A Large-Scale Study of Application Incompatibilities in Android

Haipeng Cai
Washington State University, Pullman, USA
haipeng.cai@wsu.edu

Ziyi Zhang
Washington State University, Pullman, USA
ziyi.zhang2@wsu.edu

Li Li
Monash University, Australia
li.li@monash.edu

Xiaoqin Fu
Washington State University, Pullman, USA
xiaoqin.fu@wsu.edu

ABSTRACT

The rapid expansion of the Android ecosystem is accompanied by continuing diversification of platforms and devices, resulting in increasing incompatibility issues which damage user experiences and impede app development productivity. In this paper, we conducted a large-scale, longitudinal study of compatibility issues in 62,894 benign apps developed in the past eight years, to understand the symptoms and causes of these issues. We further investigated the incompatibilities that are actually exercised at runtime through the system logs and execution traces of 15,045 apps. Our study revealed that, among others, (1) compatibility issues were prevalent and persistent at both installation and run time, with greater prevalence of run-time incompatibilities, (2) there were no certain Android versions that consistently saw more or less app incompatibilities than others, (3) installation-time incompatibilities were strongly correlated with the minSdkVersion specified in apps, while run-time incompatibilities were most significantly correlated with the underlying platform's API level, and (4) installation-time incompatibilities were mostly due to apps' use of architecture-incompatible native libraries, while run-time incompatibilities were mostly due to API changes during SDK evolution. We offered further insights into app incompatibilities, as well as recommendations on dealing with the issues for bother developers and end users of Android apps.

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software.*

KEYWORDS

Android, compatibility, installation failure, run-time failure

ACM Reference Format:
Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-Scale Study of Application Incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330564

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330564

1 INTRODUCTION

Due to the shift of personal computing to mobile platforms and the predominance of Android [38], to most people today software applications are mostly Android apps. Like software in other application domains, mobile apps are subject to common issues with various quality factors, including not only those related to reliability and security [3, 8, 42] which have received wide attention [4, 5, 12], but also compatibility issues which have not been as much attended. In particular, incompatibilities lead to low usability hence poor user experiences, harming the health of the mobile software ecosystem. Since they compromise the production and adoption of apps, compatibility issues also impede the productivity of app developers. Intuitively, the larger the user base of the ecosystem, the greater the (negative) impact of these issues.

As Android increasingly gains its momentum, compatibility issues in Android also have been on the rise [30, 39, 43, 47, 48]. The open-source nature of Android has led to the diversification of Android devices [34] and customized Android platforms (e.g., the Android operating system kernel) [11]. The Android system itself also constantly evolves, leading to continuous SDK/API changes [35, 44]. While this facilitates the growth of Android on mobile computing markets, it also has caused various incompatibilities in Android apps. In consequence, apps developed for mobile devices of one model and/or the Android system of a particular version may not normally function, or cannot even be installed to, devices of a different model or other versions of the Android system. As people increasingly rely on Android apps for their daily lives, it is crucial for app developers and end users to understand application incompatibilities in Android, a first step towards mitigating and even preventing relevant issues.

A few prior studies concerned compatibility issues in Android apps, but considered only those due to a particular kind of causes (e.g., fragmentation [27, 29, 44] and API evolution [33, 35]). Also, these studies examined a relatively small set of app samples, without considering the time factor of the samples or the issues. Latest efforts [28, 32] focus more on detecting/predicting *potential* and *specific* compatibility issues induced by API changes through static code analysis (thus suffer false positives), and/or characterizing how developers deal with compatibility issues in app's code [28]. Other relevant research investigated app bugs and crashes in general [10, 36], not necessarily due to compatibility issues.

As it stands, there has been no study that addresses (1) the status quo of app incompatibilities in Android that are ***actually observed*** both at a large scale and with an evolutionary perspective and (2) varied symptoms of these issues occurring at

different stages (i.e., installation and execution) of app use and corresponding root causes. Such a study would shed light on a comprehensive understanding of the compatibility issues in Android apps and how they evolve, so as to offer insights on mitigating and preventive strategies against those issues.

To fill this gap, in this paper, we conducted *a comprehensive and longitudinal study at a large scale of* **observed** *incompatibilities in Android apps*. We collected 62,894 benign apps from various sources that were developed in eight different years (2010 through 2017), and profiled 15,045 of these apps with random inputs each for five minutes. We then investigated both *installation-time* and *run-time* incompatibilities according to the corresponding APKs and their execution traces, respectively, on all the 8 major Android versions of a non-trivial market share (from API level 19 through 27) [16]. We regarded an app as *installation-time incompatible* if it can be successfully installed (as indicated by the return code/message of the installation) to at least one (version of) Android platform but cannot to another. Similarly, an app was regarded as *run-time incompatible* if it can run for a specified time period successfully (i.e., without producing any system error messages, exceptions, or crashes) on at least one (version of) Android platform but cannot on another. We differentiate these two classes of app incompatibilities to enable an in-depth understanding of the extent and phases of the corresponding compatibility issues.

Through these datasets and the two complementary experiments (on the two types of compatibility issues, respectively), we aim to assess the prevalence of both types of compatibility issues, as well as their distribution with respect to various symptoms and causes. We also intend to disclose key properties of apps (e.g., age) and those of the Android platform (e.g., release time) that have a significant impact on app incompatibilities. Finally, in a longitudinal view, we attempt to reveal the evolutionary patterns of these characteristics of app incompatibilities over time.

In particular, we explore the following research questions:

- **RQ1**: *How prevalent are app compatibility issues in Android?* Previous relevant works have suggested the *potential* existence of compatibility issues in Android apps due to *particular* reasons (e.g., SDK evolution) [28, 32, 35, 44]. To understand compatibility issues as exhibited due to *any* possible reasons, we actually installed and ran each sample app to characterize such issues in them as observed, and differentiated the different stages (installation and run time) in which such issues occur.
- **RQ2**: *How are the compatibility issues in Android apps distributed over major symptoms?* App incompatibilities are exhibited through observable symptoms (e.g., error logs upon installation failures, and crash traces upon execution failures, that are induced by compatibility issues) [10, 36]. Examining what contributed to incompatibilities with respect to particular symptoms can help us understand the root causes of those symptoms. Thus, we examined the main symptoms (effects) of installation- and run-time incompatibilities and looked into the distribution of installation/execution failures over those symptoms.
- **RQ3**: *What are the main factors that contributed to the incompatibilities in Android apps?* Current understanding about Android app incompatibilities attributes them mainly to the issues with the Android platform (e.g., fragmentation [44] and SDK evolution [28]). To complement this understanding, we further study possible properties of apps that are related to the compatibility issues. We conducted statistical analyses to discover possible correlations between installation/execution failures and relevant app properties, including the age of apps and their specification of minimal SDK version.

Guided by these questions, our study revealed, among others:

(1) Substantial portions (15% on average) of our benchmarks from varied years suffered incompatibilities that failed their installation to one or more of the eight Android platforms we studied. These issues were strongly correlated to the *minSdkVersion* specified in apps and its distance from the API level of the platform the app was attempted to install to.
(2) 90% of the installation-time incompatibilities were due to the apps using native library functionalities that are not supported by the underlying hardware architecture. Among other cases, the issues were mainly attributed to vendor customizations of the Android system.
(3) Run-time compatibility issues were even more prevalent (30–50%) than the issues at installation time in our benchmarks, on Android versions prior to API 24. Opposite to apps' *minSdkVersion* being the major contributor to installation-time issues, primarily contributed to run-time incompatibilities was the Android platform's API level. Android versions since API 24, however, did not see any run-time compatibility issues.
(4) On older Android platforms (with API levels prior to 24), run-time compatibility issues were exhibited mostly via verify errors (50%) and native crashes (20%), both mainly caused by API changes during Android SDK evolution. Counter-intuitively, apps developed in years closer to a platform's release year tended to be more likely to be run-time incompatible on that platform.

From these observations, we also distill lessons learned and make recommendations for developers and app users to deal with app incompatibilities based on the lessons. We have released all of the source code and datasets used in our study, found here.

2 BACKGROUND

In this section, we give brief descriptions of the basic concepts and terms about the Android system and Android apps that are necessary for readers to understand the remainder of this paper.

2.1 Android Platform and SDK

The middle layer between the Android operating system (OS) (a customized Linux kernel) and its user applications constitutes the Android framework. This framework provides the implementation of application programming interface (API) methods through which user apps can receive system services and invoke common functionalities associated with mobile devices. The API is typically part of the Android software development kit (i.e., *SDK*) which also includes tools to support app development.

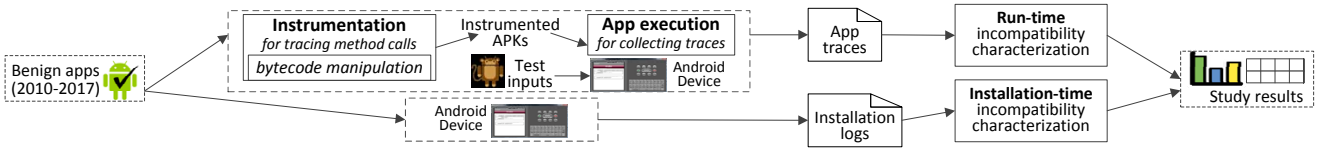

Figure 1: Overview of the process flow of our app incompatibility study.

Under the framework-based development paradigm, Android apps usually comprise building blocks called *components* of four types: *Activity* forming the basis of user interface, *Service* performing background tasks, *Broadcast Receiver* (or just *Receiver* in short) responding to system-wide broadcasts, and *Content Provider* offering database capabilities.

2.2 Incompatibilities in the Android Ecosystem

As it stands, the same version of Android system may run on devices with different hardware configurations (e.g., varied processor models and screen sizes, with/without certain kinds of sensors). Meanwhile, different vendors may customize the Android system in various ways to help promote their mobile device products. These variations constitute a phenomena in Android, called *fragmentation* [27]. Due to the fragmentation problem, it is difficult to develop an app that functions normally on any device with any Android version, resulting in fragmentation-induced compatibility issues [44].

To accommodate its marketing demands, Android provides a large variety of operating system and SDK versions that both evolve constantly, with faster evolution seen by the SDK (especially the API) [16]. For instance, during the past eight years, Android has released over 20 API versions (each corresponding to an *API level*). Due to the API evolution, an app developed with one API level might not be installable or runnable with Android of different API levels. Thus, API evolution constitutes another important cause of incompatibilities in Android apps [35].

There are two types of compatibility in Android: *device compatibility* and *app compatibility* [14]. Device compatibility concerns whether a mobile device is Android compatible (i.e., able to run apps written for the Android runtime). Only Android-compatible devices include Google Play Store. Also, the regular avenue for users to install apps is from Google Play Store. Thus, device compatibility is often not an issue for app developers. App compatibility is a primary concern of app developers, which is thus the focus of our study. An app can have incompatibilities with the device configurations, the Android system (mainly the framework/API), or both.

2.3 Compatibility Attempts in Android Apps

When an app is built, the developer can choose which API level the app targets and the minimum API level required for the app to function [48]. These numbers on API levels may be recorded in the manifest file of the app package (i.e., APK), as *minSdkVersion* (i.e., the minimum API level) and *targetSdkVersion* (i.e., the targeted API level). It is specified in the official Android developer guide (OADG) [17] that an app should always declare *minSdkVersion* (or it will be defaulted to 1) [25]. Since API level 4, apps can also declare *targetSdkVersion*, which is optional though—when unspecified, it would be defaulted to *minSdkVersion*. It is allowed (also since API level 4) yet not recommended to declare another API level number in the manifest: *maxSdkVersion*, which gives the maximal SDK version that the app can run with. According to the OADG, an Android system does not allow an app to be installed if the API level used by the system is lower than the app's *minSdkVersion* or higher than its *maxSdkVersion*.[1] However, Android promises *backward compatibility* [22]: an Android platform with an API level higher than an app's *minSdkVersion* allows the app to be installed and function as expected; if the platform API level is higher than the app's *targetSdkVersion*, the system also enables compatibility behaviors allowing the app to work as expected [25]. Nevertheless, between API level 4 and API level 6, the check with respect to *maxSdkVersion* is enforced: the installation will fail if the app specifies a *maxSdkVersion* that is smaller than the platform's API level.

3 METHODOLOGY

This section gives an overview of our study process, and describes the dataset and tools used. We then define main metrics and measures used for quantifying incompatibilities hence answering our research questions.

3.1 Process Overview

Figure 1 depicts the process of our study. We used the APKs of benign apps as benchmarks. To enable an evolutionary viewpoint in examining app incompatibilities, our datasets included samples developed in different years (2010–2017). We considered all the 8 Android versions (API level 19 through 27), except for API 20 which is dedicated for wearable devices. These versions combined constitute 96.5% of the entire Android market share (by late 2018 [16]). We dismissed other versions concerning their old age (released in 2012 or earlier) and tiny market share (1.5% or lower).

We concern both installation- and run-time incompatibilities of apps. To characterize the former, we attempted to install the original APK of each app to an Android device for each of studied Android versions. We collected the installation logs, and then analyzed these logs to recognize the installation as a success or failure. The logs also give information for us to understand the effects of installation failures. We treated uninstallation success as part of installation success. Thus, for each app installation test, we uninstalled the app after successfully installing it.

To characterize run-time incompatibilities, we need to examine app executions. We performed lightweight instrumentation and profiling to facilitate differentiating two classes of run-time incompatibilities: (1) *incompatible launch*, with which an app cannot even be launched successfully, and (2) *incompatible running*, with which a launch-compatible app exhibits incompatibilities after successfully running for a while. To that end, we first

[1]The check against *maxSdkVersion* has been abolished by Android since its version 2.0.1 [25] (which corresponds to API level 6 [21]).

Table 1: Subject apps used in our study

Data use	number of samples from each year within 2010-2017								Total
	2010	2011	2012	2013	2014	2015	2016	2017	
Installation-time incompatibility study	16,835	9,977	10,991	9,688	5,300	5,406	2,431	2,266	***62,894***
Run-time incompatibility study	1,531	2,020	2,054	1,750	1,335	3,127	1,548	1,680	***15,045***

instrumented each app for tracing (all) method calls, with the support of an underlying framework Soot [31] for Dalvik bytecode manipulation. We then ran the instrumented app for five minutes on an Android device, for each of the 8 Android versions, to gather the app trace as well as the system log on the app's execution. In this way, differentiating the two run-time incompatibility situations was enabled by simply checking the trace: if the trace contains valid records of calls, we can exclude the app from the *incompatible launch* category. We identified the effects of run-time incompatibilities by further checking the system log.

The outputs of our study pipeline are the characterization results on both types of incompatibilities. Next, we elaborated on key elements of our study design.

3.2 Subject Apps

A summary of subjects (62,894 benign apps) used in our study is listed in Table 1. These apps were developed in different years from 2010 through 2017. The 2,266 apps of 2017 were downloaded directly from Google Play [24]. All other benign apps were obtained from AndroZoo [2], a diverse collection of apps from various sources. All samples were confirmed as benign via VirusTotal [1].

The year of each app was obtained according to the dex date retrieved from the app's APK, which is the last modification date of the app bytecode (as stored in `classes.dex`). During our data collection, we discarded corrupted APKs which either cannot be unzipped or are missing resource files. These corrupted apps are not installable, but they are not relevant to app incompatibilities. Eventually, we used 62,894 apps for the installation-time compatibility study. For our study, we needed the *minSdkVersion* for each app. Thus, we defaulted it as 1 (as Android does [25]) for apps that did not specify *minSdkVersion*.[2]

We limited the number of samples to be used for our run-time compatibility study concerning the overheads (i.e., of executing each app for five minutes for each of the 8 Android versions). We started with the apps used in the installation-time compatibility study from each year. After finding out that many of the apps cannot be successfully installed (to one or more of the 8 Android versions), we continued to select more apps from respective sources, until we had at least 1,000 installable apps for each year. In total, we used 15,045 apps for the run-time compatibility study.

3.3 Experimental Setup and Procedure

For the scalability of our study pipeline and control of study overheads, we used 8 Android virtual devices (AVDs), all Nexus One with 2G RAM and 1G SD but with varied API levels (i.e., 19, 21, 22, 23, 24, 25, 26, and 27). We ran these AVDs via the Android emulator [18] shipped with each corresponding Android version.

To trace apps for studying run-time incompatibilities, we need to feed the apps with run-time inputs. We used the *Monkey* tool [20] shipped with the Android SDK to generate random inputs for app exercising. Both the app call traces and system logs were serialized using the Logcat tool [19], also part of the SDK. For app instrumentation and method-call tracing, we used our Android characterization toolkit [7] and dynamic analysis utilities [6]. We used *apktool* [45] to retrieve the manifest data of an app, including the *minSdkVersion*, *targetSdkVersion*, and *maxSdkVersion* of the app. We utilized the *adb* tool [15] for app installation and uninstallation.

With these study facilities, we now define key notions/terms we referred to earlier but only intuitively. An app installation (uninstallation) to a device A is regarded as successful only if running the `install` (`uninstall`) command of *adb* on the app to A returns a code explicitly indicating the success. Then, an app is *installation-time compatible* with A if the app can be installed to and then uninstalled from A, both successfully; otherwise, the app is *installation-time incompatible* with A. For a given app, failure in installing the app to a device could be due to reasons other than (installation-time) compatibility issues. We regarded the failure as caused by compatibility issues with one device if the app can be successfully installed to at least one different device (i.e., with a different Android version in our study). In addition, Android does not allow apps with *minSdkVersion* higher than the underlying platform's API level to be installed to that platform [25]. Thus, we did *not* treat installation failures of such apps on those platforms as induced by (installation-time) compatibility issues. With Android versions of API level 4, 5, and 6, app with *maxSdkVersion* lower than the underlying platform's API level is not allowed to install either. However, since our study only experimented with Android versions of API level 19 or higher, the check policies against *maxSdkVersion* did not affect our results.[3]

Similarly, for a given app, failure in executing the app on a device A, as indicated by execution error messages (in the system log for the app's execution), exceptions, and crashes, could be ascribed to reasons other than (run-time) compatibility issues (e.g., bugs in the app or invalid user operations). To exclude failures induced by non-compatibility issues with one device, we ran the app on various devices of different Android versions (all for five minutes with Monkey inputs). If the app can run on at least one different device without exhibiting any of the execution failure symptoms, yet it exhibited failures during the execution on A (also for five minutes and with the same sequence of Monkey inputs as used in the successful run), then we regarded the app as run-time incompatible with A. Then, accordingly, the error messages, exceptions, and crashes exhibited in the failing execution on A were regarded as run-time incompatibility symptoms.

Thus, one challenge to our study was to exclude irrelevant (non-compatibility induced) failures during app installation and executions, as it was time-consuming to find the successful

[2] 0.32–1.12% of the apps in our yearly datasets did not specify *minSdkVersion*.

[3] Our initial datasets of year 2010 to 2017 had 1.06%, 0.83%, 0.98%, 0.52%, 1.03%, 2.89%, 3.51%, and 6.1% of the apps with *maxSdkVersion* specified, and 0, 1, 0, 0, 0, 3, 1, 2 apps with *minSdkVersion* higher than 19, respectively.

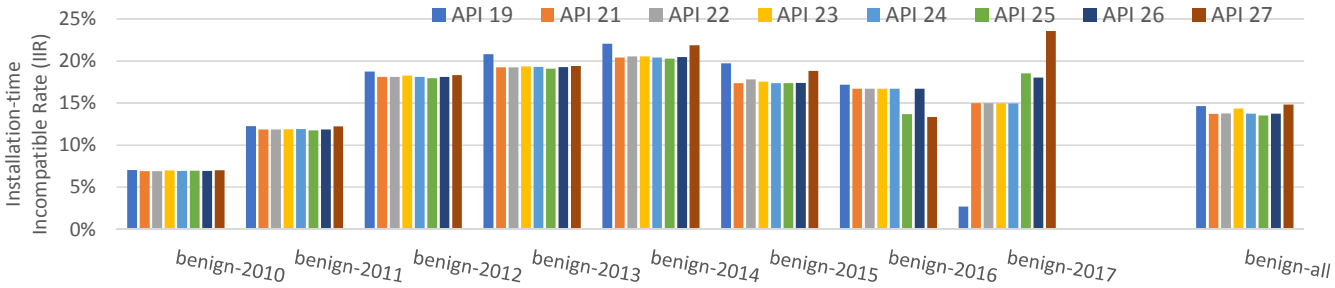

Figure 2: Installation incompatibilities in terms of IIR (y axis) in benign apps of varied years (x axis).

installation/execution. The cost was particularly high for finding the successful execution, because of the great overhead of the dynamic analysis (e.g., exercising each app for five minutes, and checking the traces and system logs for failure symptoms). Fortunately, we installed and ran each sample on the 8 different Android versions, so we excluded irrelevant failures for each version by referring to the results with all other 7 versions. We would only have needed to try additional devices if the installation/execution of an app failed on all the 8 versions, which fortunately did not happen for any samples in our study.

3.4 Measurements

To answer our research questions, we computed a set of measures for the two classes of app incompatibilities. For installation-time incompatibilities, we measured the *installation-time incompatible rate* (IIR) for each app set (apps of a particular year) against an API level as the percentage of apps that cannot be successfully installed to the AVD with that API level. We also computed the distribution of IIR over major installation-time incompatibility effects. For run-time incompatibilities, we computed *run-time incompatible rate* (RIR), but separately for the two subclasses: incompatible launch and incompatible running. We then looked into the distribution of RIR over major run-time incompatibility effects.

In addition, we conducted a series of statistical analysis to discover correlations between app incompatibilities and several properties of the Android platform itself and its apps. Specifically, in this study we were concerned about compatibility-related platform properties (*release year* and *API level*) as studied by prior works [28, 32]; more importantly, we also looked at multiple app properties that are potentially relevant to app incompatibilities: *minSdkVersion*, *app (creation) year*, *app lapse*, and *API lapse*[4]. We retrieved the SDK release years per the Android history [21].

We introduce and study the two derivative properties (app lapse, API lapse) because intuitively they can be used to examine the length of Android's forward and backward compatibility with apps [32]. To measure the correlations of interest, we computed the Spearman's correlation coefficients [37] for relevant variables. We chose this method because it is a non-parametric correlation statistics that makes no normality assumption about underlying data points. We refer to [46] in discerning correlation strengths based on the range of the coefficient's absolute value: $[0.0, 0.19]$: very weak, $[0.2, 0.39]$: weak, $[0.4, 0.59]$: moderate, $[0.6, 0.79]$: strong, and $[0.8, 1.0]$: very strong.

[4]In particular, we define app lapse as (SDK release year − app year), and API lapse as (platform's SDK API level − app minSdkVersion).

Next, we present the results of our empirical studies on installation-time and run-time incompatibilities in Android apps. We investigate the prevalence, contributing factors, distribution, and security relevance of these incompatibilities, all with an evolutionary perspective. For better clarity, we focus on installation- and run-time incompatibilities in two separate studies, Study I (Section 4) and Study II (Section 5), respectively. We address the three research questions in both studies and discuss major findings around these questions.

4 STUDY I: INSTALLATION-TIME INCOMPATIBILITIES

As we mentioned earlier, an app is regarded as installation-time compatible if its installation and uninstallation are both successful. Intuitively, if an app is installable to a device, it should be uninstallable from the same device as well. Our study results confirmed this assumption: all of our subject apps that were installed successfully were all uninstalled successfully too.

4.1 RQ1: Prevalence of App Incompatibilities

Figure 2 delineates the overall IIR of benign apps from each of the eight years, and that of all benign apps considered in this study as a whole, despite the varied symptoms of installation failures. A high-level observation is that, within each yearly dataset, the failure rate was quite close across the eight Android versions, indicating largely negligible impact of the platform's API level. In fact, the aggregate result for all the benign apps put together as one amalgamated dataset (*benign-all*) confirmed the same observation. On the other hand, across the eight years, IIR of the benign apps experienced a steady growth until 2014, followed by a slowly decreasing trend. This evolution pattern potentially reflects the gradual (albeit still not complete) compatibility adaptation between apps and the rapidly evolving platform in the Android ecosystem—initially the IIR grew as API lapse increased (as indicated by the strong positive correlation between IIR and API lapse as we found; see Table 2), and then the ecosystem took a few (4, as per Figure 4) years to iron out the compatibility issues. Concerning quantitative measures, the numbers show the IIR ranging from 7% to almost 23% for individual app years and Android versions, and that the aggregate IIR was about 15%. There is no clear/consistent association between dataset sizes and IIR, suggesting no substantial impact of the large size variations of our datasets on these general observations.

A look into the results for the benign-2017 dataset revealed an outlier with the newest version studied (API 27), with which the IIR

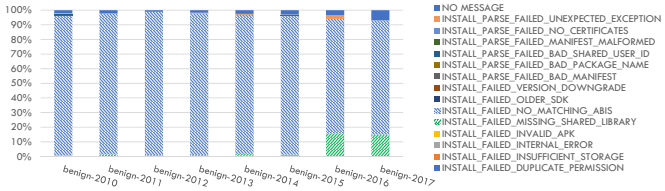

Figure 3: Percentage distribution of installation-time incompatible apps over the varied symptoms exhibited.

of this dataset was higher than any other datasets on any Android versions. API 27 (Android 8.1) was released at the end of 2017 [21], when almost all of the studied benign-2017 apps had already been created. Thus, Android 8.1 had the worst backward compatibility with the most recent prior apps. Another outlier of the general trend, API 25 and API 26 saw IIR rising again. An implication of these contrasts is that newer/older Android versions did not necessarily had better/worse compatibility with recent past/future apps.

> **Finding 1:** *Over the eight years studied, installation-time app incompatibilities persisted with varying (7% to 23% IIR) yet substantial presence (15% IIR on overall average), largely independent of the underlying Android versions used.*

4.2 RQ2: Distribution of App Incompatibility Symptoms

Figure 3 depicts the percentage distribution of apps with installation failures due to compatibility issues over the 15 failure effects (shown in the legend) we observed in this study. Of these effects, 14 were error codes (e.g., `INSTALL_FAILED_INVALID_APK`) produced by *adb* as we observed at installation time. The last one (`NO_MESSAGE`) was what we used to indicate the situation in which the installation simply failed without resulting in any error message/code. While we also computed such a distribution for each Android version separately, given the very-high similarity of these distributions across those versions, here we only present and discuss the aggregate distribution: installation failure effects were amalgamated over all the versions for each of the eight app years.

Despite this great variety of symptoms and the noticeable percentage (5%) of `NO_MESSAGE` cases, the top two dominating symptoms were `INSTALL_FAILED_NO_MATCHING_ABIS` and `INSTALL_FAILED_MISSING_SHARED_LIBRARY`. As highlighted via pattern fill (opposed to solid fill for the minor symptoms), for most app years, over 90% of the apps that failed to be installed encountered the `INSTALL_FAILED_NO_MATCHING_ABIS` error[5], meaning these apps use native libraries that are not compatible with the hardware (CPU) architecture. For the last two app years (2016 and 2017), a substantial percentage (15%) of apps failed at installation because they use a library that does not exist in the underlying Android framework (mostly due to the removal of those libraries during vendor customizations), as indicated by the `INSTALL_FAILED_MISSING_SHARED_LIBRARY` error.

[5]As per our approach to excluding non-compatibility-induced installation failures (§ 3.3), we validated that these apps ran normally (did not crash nor encounter such errors) on at least one different device, a core part of our experimental methodology.

Table 2: Spearman correlation (moderate or stronger coefficients in boldface) between IIR and contributing factors

	app lapse	API lapse	minSdkVersion	SDK API level	app year
overall	0.046	**0.680**	**-0.709**	-0.047	-0.061
ABI	-0.033	**0.651**	**-0.695**	-0.133	-0.015
LIB	-0.081	**0.418**	**-0.437**	-0.056	0.078

> **Finding 2:** *Installation failures in apps were predominantly (over 90%) due to their reliance on architecture-incompatible native libraries, and others also (up to 15%) due to their use of libraries missing in vendor-customized Android frameworks.*

4.3 RQ3: Contributing Factors of App Incompatibilities

To further understand the causes of installation-time incompatibilities hence distill insights for dealing those issues, we examined the distribution of IIRs over app lapse and API lapse. This distribution is depicted by the heatmap of Figure 4, including the overall distribution (regardless of symptoms) and the distribution for the top two dominating symptoms separately. Each data point (square) in the plot represents the IIR (with the square's color) of all the benign apps with a specific app lapse and a specific API lapse, regardless of their creation years.

The overall distribution (left) suggest that the IIRs were very low with apps of an API lapse in $[-10, 10]$, and high mainly with apps of high API lapse (>10), especially with those of >15 API lapse. App lapses, on the other hand, were of no considerable impact on IIRs: apps with the same API lapses have very close IIRs despite their varying app lapses. This general observation still held in the separate distributions associated with the two dominating symptoms: out of the $[-10, 10]$ range, higher API lapses were associated with high IIRs, and app lapses had no major impact. Comparing the IIR distribution between the symptoms reveals that apps with very high IIRs (and associated very large API lapses) failed more often due to vendor customizations of the Android framework (missing libraries) than device hardware incompatibilities (architecture-incompatible native libraries).

As shown in Table 2, our Spearman correlation coefficients between IIR and various possible contributing factors (including API lapse, and the three individual app/platform properties mentioned in Section 3.4) confirmed the strong correlation (0.68) of IIR with API lapse, rather than with the app years. This level of correlation strength was observed similarly for the top two symptoms. The strength was greater for apps failing in installation due to `INSTALL_FAILED_NO_MATCHING_ABIS` (*ABI*) than those due to `INSTALL_FAILED_MISSING_SHARED_LIBRARY` (*LIB*). This is because the former has much greater dominance in the overall distribution of IIRs (see Figure 3). Apps' *minSdkVersion* was similar to their API lapses in terms of the correlation with IIR, indicating that *minSdkVersion* was the main contributor to the correlation strength that *API lapse* had with IIR. The fact that, like *app lapse*, *app year* and *SDK API level*, had no/negligible correlation with IIR implies that the impact of app year on IIR seen in Figure 2 can be further explained by the underlying varying minSdkVersions specified in apps developed in varying years.

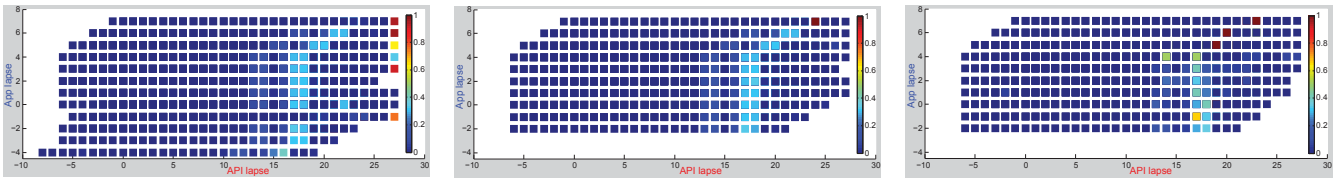

Figure 4: Distribution of IIR (encoded by square color) over app lapse and API lapse, for all symptoms (leftmost) and the two dominating ones: `INSTALL_FAILED_NO_MATCHING_ABIS` **(middle) and** `INSTALL_FAILED_MISSING_SHARED_LIBRARY` **(right).**

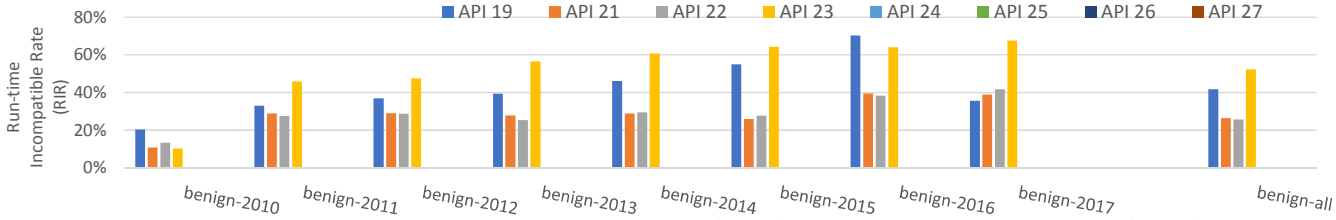

Figure 5: Run-time incompatibilities in terms of RIR (y axis) in benign apps of varied years (x axis).

Finding 3: *Apps of greater API lapses (especially out of* $[-10, 10]$*), mainly due to their minSdkVersions being further away from the underlying platform's API level, had higher IIRs. Vendor customization of Android contributed more than hardware incompatibilities to very-high IIRs.*

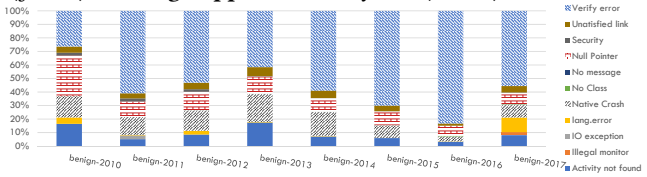

Figure 6: Percentage distribution of run-time incompatible apps over the varied symptoms exhibited.

5 STUDY II: RUN-TIME INCOMPATIBILITIES

We present our empirical results on run-time app incompatibilities following the same structure of the first study. We originally intended to study two categories of run-time incompatibilities: incompatible launch and incompatible running. However, our results revealed that in our datasets there were no apps found to be associated with the incompatible launch issues. Thus, in the rest of this section, we only look at the first category of run-time incompatibilities—incompatible running.

5.1 RQ1: Prevalence of App Incompatibilities

In the same format as Figure 2, Figure 5 illustrates the overall RIR of our benchmarks per yearly dataset and that of all benchmarks amalgamated as a whole (*benign-all*). Notably, none of the four newer Android versions (API levels 24, 25, 26, and 27) saw any run-time incompatibilities with respect to the apps used in this study. This largely facilitated our confirmation of execution failures as indeed induced by compatibility issues. More importantly, the sudden disappearance of run-time incompatibilities since API 24 indicates the substantial improvement of Android in accommodating apps of various ages since that version, implying the changes made in that version [13] largely addressed forward and backward compatibility issues. In fact, API 24 was the most substantially changed version during the Android history we studied with respect to prior versions [28].

Among the four older versions, API 19 and API 23 had mostly considerably higher (by 15% or above) RIRs than the other two versions across app years. A plausible reason is that API 19 is now the oldest Android version with a non-trivial market share, while API 23 is the first version adopting the run-time permission mechanism which created quite some execution compatibility issues [9]. Generally, in terms of the absolute RIR numbers, run-time incompatibilities were on the rise (albeit slightly) over the eight-year span we considered, and these compatibilities issues were much more prevalent than installation-time incompatibilities (30–50% RIR versus 15% IIR on average overall all apps).

Finding 4: *Run-time incompatibilities extensively (30–50% RIR) and increasingly persisted on API 23 and older Android versions. Yet newer versions (since API 24) had no run-time compatibility issues with apps created in the studied span.*

5.2 RQ2: Distribution of App Incompatibility Symptoms

The percentage distribution of apps that encountered execution failures over various symptoms is shown in Figure 6. The symptoms were represented by 9 keywords (as listed in the legend) most frequently appeared in the traces of failed app executions. As for the installation-time incompatibility symptoms (Figure 3), here we show the aggregated distribution for each app year over all the Android versions given the similarity of per-version distributions.

For any of these app years, the run-time incompatibilities were predominated by three symptoms: *verify error*, *native crash*, and *null pointer*, in a descending order of dominance. Despite fluctuations in their relative portions, these symptoms largely remained substantial over the years. Given the top dominance of *verify error* and the fact that we already know all these symptoms as caused by (run-time) incompatibilities, SDK/API changes (to which verify errors are imputed) were plausibly the primary cause of these incompatibilities. The reason is that the apps were compiled against some older SDKs and then ran on newer ones, while this kind of inconsistencies is generally a major cause of verify errors [26, 41]. *Native crash* can be attributed to bugs in the

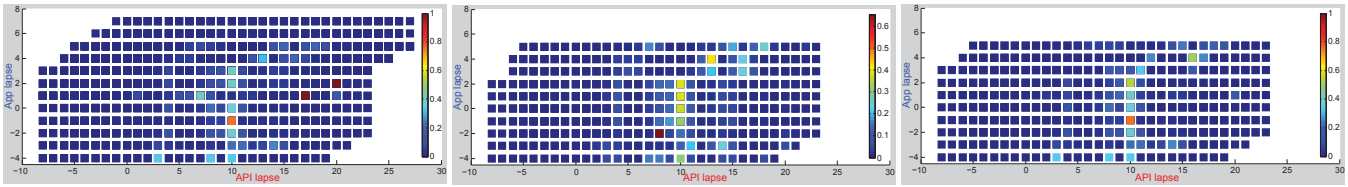

Figure 7: Distribution of RIR (encoded by square color) over app lapse and API lapse, for all symptoms (leftmost) and the two top dominating ones: `verify error` **(middle) and** `native crash` **(right).**

Android Support Library [40] and more generally to problems with the native (C/C++) code layer of Android itself. While the Support Library is an Android effort for overcoming compatibility issues, it only provided support for less than 23% of newly introduced APIs [28]. *Null pointer* errors in this context are often just an additional/derivative symptom of such issues [23]. Thus, we regard SDK/API changes as the main cause of run-time incompatibilities.

> **Finding 5:** *Run-time incompatibilities were primarily exhibited via verify errors (over 50%) and native crashes (over 20%) as top dominating symptoms, which were mainly caused by SDK/API changes during Android evolution.*

5.3 RQ3: Contributing Factors of App Incompatibilities

As in Study I, we examined the distribution of RIR over the same possible contributing factors. In particular, Figure 7 highlights the distribution with respect to app lapse and API lapse, over all symptoms and separately for the top two symptoms, in the same format as Figure 4. Each data point (circle) in the plot represents the RIR of all the apps with a specific app lapse and a specific API lapse, regardless of their creation years. In the overall distribution (left), the RIRs were generally much lower than those seen in the per-symptom distributions. This is because the overall distribution included all of our benchmarks regardless of their app lapses and API lapses, and we now know that those benchmarks that were associated with platform API levels 24, 25, 26, and 27 were all of zero RIRs (i.e., the corresponding apps did not show any run-time incompatibilities). The two per-symptom distributions, however, only concern the apps that exhibited run-time incompatibilities during their executions (i.e., having RIRs greater than zero).

Referring to the overall distribution, unlike our observations in Study I, larger API lapses were not associated with higher RIRs, while what really impacted the RIRs were the app lapses. In fact, the association was quite the opposite. For example, fixating at API lapse of 20, the RIRs were mostly zeros for greater app lapses of 5 to 7 yet substantially higher for smaller app lapses (e.g., 2 and 3). The two per-symptom distributions also revealed the relatively weak impact of API lapse, and that larger absolute values of app lapse were associated with lower RIRs. Notably, in terms of these two top symptoms, significant RIRs were mostly concentrated in the areas where the app lapse was in $[-4, 4]$, implying that apps had most run-time compatibilities issues on the Android versions (within API 19, 21, 22, and 23) that were 4 years older or newer than they (the apps) were. An implication of this result is that Android's promise for backward compatibilities, albeit claimed by

Table 3: Spearman correlation (moderate or stronger coefficients in boldface) between RIR and contributing factors

	app lapse	API lapse	minSdkVersion	SDK API level	app year
overall	**-0.46**	0.117	-0.274	**-0.513**	-0.052
native crash	-0.266	0.092	-0.179	-0.278	-0.034
verify error	**-0.437**	0.123	-0.268	**-0.482**	0

Android to offer [22], were not well fulfilled during the evolution of the Android framework. Apparently, here we see that Android's support for forward compatibilities cannot be always anticipated either (e.g., API 23, introduced in late 2015, saw over 60% RIRs for apps developed in years 2016 and 2017).

These visually-appearing correlations between RIR and app/API lapses were largely corroborated by the Spearman correlation coefficients of Table 3. Overall, app lapse was significantly (with moderate coefficients) correlated to RIR, as was the cases in which only the RIRs associated with verify errors were concerned. The negativity of the correlation confirmed our observations above: higher RIRs were connected to shorter app lapses (in terms of the absolute values of this app property). The correlation strengths with respect to native crashes were weaker than those to verify errors, because the latter had much greater dominance as seen in Figure 6. On the other hand, among the other potential contributing factors, API level (Android version) was most strongly correlated with RIR, suggesting that SDK/API changes might have been highly responsible for the run-time incompatibilities we observed. This further implies that SDK API level (accordingly the Android version release year) was the main reason why app lapse had significant impact on RIR (given the negligible impact of app year). Note that the negative correlation between RIR and SDK API level was largely attributed to the fact that the four newest (largest) Android versions had seen no incompatibilities in our apps.

> **Finding 6:** *App lapse and SDK API level had relatively strong correlations with RIRs: Apps of a greater app lapse had lower RIRs on higher Android versions.*

6 THREATS TO VALIDITY

One threat to *internal validity* of our study results lies in possible errors in the implementation of experimentation utilities (tools and scripts). To reduce this threat, we have conducted careful code review of our own toolkit and scripts in addition to manual verification of their functional correctness against our experimental design. We have further done so by manually validating partial experimental results against selected

benchmarks. Other tools used in our study are part of the Android SDK, which have been used widely by researchers and developers.

The primary threat to the *external validity* of our study results concerns our choice of benchmark apps. While we have attempted to use a large set of apps collected from a variety of sources, the huge number of Android apps available to users or on the various app markets still renders our selection a relatively small subset. Thus, the studied apps from each of the eight years may not be well representative of all Android apps of that year. Our results and conclusions based on the results are limited to the apps studied.

The coverage of run-time inputs affects the ability of dynamic analysis and the quality of its results in general. While the dynamic analysis employed in our study is simple, the identification of run-time incompatibility effects was immediately subject to how much of the apps' behaviors were exercised during the five-minute execution. The random inputs generated by Monkey might have missed some execution paths, hence certain incompatibility effects, of the apps. Also, although we ensured same test inputs against each app across the eight runs (each for a different Android version), the app might still have been covered differently in different runs due to changes in the Android platform across the versions—an external threat to the validity of our run-time incompatibility study results.

Yet another threat to *external validity* concerns the multi-APK phenomenon in Android: a developer may upload multiple APKs of an app to support different devices, and Google Play will provide the correct APK to users as per the characteristics (e.g., hardware, vendor, and Android version) of their devices. To preliminarily evaluate the possible impact of this phenomenon on our results, we randomly chose 10 apps that were found incompatible with API 21 in our study and tried manually installing (from Google Play directly) and running them on a real Samsung Galaxy S4 (API 21) phone—in our study these apps were downloaded from AndroZoo without using a device ID. Our results show that all 10 apps are still incompatible, suggesting that multiple APKs may not exist for these apps. However, a systematic examination of all our benchmarks against this phenomenon would be needed to thoroughly assess its impact. Similarly, we did not consider the possibility that an app, out of our benchmarks and found incompatible, may have an updated version provided later that fixed the compatibility issues. Together, these possibilities cause our study to suffer from potential over-estimation of incompatibilities.

The main threat to *construct validity* concerns the metrics and measurement procedures we used to assess the extent and distribution of app incompatibilities. There might have been other measures and metrics we missed that would better or further support our conclusions. To mitigate this threat, we have considered a diverse set of measures to characterize the compatibility issues in Android apps from multiple perspectives. In addition, we examined our data via various ways of measurement, including the group statistics for understanding the overall characteristics (e.g., the aggregate RIR) and the statistics of yearly subsets for understanding relevant evolutionary patterns. Also, our current study did not address app incompatibilities with respect to the *targetSdkVersion* specified in apps (e.g., IIR of apps on the platform of an API level that is equal to, versus smaller/greater than, the app's *targetSdkVersion*). Addressing this aspect could affect our overall conclusions on installation-time incompatibilities (especially those on Android's forward and backward compatibility). Finally, the eight AVDs used in our studies all used x86/x86_64 processors. However, some of our benchmarks might have been developed for ARM architectures. For those benchmarks, the installation incompatibilities exhibited via the `INSTALL_FAILED_NO_MATCHING_ABIS` errors may be justified by the apps' architecture preferences. Running these benchmarks on devices with ARM processors could lead to different IIR results hence changes to our current conclusions based on such results.

Due to heavy overheads of our study, we only considered a single set of hardware configuration parameters for all the eight devices (corresponding to the eight Android versions studied). Thus, a threat to *conclusion validity* is that our results may not generalize to all possible hardware configurations of Android devices in use. Thus, our conclusions are best interpreted with respect to the device API levels and configurations already used in our study. Given the dominating symptoms and corresponding causes of the incompatibilities we found and our case study above (with the 10 apps on the Samsung phone), the incompatibilities appeared to be attributed to the Android ecosystem (rather than to the Nexus One device we used). Thus, we expect our results would be similar if the study were conducted on a different device (such as a Samsung phone). However, to be fully conclusive, we would need to experiment with more than one hardware devices.

7 LESSONS & RECOMMENDATIONS

Based on our empirical findings as presented before, we now distill further insights into app incompatibilities and provide practical recommendations on how to deal with those issues.

7.1 On Installation-time Incompatibilities

Lessons learned. Concerning installation-time compatibility issues only, while it seems that these issues are significantly related to the age of an app, what really matters is the *minSdkVersion* specified in the app. Installing an app to an Android version with a delta within 10 between its *minSdkVersion* and the platform API level might not be a big issue, but going too far could risk failing the installation, especially when the gap goes beyond 15. When the installation did fail, it is most likely because the app uses some native functionalities that are not supported by the targeted hardware architectures.

Recommendations. For an app to be successfully installed to a device, the developers should carefully specify the *minSdkVersion*, making sure it is not too far from the API level of targeted Android platform. Since *minSdkVersion* would be defaulted to 1 if it is not specified in an app [25], not specifying this attribute would be a risky decision (as the API lapse would be large, especially with respect to newer Android versions). Also, to avoid installation failures, the developer should check if the targeted devices support all the app functionalities in terms of the hardware architecture of the devices and the Android customizations by the device vendors.

7.2 On Run-time Incompatibilities

Lessons learned. Compared to installation-time incompatibilities, run-time compatibility issues are a greater problem in the Android

ecosystem prior to API 24, suggesting relevant efforts made by the Android team (e.g., providing Support Libraries) to deal with these issues were not sufficient. After an app is successfully installed to a device, the *minSdkVersion* specified in the app is no longer a major concern (as it does not affect much whether the app can run well on the device). The major concern, with older Android versions (before API 24), is that the SDK an app is compiled against might not be compatible with the SDK it runs against due to API changes. In particular, during the years of those older versions, the corresponding SDK/API changes seem to need four years to iron out most forward and backward compatibility issues. The good news is that since API 24, Android seems to have made a giant leap in addressing run-time incompatibilities in general.

Recommendations. With apps targeting Android versions before API 24, the developers are recommended to avoid building the apps based on Android SDKs that are released more than four years away. Further, to minimize run-time compatibility issues, the developers should look into the API changes made in the SDK compared to prior SDK versions to make sure APIs used in the app respect those changes. When the resulting apps fail during execution, messages/logs on verify errors and native crashes might be good places to look into when identifying and diagnosing potential compatibility issues. For app users, updating to newer/latest Android versions would be a good idea in order to use more apps without much trouble. If they stick to older platforms (prior to API 24), they should realize that apps developed in years closer to the platform year might be even easier to fail.

8 RELATED WORK

In earlier works, researchers have been concerned about the compatibility issues in Android apps. In order to obtain empirical evidences of the Android fragmentation phenomena, Han et al. [27] focused on two Android device vendors (HTC and Motorola) to study related bug reports submitted by users of these devices, so as to identify the evidences through topic models and topic analysis. The authors aimed at offering an understanding of the fragmentation problem itself, but not of the compatibility issues that problem had led to.

Several previous works investigated crashes of Android apps to understand their causes [10] and to reproduce the crashes [36]. The crashes studied were not necessarily tied to app incompatibilities, though. We only looked at the crashes that are due to compatibility issues in order to characterize app incompatibilities. Fazzini et al. [11] devised an automated tool for detecting the inconsistent behaviors of apps across different Android platforms. Incompatibilities can be part of the reasons for the detected behavioral differences, yet our work focuses on the incompatibilities themselves and their effects.

In [44], Wei et al. studied 191 instances of fragmentation-induced compatibility (FIC) issues in five Android apps and characterized the causes of FIC issues from this dataset. They further devised a tool for automatically detecting FIC issues based on their empirical findings. While Android fragmentation is a lasting cause of these issues, it is not the only cause. Zhang et al. [47] proposed an approach to testing the compatibility of apps for reducing the testing cost. Recent studies [28, 32] focus on compatibility issues induced by SDK evolution and API changes, where these changes, along with developers' strategies dealing with the issues, are characterized.

In contrast, our study is not as limited to a specific cause, and with a much larger scale than prior peer works. We also studied the compatibility issues at different phases (installation and execution) that were actually *observed*, opposed to characterizing/detecting such issues based on static analyses of app's code. The detection approaches are of a predictive nature thus suffer from false positives as evidenced in their results [28, 32]. In comparison, all the compatibility issues we studied are *true-positive* issues. Also, none of those previous studies addressed installation-time compatibility issues, nor did they examine such issues with an evolutionary view. On the other hand, their results are complementary to ours in that they covered developers' practice in preventing and fixing incompatibilities. Also, our study results can be leveraged to, but not limited to, help devise better incompatibilities diagnosis techniques.

A few studies on Android applied the evolutionary lens to certain characteristics of apps. McDonnell et al. [35] examined how the Android API evolved prior to the year of 2013 by looking at the API update rate in Android. They revealed that the API evolution was faster than the adoption pace of clients. In [42], the authors investigated the evolution of malware but with respect to the effectiveness of anti-malware analysis tools, rather than examining the incompatibilities in malware. In comparison, our study characterizes the incompatibilities in benign apps. Also, we investigated the evolution of app incompatibilities, as opposed to that of the API evolution which is a cause of app incompatibilities. Compared to previous evolution studies, ours also spanned a much longer period of time (of eight years). Recently, we started looking into developers' intentions for app compatibilities [48], which provides another angle of understanding the causes of incompatibilities but limited to installation-time issues.

9 CONCLUSION AND FUTURE WORK

We conducted a large-scale study of app compatibility issues in Android, concerning the occurrences of these issues at installation time in 62,894 apps and those exercised at runtime in 15,045 apps. We characterized both types of compatibility issues in terms of their prevalence, distribution, and evolutionary patterns over an eight-year span from 2010 through 2017. We examined the relationships between app incompatibilities and the API level that was specified in apps themselves and that was used by the device. We introduced two derivative app properties relevant to incompatibilities, app lapse and API lapse, and investigated the correlation between them, along with other individual app/platform properties, with compatibility issues at installation and run time. This study design has enabled us to discover many new findings, which further led us to novel lessons regarding compatibility issues in Android apps and to practical recommendations for dealing with those issues. Part of our future work is to expand our study by running the experiments on more devices of various hardware configurations to gain deeper understanding of hardware relevance to app incompatibilities.

ACKNOWLEDGMENT

We thank the reviewers for their constructive and insightful comments, which helped us tremendously improve this paper.

REFERENCES

[1] Hispasec Sistemas (a Spanish security company). 2019. VirusTotal. https://www.virustotal.com/. Last accessed: May 20, 2019.

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 468–471. https://doi.org/10.1145/2901739.2903508

[3] Haipeng Cai and John Jenkins. 2018. Leveraging Historical Versions of Android Apps for Efficient and Precise Taint Analysis. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 265–269. https://doi.org/10.1145/3196398.3196433

[4] Haipeng Cai and John Jinkens. 2018. Towards Sustainable Android Malware Detection. In *IEEE/ACM International Conference on Software Engineering (ICSE), Poster track*. 350–351. https://dl.acm.org/authorize?N661848.

[5] Haipeng Cai, Na Meng, Barbara Ryder, and Danfeng (Daphne) Yao. 2019. DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling. *IEEE Transactions on Information Forensics and Security (TIFS)* (2019), 1455–1470. https://doi.org/10.1109/TIFS.2018.2879302

[6] Haipeng Cai and Barbara Ryder. 2017. Artifacts for Dynamic Analysis of Android Apps. In *International Conference on Software Maintenance and Evolution (ICSME), Artifacts track*. 659. https://doi.org/10.1109/ICSME.2017.36

[7] Haipeng Cai and Barbara Ryder. 2017. DroidFax: A Toolkit for Systematic Characterization of Android Applications. In *International Conference on Software Maintenance and Evolution (ICSME)*. 643–647. https://doi.org/10.1109/ICSME.2017.35

[8] Haipeng Cai and Barbara Ryder. 2017. Understanding Android Application Programming and Security: A Dynamic Study. In *International Conference on Software Maintenance and Evolution (ICSME)*. 364–375. https://doi.org/10.1109/ICSME.2017.31

[9] Malinda Dilhara, Haipeng Cai, and John Jenkins. 2018. Automated detection and repair of incompatible uses of runtime permissions in Android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 67–71. https://doi.org/10.1145/3197231.3197255

[10] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 408–419. https://doi.org/10.1145/3180155.3180222

[11] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 308–318. https://doi.org/10.1109/ASE.2017.8115644

[12] Xiaoqin Fu and Haipeng Cai. 2019. On the Deterioration of Learning-Based Malware Detectors for Android. In *IEEE/ACM International Conference on Software Engineering (ICSE), Poster track*.

[13] Google. 2019. Android API 24 Changes. https://developer.android.com/studio/releases/platforms#7.0. Last accessed: May 20, 2019.

[14] Google. 2019. Android Compatibility. https://developer.android.com/guide/practices/compatibility.html. Last accessed: May 20, 2019.

[15] Google. 2019. Android Debug Bridge. https://developer.android.com/studio/command-line/adb.html. Last accessed: May 20, 2019.

[16] Google. 2019. Android Developer Dashboard. http://developer.android.com/about/dashboards/index.html. Last accessed: May 20, 2019.

[17] Google. 2019. Android Developer Guide. https://developer.android.com/guide. Last accessed: May 20, 2019.

[18] Google. 2019. Android emulator. http://developer.android.com/tools/help/emulator.html. Last accessed: May 20, 2019.

[19] Google. 2019. Android logcat. http://developer.android.com/tools/help/logcat.html. Last accessed: May 20, 2019.

[20] Google. 2019. Android Monkey. http://developer.android.com/tools/help/monkey.html. Last accessed: May 20, 2019.

[21] Google. 2019. Android Version History. https://en.wikipedia.org/wiki/Android_version_history. Last accessed: May 20, 2019.

[22] Google. 2019. Backwards Compatibility. https://developer.android.com/design/patterns/compatibility.html. Last accessed: May 20, 2019.

[23] Google. 2019. Diagnosing native crashes. https://source.android.com/devices/tech/debug/native-crash. Last accessed: May 20, 2019.

[24] Google. 2019. Google Play Store. https://play.google.com/store. Last accessed: May 20, 2019.

[25] Google. 2019. uses-sdk elements. https://developer.android.com/guide/topics/manifest/uses-sdk-element. Last accessed: May 20, 2019.

[26] Google. 2019. Verify Error. https://developer.android.com/reference/java/lang/VerifyError. Last accessed: May 20, 2019.

[27] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *Proceedings of IEEE Working Conference on Reverse Engineering*. 83–92. https://doi.org/10.1109/WCRE.2012.18

[28] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 167–177. https://doi.org/10.1145/3238147.3238185

[29] Simon Hill. 2016. Android Fragmentation Issue. http://www.digitaltrends.com/mobile/what-is-android-fragmentation-and-can-google-ever-fix-it/. Accessed online 09/20/2016.

[30] howtogeek. 2019. The Ultimate Guide to Installing Incompatible Android Apps from Google Play. https://www.howtogeek.com/138500/the-ultimate-guide-to-installing-incompatible-android-apps-from-google-play/. Last accessed: May 20, 2019.

[31] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*. 1–11. https://doi.org/citation.cfm?id=782008

[32] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. 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*. ACM, 153–163. https://doi.org/10.1145/3213846.3213857

[33] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*. 477–487. https://doi.org/10.1145/2491411.2491428

[34] Xuan Lu, Xuanzhe Liu, Huoran Li, Tao Xie, Qiaozhu Mei, Dan Hao, Gang Huang, and Feng Feng. 2016. PRADA: Prioritizing android devices for apps by mining large-scale usage data. In *Proceedings of the 38th International Conference on Software Engineering*. 3–13. https://doi.org/10.1145/2884781.2884828

[35] Tyler McDonnell, Bonnie Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of IEEE International Conference on Software Maintenance*. 70–79. https://doi.org/10.1109/ICSM.2013.18

[36] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. 33–44. https://doi.org/10.1109/ICST.2016.34

[37] Leann Myers and Maria J Sirois. 2004. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences* 12 (2004). https://doi.org/10.1002/0471667196.ess5050

[38] International Data Corporation (IDC) Research. 2019. Android dominating mobile market. https://www.idc.com/promo/smartphone-market-share/os. Last accessed: May 20, 2019.

[39] stackoverflow. 2019. Android Device Compatibility Issues. https://stackoverflow.com/questions/31009186/android-device-compatibility-issues. Last accessed: May 20, 2019.

[40] stackoverflow.com. 2019. Android native crash. https://stackoverflow.com/questions/47063114/android-native-crash. Last accessed: May 20, 2019.

[41] stackoverflow.com. 2019. Causes of getting a java.lang.VerifyError. https://stackoverflow.com/questions/100107/causes-of-getting-a-java-lang-verifyerror. Last accessed: May 20, 2019.

[42] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 76. https://doi.org/10.1145/3017427

[43] techrepublic. 2019. How fragmentation affects the Android ecosystem. https://www.techrepublic.com/article/how-fragmentation-affects-the-android-ecosystem/. Last accessed: May 20, 2019.

[44] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237. https://doi.org/10.1145/2970276.2970312

[45] Ryszard Wisniewski and Connor Tumbleson. 2019. A tool for reverse engineering Android apk files. https://code.google.com/p/android-apktool/. Last accessed: May 20, 2019.

[46] www.statstutor.ac.uk. 2019. Spearmanąŕs correlation. http://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf. Last accessed: May 20, 2019.

[47] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. 2015. Compatibility testing service for mobile applications. In *IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 179–186. https://doi.org/10.1109/SOSE.2015.35

[48] Ziyi Zhang and Haipeng Cai. 2019. A Look Into Developer Intentions for App Compatibility in Android. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft)*.