A Framework for Detecting Compatibility-Issues-Proneness Apps Based on Multimodal Analysis in Android Platform

Chen Xu[1], Caimei Wang[2], Yan Xiong[1], Wenchao Huang[1], Zhaoyi Meng[1], and Fuyou Miao[1]

(Corresponding author: Caimei Wang)

School of Computer Science and Technology, University of Science and Technology of China [1]

Elec-3 (Diansan) Building, West Campus of USTC, Huang Shan Road, Hefei, Anhui Province, China

Department of Computer Science, Hefei University[2]

Building 38, No. 99, Jinxiu Avenue, Hefei, Anhui Province, China

Email: wangcmo@mail.ustc.edu.cn

(Received Jan. 6, 2021; Revised and Accepted July 5, 2021; First Online Nov. 13, 2021)

Abstract

With the prosperity of Android, the compatibility issues in apps cause security flaws and bring damages to the user experience. Unfortunately, recent studies cannot help the developer to identify the apps that are compatibility-issue-proneness. It motivates us to propose an automated approach to identify these apps derived from multimodal learning. To this goal, we first present some potential modalities of apps based on previous insight and then leverage statistical methods to test the modalities. Finally, we use the selected modalities to identify these apps on a real dataset. Experimental results on apps demonstrate the effectiveness of our work.

Keywords: Android; Compatibility Issues; Modality

1 Introduction

Android has been the largest mobile platform in the world, with 74.43 % market share of global smartphone shipments in Sept 2020 [28]. For profits and the competitive power, manufacturers (*e.g.*, Xiaomi, Huawei, Samsung) choose to release new devices and customized systems on the Android platform. At the I/O developer conference in 2019, Google announced that there are more than 2.5 billion active Android devices with 180 hardware manufacturers [7]. Given the large number of devices with different hardware and system configurations, it is a non-trivial task for Android developers to ensure their apps behave as expected among those myriad devices as possible [11]. The *cross-devices inconsistencies problem* is defined as the *compatibility issue* in many studies [30,33]. The compatibility issue may bring damage to user experience and cause security issues [11,34]. For this reason, it was reported that 94% of developers identified the issues as the main reason to cause themselves to avoid working on the Android platform [17,22].

The problem of identifying compatibility-issue-proneness apps is crucial because it is good for each stakeholder in the whole Android ecosystem. For example, from the view of the market maintainer, the compatibility issues that occur in apps can be detrimental to the user base. If the problem has been solved, the mobile app market can provide maintenance advice to developers. Besides, the developer can reduce the test efforts and fix the compatibility issue for targeted apps, which is also beneficial for the user experience. The potential value of solving this problem motivates our work.

Existing researches deal with compatibility issues on Android apps from several aspects. Some studies help Android developers to prioritize major test device via user reviews [17] or app usage data [22,33]. However, the developers still need to conduct extensive testing for each app on selected test devices. Some studies discover that compatibility issues derive from multiple reasons [30], such as device variations [17,22,31], complex user interfaces [11], API evolutions [13,15,20], etc.

PIVOT [31] discovers APIs in the Android framework which are caused by compatibility issues among different devices. DiffDroid [11] leverages a differential testing to automatically identify cross-platform inconsistencies in the UI of Android apps. API-evolution-based approaches [13,20] compute the additions and removals of Android framework APIs between consecutive API levels to find fragmentation-induced compatibility issues. Nevertheless, these automated approaches can only help to detect a specific type of compatibility issues.

Detecting compatibility-issue-proneness apps is a non-trivial and difficult problem, as our goal is to find the apps that are derived from multi reason. In this paper,

Table 1: Android OS distribution

Version	Codename	API	Distribution	Release Date
4.0.3-4.0.4	Ice Cream Sandwich	15	0.2%	Dec, 2011
4.1.x	Jelly Bean	16	0.6%	Jul, 2012
4.2.x		17	0.8%	Nov, 2012
4.3		18	0.3%	Jul, 2013
4.4	KitKat	19	4.0%	Nov, 2013
5.0	Lollipop	21	1.8%	Nov, 2014
5.1		22	7.4%	Mar, 2015
6.0	Marshmallow	23	11.2%	Oct, 2015
7.0	Nougat	24	7.5%	Aug, 2016
7.1		25	5.4%	Oct, 2016
8.0	Oreo	26	7.3%	Aug, 2017
8.1	Oreo	27	14.0%	Dec, 2017
9.0	Pie	28	31.3%	Aug, 2018
10.0	10	29	8.2%	Sept, 2019

we try to solve this problem by exploiting multi-modal heterogeneous app data from different source.

The information presents conceptual characteristics of apps, and thus is helpful in addressing our problem.

The challenge in our work is how to select and use the multi modalities of data to measure the degree of compatibility-issue-proneness. The previous researches only leverage one specific type of data and detect compatibility issue derived from a single cause. To solve the problem, we model the degree of compatibility-issue-proneness by leveraging multi-modal heterogeneous data. Specifically, we select some potential modalities from different source, such as app market, resource file and class files. Then we adopt statistical approach to test whether significant difference among the two samples in term of each potential modality. In the end, we use the selected modalities to identify the compatibility-issue-proneness apps.

Our work is evaluated on a large real-world dataset consisting of 7,526 Android apps in Google Play. The evaluation results show that our work is an effective approach to identify compatibility-issue-proneness apps.

This paper makes the following contributions:

- We provide an insight into the root causes of compatibility issues in Android apps and present potential modalities of app that may influence on the compatibility issues. To the best of our knowledge, such relations have not been empirically investigated yet.
- We study the problem of identify the modeling compatibility-issue-proneness apps based on multi-modal learning.
- We conduct the evaluation on our work among a real-world dataset.

The structure of this paper is as follows. In Section 2, we present the background of compatibility issues and APK files on Android platform. We describe our modalities extraction methodology and data processing steps in Section 3. We present the experimental results of modalities extraction and app identification in Section 4. We discuss threats to validity in Section 5. Related work is described in Section 6. We finally conclude and briefly mention future directions in Section 7.

2 Background

2.1 Compatibility Issues

Compatibility issues come up when an app may not suitable for all devices that carry it. Specifically, an Android app may present different outputs across devices, and do harm to user experience [11, 33]. For example, an app behaves as expected on a Huawei device, but its behavior is inconsistent among others devices, even the app may crash on some devices.

Compatibility issues can be small, for example a features not working properly, but they can also be problematic, such as the crash of the app or the system may come up.

Compatibility issues can refer to interoperability between the device and the app. While Android devices are released frequently, it is a challenge for developers to deal with compatibility issues in apps. The developers are unable to choose some device models among thousands of devices to test compatibility issues. According to OpenSignal [26], there are more than 682,000 Android devices, covers 24,093 distinct device models and 1,294 device brands as early as 2015. OpenSignal also referred that the number is more than doubled from the 11, 868 models based on a survey in 2013. Besides, the hardware configuration composition and driver implementation of these devices varies, which bring the heavy workload to the test process.

Meanwhile, the Android OS carrying on the devices varies as well. The OSs are evolving regularly for profits and security needs, with 115 API updating per month on average [25]. Even after releasing a new version, its market could not notably increase.

Table 1 lists the information of the major Android OS versions on April 10, 2020. Note that Android Oreo had been released about 1 year, yet the market share only reached 8.2 %.

2.2 APK Files

Android Package (APK) is a package file which is used in installation and execution of an app on the Android platform. Specifically, Android apps are written in Java by leveraging Android Software Development Kit (SDK). Then, all of app's part is required to be compiled into one package file with a ".apk" extension. If the app is required to be released the app on the application market, the compilation process is necessary. With the APK file, user can manually install the app on the Android device.

For the intellectual property right, the resource code of apps is not available to the public. The primacy approach to analyze the Android app is to transform Java bytecode into intermediate representation for code analysis [3, 19].

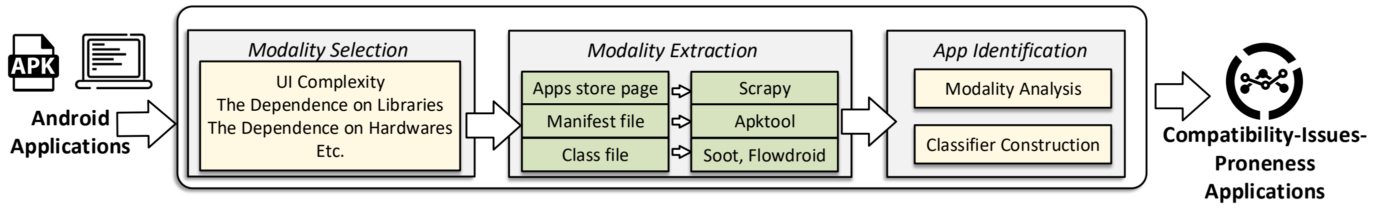

Figure 1: Architecture of our work

3 Empirical Study Methodology

Figure 1 illustrates the workflow of our work. first, we select the potential modalities based on root causes of compatibility issues in apps. Then we extract these modalities from three sources that are store page, manifest file and class file. For the each modality, we measure its usability in app identification, and finally build a classifier to identify the compatibility-issues-proneness apps.

3.1 Modality Selection

First we choose 5 potential modalities and elaborate them as follows:

The category. Indicates a group of apps having appropriately similar characteristics on the application market. It also presents the purpose of this type of app for users to select. For example, the apps, with the Entertainment category tagging, are often used to fill your time and enjoy yourself. Developers might pay varies attention to the compatibility issues among different categories of apps. For example, the developer might be more conscious about compatibility issues for financial apps than other apps. Till now, there are 32 app categories in the Google Play Store.

The size. Indicates the size of code in apps, which includes the size of the APK file and the LOC in the decompiled Java files. The larger app in terms of amounts of code may be harder to be maintained. As a result, large size code in apps may introduce more compatibility issues in apps.

The UI complexity. Indicates the amount of the user interface in apps. Note that activity represents a single-screen UI in apps. A representation of compatibility issues is the inconsistent output on the UIs. Therefore, the app with more complex UIs may have more compatibility issues. Here we use the amount of activity to measure the UI complexity.

The dependence on libraries. Indicates the amount of the dependence of the third-party libraries in apps. The third-party libraries are extensively used in Android to provide functionalities and ease human cost in the development process. However, the dependence on these libraries may bring compatibility issues in apps.

The dependence on hardwares. Indicates the amount of the dependence of hardwares in apps. Some functions in apps rely on hardware composition. For example, A social app have to use the camera to provide features with photos. However, compatibility issues may occur with the problematic implementations of hardware drivers.

3.2 Modality Extraction

In our work, the modalities are derived from three sources as follows.

Google play store page. The description of the app on the store page explicitly presents information (*e.g.*, descriptions, category, install size, user reviews, installs and rates) to users. Therefore, the information can be achieved by indirectly processing the google play store page [24].

Manifest file. The AndroidManifest.xml file is located in the root directory of project source set, which is used to statically define some essential information of the app.

For each component (*e.g.*, activity, service) that the developer creates, and each permission or hardware feature that app requires, must be declared in it [4]. For example, if an app needs to access the camera, CAMERA permission is necessary to be declared in its AndroidManifest.xml. Specifically, the manifest file would have *<uses-permission android:name="android.permission.CAMERA">* in it. Besides, to relieve compatibility issues, the developer can declare the minimum API level required to run the app in AndroidManifest.xml. Note that each line in the manifest file always begins with an element (*e.g.*<activity>, <service>, <uses-permission>), which is used to indicate the kind of the information in the line.

Class files. Some modalities involving code complexity required static code analysis. We first decompile the Android apps from their DEX byte code into intermediate Smali code by Apktool [2]. To capture the dependency of libraries, we use package names in the Smali code to identify third-party libraries. The same process is also used by recent studies [5, 23].

3.3 Dataset

The dataset is built on a server with an Intel Xeon E5-2620V4 2.2G CPU and 128 GB physical memory. We first randomly collect 30,000 apps in Androzoo [1]. Note that these apps were derived from Google Play. We map each app in our dataset with its Google Play store page if exists and leverage the scrapy framework [27] to extract the data. Then, we use Apktool [2] to decompile the APK file and extract AndroidManifest.xml, and leverage FlowDroid [3] to conduct code analysis. Specifically, we remove some subject apps out of 30,000 apps for some reasons below:

- **Unavailable store page.** Some apps maybe withdrawn by developers or pulled off shelves by market maintainers thus failed to be found in Google Play.
- **Low installs.** We filter apps if they have less than 100 installs to make sure the quality of the apps.
- **Limitations of the code analysis tool.** Some apps are heavily obfuscated and unable to be used in our evaluation. Besides, FlowDroid [3] runs out of memory or exceeded the time limit or threw the Soot exception in the process of doing static analysis on some apps.

After the filtering process, we have 7,526 apps in the dataset. We present a summary of these apps with the descriptive statistics in Table 2.

In order to measure the degree of compatibility-issue-proneness in term of each modality, we compute the statistical differences among these apps whose amount of compatibility issues differs. Since no existing the ground truth to distinguish the app has compatibility issues or not (it is also our motivation), we conduct program analysis on these apps and manually label each app is compatibility-issues-proneness or not in term of the compatibility-related APIs it contains. We leverage the statistical result of CRA provided by the tool ICARUS [32] and Pivot [31].

Besides, note that third-party libraries account for a large portion of the code in Android apps, program analysis on Android apps typically requires detecting or removing third-party libraries first. We remove the codes imported by third-party libraries via a library list. More precisely, we first add the package names of the identified libraries into a list and remove such packages according to the list in decompiled apps. The test result shows that about over 95 % apps in our dataset contain the third party libraries such as com/google/ads, com/facebook and com/umeng. By program analysis, we sort each app into three samples in term of the amount of compatibility-related APIs it contains, and then consider the bottom 10% apps as the high-quality apps and top 10% apps as compatibility-issues-proneness apps. Finally, we get 139 high-quality apps and 131 compatibility-issues-proneness apps for the study.

Table 2: Summary of the Apps Used in Our Dataset

Category	Apps(%)	KLOC
Arcade	327 (4.3%)	623-8K
Books and reference	416 (5.5%)	92-18K
Brain	357 (4.7%)	1K-32K
Business	257 (3.4%)	1K-18K
Casual	426 (5.7%)	372-12K
Comics	14 (0.2%)	1K-2K
Communication	610 (8.1%)	297-12K
Education	307 (4.1%)	1K-8K
Entertainment	825 (11.0%)	493-8K
Finance	142 (1.9%)	726-14K
Games	294 (3.9%)	79-146K
Health and fitness	21 (0.3%)	1K-28K
Libraries and demo	124 (1.6%)	182-7K
Lifestyle	214 (2.8%)	2K-7K
Media and video Cards	610 (8.1%)	1K-23K
Medial	12 (0.2%)	2K-8K
Music and audio	119 (1.6%)	1K-11K
News and magazines	163 (2.2%)	615-17K
Personalization	491 (6.5%)	393-8K
Photography	316 (4.2%)	194-87K
Productivity	241 (3.2%)	782-39K
Racing	121 (1.6%)	4K-38K
Shopping	73 (0.1%)	1K-5K
Social	92 (1.2%)	2K-13K
Sports	243 (3.2%)	870-43K
Tools	619 (8.2%)	169-38K
Transportation	10 (0.1%)	3K-13K
Travel and Local	71 (0.9%)	3K-8K
Weather	11 (0.1%)	708-7K
Total	7526	109-146K

4 Study Results

This section presents and discusses the results of our selected modalities in Section 3.1. First, we leverage the statistical methods to measure the association between each modality and compatibility issues in apps. Then we use these modalities to identify the compatibility-issues-proneness apps.

4.1 Evaluation on Modalities

Approach. Here, we measure each selected modalities among two group of samples. Specifically, we first analyze the statistical significance of the difference between the two samples that respectively contain 139 high-quality apps and 131 compatibility-issues-proneness apps by applying non-parametric Mann-Whitney-Wilcoxon (MWW) test [10] at p-value =0.01 [8]. We also used Cliff's Delta statistic that is a nonparametric effect size to measure effect size of the difference between the two groups [12].

We interpret the effect size values as small for 0.147 $<d<0.33$, medium for 0.33 $<d<0.474$, and large for $d>0.474$ with the guidelines in previous work [14,21, 29].

Results. We find that the two groups have statistically significant differences in term of selected modalities except for size and category.

In terms of size and category, we found no significant difference with d of 0.116 and 0.071. For UI complexity, the effect size is medium with d of 0.372.

In terms of dependence of libraries and hardwares, the results show statistical significant difference, with p-value<of 0.0001 and the large effect size (d=0.682, 0.517). We discuss each modality one by one as follows.

UI complexity. Compared with high-quality apps, compatibility-issues-proneness apps tend to have more complexity UI. In fact, a symptom of compatibility issues is the inconsistencies in the UI of Android apps among various devices. Users have direct interaction with the UIs of the apps thus the inconsistent behaviors among UIs are easily noticed. When an app has more UIs to present to the user, it is more difficult to maintain app compatibility. For example, an app may not have specific UI declarations for diverse density screens.

Besides we find the APIs which directly involving what is displayed on the device screen are prone to compatibility issues. Such as the class *android.widget.ZoomButtonsController*, which is used to handle showing and hiding the zoom controls and positioning it relative to an owner view in UI. However, the class was deprecated in API level 26 and may introduce compatibility issues on devices carrying the new OS. If the app contains more UI, the developers may unconsciously use these APIs thus bring compatibility issues.

Dependence on libraries. Compared with high-quality apps, compatibility-issues-proneness apps tend to have more dependence on libraries. Although the usage of libraries eases the development process, the third-party code of the libraries introduce more compatibility issues. For some libraries, the potential compatibility it may introduce may not explicitly indicate in its documentation.

To elaborate our results, we present a code segment in keepass2android [16], a popular Android project with 11,410 stars on GitHub, as shown in Listing 1. In this example, the app leverages a library to create a cipher object for encryption and decryption. Before invoking the library, the developer probes the device model, and verifies whether it is Acer Iconia A500 (Line 2) and records the result into a boolean value. The boolean value is then used in the conditional statement (Line 7) to control the callsite of API *Cipher.getInstance()*. Notably, the developer writes “The Acer Iconia A500 is special and seems to always crash in the native crypto libraries” as an annotation (not showing on the code example for brevity) below this line, which implies that the usage of the API provide by the library may cause the compatibility issue.

```
1  public static boolean deviceBlacklisted() {
2    blacklisted = Build.MODEL.equals(”A500”);
3      return blacklisted;
4  }
5
6  public static Cipher getInstance( ) {
7    if (!deviceBlacklisted())
8      return Cipher.getInstance();
9    }
```

Listing 1: A code segment in keepass2android

As the example shows, the dependence on libraries may cause serious compatibility issue. Therefore the documents of third-party libraries are required to be carefully read. However, most libraries either lack a full documentation or do not indicate its potential compatibility issue in the documents.

Dependence on hardwares. Compared with high-quality apps, compatibility-issues-proneness apps tend to have more dependence on hardwares. The more hardwares the app used, the greater chance for compatibility issues arise. The function of hardwares relies on low-level drivers, whose implementations can make inconsistent behaviors among different devices. Besides, the diversity of hardware composition can easily lead to compatibility issues.

For example, the usage of SD card may introduce the compatibility issues. Some devices (*e.g.*, Samsung Galaxy S2, HTC Evo 4G) do not use the external storage convention /mnt/sdcard/. Besides, there exists other devices with no SD card and multiple SD cards on the market. To deal with the issues, developers have to make extra effort among the various devices. Specifically, the developers may hardcode the SD card path for some targeted devices. However, the issues would still occur since the new devices continue to emerge.

4.2 Evaluation on Identification

Approach. We leverage the 3 modalities to identify the compatibility-issues-proneness apps. We use the labeled apps both as training and test data in a ten-fold cross-validation [18], which is a standard approach for evaluating the approach. Specifically, we partition the apps in 10 subsets, and we use 9 subsets for training the model and 1 for testing. We run this for 10 times, each time we use a different subset for testing. Here we adopt the support vector machine (SVM) as the classifier.

Metrics. We consider two evaluation metrics, the precision and recall. Precision means the fraction of compatibility-issues-proneness apps correctly identified as compatibility-issues-proneness apps among those labeled. Recall means the fraction of compatibility-issues-proneness apps correctly identified as compatibility-issues-proneness apps among those reported by our approach.

Given the ground truth and the detection results, there are four possible outcomes: True positive(TP),

Table 3: Effectiveness of our approach to identify compatibility-issues-proneness apps

Modalities	Recall(%)	Precision(%)
The UI Complexity (UC)	72.1	74.1
The Dependence on Libraries (DL)	78.2	70.1
The Dependence on Hardwares (DH)	75.1	72.2
UC & DH	75.5	76.3
UC & DL	85.2	80.5
DL & DH	82.3	76.4
Total (UC & DL & DH)	84.3	80.4

true negative (TN), false positive (FP) and false negative (FN). TP means that an app is compatibility-issues-proneness with respect to the ground truth and it is identified by our approach. TN means that an app is compatibility-issues-proneness with respect to the ground truth and our approach does not identify it correctly. FP means that an app is not compatibility-issues-proneness with respect to the ground truth but our approach identifies it by mistake. FN means that an app is not compatibility-issues-proneness with respect to the ground truth but our approach truly does not identify it. Finally, the precision and recall are computed by the following formulas:

$$\begin{aligned} Precison &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \end{aligned}$$

Finally, we report the average precision and recall in Table 3.

Results. Results show that our work achieves both higher recall and precision. The precisions and the recalls of the analysis results for apps from different categories are listed in Table 3 (the last two columns). The first column shows the modalities used. The second and the third column list the recall and precision. We also present the experimental results based on each single modality and the combination of different modalities. Note that each modality is useful to improve both the precision and recall of the identification results.

We also analyze the misidentified samples. Specifically, we invite three Android experts to test the apps that are misidentified by our approach. We find some apps that are false positives have compatibility issues in the practical scenario. However, due to the limitation of the compatibility-related APIs dataset, we label them as negative samples. To improve the recall and precision of identification, we will label the app with consideration for test results from Android devices in the future.

5 Threats to Validity

In this section, we present and discusses threats to validity as follows.

Construct validity. Is related to whether our study reflects real-world situations. A possible threat to the validity of our study could be due to the limitation of the dataset. In our study, we have tried our best to make dataset general and representative. Given the fact that the Android platform and app ecosystems are quickly evolving, the investigated apps in dataset over five years, which make sure our experimental results may generalize to most apps.

Internal validity. Is related to uncontrolled aspects that may affect the experimental results. Our results are based on the static analysis that may be subject to issue from analysis tool Apktool and Flowdroid. We may consciously or unconsciously favor the results it presents. Another threat is related to the manual inspection in misidentified samples. We indeed understand that such manual inspection can be error-prone, so this activity has been done with special attention, double-checking and support of the second and third experience developers. We believe that the threat to construct validity is minimal.

External validity. Is related to the possibility to generalize our results. We try to study several apps from different categories. Note that a threat to external validity is that we focus on the free apps in Google Play rather than the paid apps whose APK files are difficult to collect. To be fully conclusive, we will construct our study with paid apps in the future. The apps on platforms other than Android are outside the scope of this paper.

6 Related Work

Some of the existing researches are confined to help developers to find compatibility issue in development test. Lu *et al.* [22] mined large-scale usage data from Wandoujia, and proposed an approach to prioritizing Android device models for individual apps to help developers to identify compatibility issues, based on mining large-scale usage data from Wandoujia. Khalid *et al.* [17] also helped game app developers deal with a similar problem. They picked the devices that have the most impact on app ratings by studying the reviews of game apps. Zhang *et al.* [33] proposed a systematic and cost-effective mobile compatibility test method for selecting mobile devices and their diverse platforms and configurations. Mattia *et al.* [11] automatically identify cross-platform inconsistencies in the UI of Android apps. These proposed schemes have effectively helped developers identify whether compatibility issues are occurring in the test process, but we note that it is challenging for developers to deal with compatibility issue in code-level. Since that existing work discovered that developers are unable to resolve nearly 40% reported crashes [9], which is a possible consequence of the compatibility issue [17, 20, 30, 34].

Besides, some studies are proposed to understand the compatibility issue at code-level in Android apps. Wei *et al.* [30] conducted the first empirical study of compatibility issues caused by Android fragmentation in real-world Android apps at the source code level. Specifically, their work manually studied root causes, symptoms and fixing strategies of compatibility issues in open-source apps. Cai *et al.* [6] conducted a large-scale study of app compatibility issues in Android, concerning the occurrences of these issues at installation time and runtime. Specifically, their work gathered the app trace as well as the system log on the apps' executions and installations, and then analyzed these logs to recognize the execution and installation as a success or failure with related reasons. FicFinder [30] is used to automatically detect compatibility issues in Android apps, but its performance completely relies on the investigation into open-source apps, which requires the labor-intensive process and may lead to a high rate of false negatives. CiD [20] generalizes FicFinder to more compatibility issues, with mining of Android framework versions and modeling the lifecycle of all API methods. PIVOT [31] extracts and prioritizes API-device correlations from a given corpus of Android apps, and consider APIs in such correlations are compatibility issues derived from the device causes. However, they cannot provide deeper insights to help the developers to relieve the human effort in the development process. one can possibly assume that some modalities in apps may jeopardize its compatibility, to the best of our knowledge such relations have not been empirically investigated yet. Our work shed light on the relationship between the compatibility issues and some modalities of Android apps, which is a complement to recent studies.

7 Conclusion

We have contributed to this paper with a novel approach to identify compatibility-issues-proneness apps. We also present some modalities of apps that are related to compatibility issues in Android apps. To this goal, our approach starts with the analysis of given apps and extracts the potential modalities from Google Play and APK files. We then use a statistical approach to measure the association and leverage a classifier to identify the compatibility-issues-proneness apps. The evaluation on a large real-world dataset shows that the accuracy and validity of these modalities.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. The research is supported by the National Key R&D Program of China 2018YFB2100300, 2018YFB0803400, National Natural Science Foundation of China under Grant No.61972369, No.61572453, No.61520106007, No.61572454, and the Fundamental Research Funds for the Central Universities, No. WK2150110009.

References

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: Collecting millions of android apps for the research community," in *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR'16)*, pp. 468–471, 2016.

[2] Apktool, *A Tool for Reverse Engineering Android apk Files*, July 6, 2021. (`https://ibotpeaches.github.io/Apktool/`)

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, pp. 259–269. ACM New York, NY, USA, 2014.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the android permission specification," in *Proceedings of ACM Conference on Computer and Communications Security*, pp. 217–228, 2012.

[5] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, pp. 356–367, 2016.

[6] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 216–227, 2019.

[7] J. Callaham ,*Android Now Running on Over 2.5 Billion Active Hardware Devices*, 2020. (`https://www.androidauthority.com/android-2-5-billion-devices-983534/`)

[8] W. J. Conover, *Practical Nonparametric Statistics*, vol. 350, 1998. (`http://140.117.153.69/ctdr/files/857_1734.pdf`)

[9] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*, pp. 408–419, 2018.

[10] M. P. Fay and M. A. Proschan, *Wilcoxon-Mann-Whitney or T-test? On Assumptions for Hypothesis Tests and Multiple Interpretations of Decision Rules*, vol. 4, pp. 1. , 2010.

[11] M. Fazzini and A. Orso, "*Automated Cross-Platform Inconsistency Detection for Mobile Apps*," vol. 4, pp. 308–318, 2017.

[12] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*, 2005. (`https://psycnet.apa.org/record/2005-04135-000`)

[13] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 167–177, 2018.

[14] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 278–289, 2016.

[15] H. Huang, L. Wei, Y. Liu, and S. C. Cheung, "Understanding and detecting callback compatibility issues for android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 532–542, 2018.

[16] keepass2android, *Password Manager App for Android*, July 6, 2021. (`https://github.com/PhilippC/keepass2android/`)

[17] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of android game apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 610–620, 2014.

[18] R. Kohavi, *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, vol. 14, pp. 1137–1145, 1995.

[19] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS'11)*, vol. 15, pp. 35, 2011.

[20] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "CiD: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 153–163, 2018.

[21] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: A threat to the success of android apps," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pp. 477–487, 2013.

[22] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, and F. Feng, "Prada: Prioritizing android devices for apps by mining large-scale usage data," in *IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*, pp. 3–13, 2016.

[23] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 653–656, 2016.

[24] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis for app releases in google play," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 435–446, 2016.

[25] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *IEEE International Conference on Software Maintenance (ICSM'13)*, pp. 70–79, 2013.

[26] Opensignal, *Android Fragmentation Visualized*, Aug. 2015. (`https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf`)

[27] Scrapy, *An Open Source and Collaborative Framework for Extracting the Data*, July 6, 2021. (`https://scrapy.org/`)

[28] Statcounter, *Mobile Operating System Market Share Worldwide*, 2020. (`https://gs.statcounter.com/os-market-share/mobile/worldwide/`)

[29] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open-source software projects," in *IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*, pp. 256–266, 2018.

[30] L. Wei, Y. Liu, and S. C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 226–237, 2016.

[31] L. Wei, Y. Liu, and S. C. Cheung, "Pivot: Learning API-device correlations to facilitate android compatibility issue detection," in *IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*, pp. 878–888, 2019.

[32] C. Xu, Y. Xiong, W. Huang, Z. Meng, F. Miao, C. Su, and G. Mo, "Identifying compatibility-related apis by exploring biased distribution in android apps," in *IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings*, pp. 280–281, 2020.

[33] T. Zhang, J. Gao, J. Cheng, and T. Uehara, "Compatibility Testing Service for mobile applications," in *IEEE Symposium on Service-Oriented System Engineering*, pp. 179–186, 2015.

[34] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *IEEE Symposium on Security and Privacy*, pp. 409–423, 2014.

Biography

Chen Xu received the B.S. degree in computer science from Anhui Agricultural University in 2014. He is currently working towards the Ph.D. degree at the Department of Computer Science and Technology, University of Science and Technology of China. His current

research interests include information security and software engineering. (Email: kakyo82@mail.ustc.edu.cn)

Caimei Wang received the Ph.D degree in computer science from University of Science and Technology of China in 2018. She is an associate professor in School of Artificial Intelligence and Big Data, Hefei University. Her main research interests include computer network, trusted computing and information security. (Email: wangcmo@mail.ustc.edu.cn)

Wenchao Huang received B.S. and Ph.D. degrees from University of Science and Technology of China in 2006 and 2011 respectively. He is an associate professor currently with Department of Computer Science and Technology, University of Science and Technology of China. His current research interests include mobile computing, information security, trusted computing, and formal methods. (Email: huangwc@ustc.edu.cn)

Yan Xiong received the B.S., M.S., and Ph.D degrees from University of Science and Technology of China in 1983, 1986 and 1990 respectively. He is a professor in School of Computer Science and Technology, University of Science and Technology of China. His main research interests include distributed processing, mobile computing, computer network and information security. (Email: yxiong@ustc.edu.cn)

Zhaoyi Meng received the B.S. degree in information security from University of Electronic Science and Technology of China in 2014, and the Ph.D. degree in computer science and technology from University of Science and Technology of China. He is currently a Post-Doctoral Researcher with the Department of Computer Science and Technology, University of Science and Technology of China. His current research interests include Android security and software formal verification. (Email:mzy516@ustc.edu.cn)

Fuyou Miao received his Ph.D of computer science from University of Science and Technology of China in 2005. He is an associate professor in the School of Computer Science and Technology, University of Science and Technology of China. His research interests include information security, information coding key management in WSN, and network security. (Email:mfy@ustc.edu.cn)