveness). As shown in Table 5, method coverage increased by 7.4%-130.4% based on the original strategy. Moreover, on average, our algorithm resulted in a 46.8% increase in method coverage. We also eliminate the app “aLogcat”. Table 6 shows the statistics of joint coverage and full coverage. “#Full Coverage” merges the method coverage on all Android APIs. “# Loss Rate” shows the decrease rate of method coverage over full coverage. As shown in Table 6, method coverage decreased by 0%-5.7% based on the joint coverage, which on average is 2.1%. In general, the equivalent runtime environmental set is efficient enough compared with the full version set.

Table 6. Statistics of two method coverage rates

APP name	Joint coverage	Full coverage	Loss rate
My Baby Piano	18.2	18.2	0%
Marvel Comics	18.0	18.3	1.6%
Filters For Selfie	31.6	31.9	0.9%
Mirror	5.8	6.0	3.4%
Speedometer	12.2	12.9	5.7%
And bible	36.3	36.7	1.1%
aLogcat	8.3	8.6	3.6%

In summary, our algorithm selects the representational API from the full runtime version set based on code analysis and dramatically increases method coverage. Unique crashes have been found in different APIs. Some of the failures occurred when the apps launched their main activity, and they can be reduced by enhancing the compatibility of the app.

6. Threats to Validity

There are several threats to the validity of our studies:

Internal validity: The main threat to internal validity is that our implementation did not include third-party libraries when we calculated method coverage. We think this may be the reason why our algorithm's performance varied on different

industrial apps. If the third-party libraries take a large proportion of the source code, the increase rate in the main class may seem negligible compared to the increase rate.

External validity: More and more developers use multidex to program their apps. We downloaded nearly 60 apps from Google Play, and more than 50 apps are multidex apps, which cannot be the instrument of current decompiled tools. However, method coverage collections tools (like Emma [11] and Acvtool [10]) need to instrument closed source apps to insert probe instructions after every method. Therefore, we evaluated only seven apps, and the average efficiency of our algorithm may vary for other apps.

7. Related Work

It is mentioned in [2] that existing Android test generation tools still have much room for improvement to handle industrial apps. Meanwhile, [9] mentioned that configurations in Android testing do matter in testing and previous works did not consider it, but this work did not point out how to effectively solve this problem.

To reduce manpower and improve test efficiency, there exist various test generation tools to automate testing apps. Google provides Monkey [3], which can generate random UI event sequences to an app under testing. Dynodroid [12] is a guided random testing tool that generates user UI events and system-level events. WCTester [4] improves Monkey's many shortcomings. First, it only triggers triggerable controls on every UI page. Second, it records test paths to avoid repeated tests. Finally, it preferentially selects controls that can change the status of the page to trigger. ACTEve [1] uses a dynamic symbol execution method to reduce redundant test events. EvoDroid [13] uses evolutionary algorithms to generate test cases to maximize test coverage, but it is no longer public. JPF-Android [14] validates and detects the specific attributes of apps based on Java Path Finder. Sapienz [5-6] is a testing tool using evolutionary algorithms that uses genetic algorithms to evolve shorter seed test sequences to maximize test coverage and improve fault detection. A3E [7] includes a systematic testing tool that performs a depth-first search strategy during exploration. GUIRipper [15] is a model-based testing tool that constructs a finite-state-machine (FSM) model of the UI and performs the depth-first search (DFS) exploration strategy. ORBIT [16] uses the same strategy as the former but adds static analysis of mobile application source code. SwiftHand [17] further improves the efficiency of testing by minimizing the number of application restarts. Stoat [8] proposes a guided stochastic model method, which uses the Gibbs sampling evolution model to change the model to achieve test coverage, aiming at code coverage and test case diversity.

8. Conclusions and Future Work

In this paper, we have presented an algorithm to generate an equivalent runtime-environment set for Android applications. By analyzing the decompiled code of the target app, we extract the runtime related code and partition the full runtime version set to generate the subsets. We advise testing apps with the best performance tools (see Section 7 for details) on the equivalent runtime environmental set to achieve equivalent test coverage. Our study results show that our approach outperforms past strategies for industrial app testing. In future works, we plan to study more industrial apps with automatic test tools and solve the multidex problem to exercise larger apps.

Acknowledgments

This work is supported by the National Key R&D Program of China (No. 2018YFB1403400).

References

1. S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1-11, ACM, Cary, North Carolina, 2012
2. W. Y. Wang, D. F. Li, W. Yang, Y. R. Cao, Z. W. Zhang, Y. T. Deng, et al., "An Empirical Study of Android Test Generation Tools in Industrial Cases," in *Proceedings of the 33rd ACM/IEEE International Conference*, 2018
3. "The Monkey UI android Testing Tool," (http://developer.android.com/tools/help/monkey.html)
4. H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, et al., "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *Proceedings of IEEE/ACM International Conference on Software Engineering*: *Software Engineering in Practice Track*, 2017
5. K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-Objective Automated Testing for Android Applications," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 94-105, 2016
6. K. Mao, M. Harman, and Y. Jia, "Crowd Intelligence Enhances Automated Mobile Testing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, Piscataway, NJ, USA, 2017

7. T. Azim and I. Neamtiu, "Targeted and Depth-First Exploration for Systematic Testing of Android Apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 641-660, New York, NY, USA, 2013
8. T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, et al., "Guided, Stochastic Model-based GUI Testing of Android Apps," in *Proceedings of Symposium on the Foundations of Software Engineering*, pp. 245-256, 2017
9. E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in Android Testing: They Matter," in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, New York, NY, USA, 2018
10. A. Pilgun, O. Gadyatskaya, S. Dashevskyi, Y. Zhauniarovich, and A. Kushniarou, "Fine-Grained Code Coverage Measurement in Automated Black-box Android Testing," 2018
11. V. Roubtsov, EMMA, Retrieved 2017-2-18 from http://emma.sourceforge.net/, 2017
12. A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 224-234, New York, NY, USA, 2013
13. R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 599-609, 2014
14. H. van der Merwe, B. van der Merwe, and W. Visser, "Execution and Property Specifications for JPF-android," *ACM SIGSOFT Software Engineering Notes*, Vol. 39, No. 1, pp. 1-5, February 2014
15. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012
16. W. Yang, M. R. Prasad, and T. Xie, "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications," *Fundamental Approaches to Software Engineering*, pp. 250-265, Springer Berlin Heidelberg, 2013
17. W. Choi, G. Necula, K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," *ACM Sigplan Notices*, Vol. 48, No. 10, pp. 623-640, 2013