Characterizing Installation- and Run-Time Compatibility Issues in Android Benign Apps and Malware

JIAWEI GUO, University at Buffalo, SUNY, USA
XIAOQIN FU, Washington State University, USA
LI LI, Monash University, Australia
TAO ZHANG, Macau University of Science and Technology, China
MATTIA FAZZINI, University of Minnesota, USA
HAIPENG CAI*, University at Buffalo, SUNY, USA

The Android ecosystem has experienced rapid growth, resulting in a diverse range of platforms and devices. This expansion has also brought about compatibility issues that negatively impact user experiences and hinder app development productivity. Existing relevant studies are focused on and limited to the "static" sense of those issues (in terms of potentialities and proneness), while only addressing compatibility issues that possibly occur during app executions. In this paper, we present an extensive and longitudinal study on app compatibility issues that are disparate from yet complementary to prior studies, characterizing the incompatibilities based on *actual*, *exercised observations* and *evidence* at both installation and run time. With a dataset of 74,545 benign apps and 56,919 malicious apps over a span of 12 years (2010 through 2021) and ten Android versions, we extensively examine the prevalence and symptoms/effects and causes of, as well as the contributing factors to, installation-time and run-time compatibility issues.

Our study reveals 12 major novel findings regarding Android app incompatibilities. Firstly (*Findings 1,2*), installation-time incompatibilities persisted significantly over the 12 years, even more so in malware than benign apps. Secondly (*Findings 7,8*), run-time compatibility issues were also seen persistently over time but only on specific Android platforms (such as API 26,27,etc.) and much less by malware than benign apps. Thirdly (*Findings 5,6,11,12*), there is a significant (moderate/stronger) correlation between an app's specified *minSdkVersion* and its incompatibilities (over all symptoms and/or with respect to one of its dominating symptom), with stronger correlations seen in malware than in benign apps, for both installation- and run-time incompatibilities. Similar observations hold (although with much stronger correlation in absolute terms) when considering, instead of the *minSdkVersion* itself, the gap between the app's *minSdkVersion* and the SDK API level of the platform the app is installed to or runs on. Lastly (*Findings 3,4,9,10*), installation-time incompatibilities are primarily caused by the utilization of architecture-incompatible native libraries within apps, while run-time incompatibilities are mainly attributed to API changes during the evolution of the Android SDK; the symptoms of run-time failures seen by malware are much more diverse than by benign apps. In addition to these insights, we provide practical recommendations for both app developers and end users on how to effectively address compatibility issues in Android apps, as well as how to devise effective defenses against malware from the compatibility perspectives.

CCS Concepts: • **Software and its engineering → Maintaining software**.

*Haipeng Cai is the corresponding author.

Authors' addresses: Jiawei Guo, University at Buffalo, SUNY, New York, 14226, Buffalo, USA, jiaweigu@buffalo.edu; Xiaoqin Fu, Washington State University, Washington, 99163, Pullman, USA, xiaoqin.fu@wsu.edu; Li Li, Monash University, Clayton, Victoria, 3800, Australia, li.li@monash.edu; Tao Zhang, Macau University of Science and Technology, Macao, China, tazhang@must.edu.mo; Mattia Fazzini, University of Minnesota, USA, mfazzini@umn.edu; Haipeng Cai, University at Buffalo, SUNY, NY, 14226, Buffalo, USA, haipengc@buffalo.edu.

Manuscript submitted to ACM

Additional Key Words and Phrases: Android, apps, compatibility, installation failure, run-time failure, security, malware

ACM Reference Format:

Jiawei Guo, Xiaoqin Fu, Li Li, Tao Zhang, Mattia Fazzini, and Haipeng Cai. 2024. Characterizing Installation- and Run-Time Compatibility Issues in Android Benign Apps and Malware. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (March 2024), 43 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Given the lasting dominance of Android [59] in the mobile operating system market and due to the shift of personal computing to mobile platforms, most software applications used today are Android apps. Mobile apps, like software in other domains, face various challenges, including compatibility, reliability, and security [11, 16, 64], among other quality issues. Yet while reliability and security have received significant attention in research and industry [8, 12, 13, 21, 24], compatibility issues have not been as extensively attended despite the continuation of research studies on this topic in the past few years [20, 53, 61]. Compatibility issues in mobile apps can significantly impact the user experience, leading to reduced usability and overall app quality. These incompatibilities[1] have a detrimental effect on the health of the mobile software ecosystem. Moreover, compatibility issues hinder the efficient production and adoption of apps, impeding the productivity of app developers. It is intuitive that the larger the user base of the ecosystem, the more severe the negative impact of these compatibility issues [9, 72].

As Android continues to gain momentum, the prevalence of compatibility issues in the Android ecosystem has also increased [42, 58, 65, 73, 74]. The open-source nature of Android has led to a diverse range of Android devices [52] and customized Android platforms, including variations in the Android operating system kernel [23]. Furthermore, the Android system itself undergoes constant evolution, resulting in continuous changes to the software development kit (SDK) and application programming interface (API) [54, 68]. While these developments facilitate the growth of Android in the mobile computing market, they also introduce various incompatibilities in Android apps. These incompatibilities can manifest as apps developed for specific device models or versions of the Android system failing to function properly or being unable to install on different device models or other Android system versions. As people increasingly rely on Android apps for their daily lives, it becomes crucial for both app developers and end users to understand and address application incompatibilities in Android. This understanding serves as a fundamental step towards mitigating and even preventing relevant issues.

Existing studies exist which aim to help understand and/or detect Android app quality problems that are relevant to compatibility issues, including fragmentation [39, 41, 68], app configuration [73], or API evolution [49, 54] as a cause of those issues and app crashes or UI differences [45] as a form of possible externalization of incompatibilities [22, 56]. Moreover, several techniques have been developed to detect/diagnose abnormal app behaviors that potentially result from incompatibilities [23, 45, 68]. However, these works do not directly address app compatibility issues themselves. Also, fragmentation, app configuration, and API evolution are not the only causes of app incompatibilities, nor are app crashes and UI abnormalities the only form of externally observed app behaviors resulting from compatibility issues.

Recent efforts [40, 43, 51, 72] have placed more emphasis on *potential* compatibility issues resulting from API changes, noted as API/evolution-induced incompatibilities. In fact, almost all of the existing techniques that aim to deal with (e.g., detect and repair) app incompatibilities are focused on those due to APIs being improperly used [47, 53, 69]. Studies evaluating those techniques [50, 60, 61] and examining how developers handle compatibility issues in the code of

[1]We use “compatibility issues" and “incompatibilities" as exchangeable terms in this paper.

their apps [40] are almost exclusively aimed at and limited to API/evolution-induced incompatibilities. However, these studies and techniques are generally conservative/predictive, meaning that issues being addressed are potential, not evidenced or manifested, issues. A main reason is that they are *static* in nature. In particular, most of the techniques are based on static code analysis techniques, which are intrinsically subject to false positives. For instance, a compatibility issue is detected when an incompatible API is used [50, 51], yet the use of such an API may or may not actually cause a compatibility issue, depending on the bigger context of the API invocation (e.g., it is on a path that is rarely exercised or guarded by a non-statically-resolvable condition that when met would ensure compatible behaviors).

Furthermore, app incompatibilities have so far only been studied from app maintenance perspectives—that is, understanding or devising solutions on how to ultimately make the app more compatible. Yet malicious apps potentially face compatibility issues too, which have different implications and opposite expectations regarding how to deal with (e.g., making them more compatible is not desirable from a security perspective). Analyzing malware incompatibility also provides unique comparative insights into compatibility challenges across benign and malicious apps, allowing us to distinguish between universal compatibility issues—such as those caused by API evolution or hardware fragmentation—and those specific to malware, often arising from poor construction practices or intentional obfuscation. Moreover, investigating malware incompatibility offers direct security implications: understanding where and why malware fails to install or execute can reveal structural weaknesses in its construction, which could inform the development of better detection and mitigation strategies. However, there have been no studies at all explicitly addressing the compatibility issues particularly in Android malware and how they are similar to or different from those issues seen by benign apps. In short, studies on the security relevance of app incompatibilities are currently lacking. Including malware in this study ensures a holistic view of the Android ecosystem, reflecting its reality as a mix of benign and malicious applications. Also, current studies are generally focused on compatibility issues that would (potentially) occur at app execution time, with few addressing incompatibilities at app installation time. Finally, extant relevant studies typically examined a limited number of app samples and did not consider the temporal aspect of the samples or the specific compatibility issues encountered—which is essential for understanding how the landscape of app incompatibilities changes over time.

As it stands, no study to date has comprehensively **(1)** addressed the actually *observed* (i.e., manifested) app incompatibilities in Android, **(2)** taking into account both a *large-scale* perspective and an *evolutionary* viewpoint. Furthermore, there is a lack of research **(3)** examining the various symptoms of these issues at different stages of app usage, such as *installation* and *execution*, along with their underlying causes. Finally, there has been no study **(4)** explicitly examining compatibility issues in *benign apps versus malware* (i.e., commonalities and differences). A comprehensive study addressing these gaps would right complement to existing peer studies by providing valuable insights into the understanding of compatibility issues in Android apps based on *evidence and observations* rather than proneness and potentialities, including their evolution over time and *security relevance*. Such insights could help inform strategies for mitigating and preventing these issues, while informing the design of effective defenses against malware from compatibility perspectives.

To address these research gaps, in this paper, we conducted a comprehensive and longitudinal study at a large scale to investigate **observed** incompatibilities in Android apps. We collected a dataset of 131,464 apps (including 74,545 benign apps and 56,919 malware) from various sources, spanning 12 different years of development (2010 through 2021). From this dataset, we profiled a subset of 41,204 apps (including 22,529 benign apps and 18,675 malware) with random inputs for a duration of five minutes each. Our study focused on two types of incompatibilities: *installation-time* and *run-time* incompatibilities. We examined these incompatibilities by analyzing the corresponding APKs and their

execution traces on ten major Android versions with significant market share (from API level 19 through 29) [27]. An app was considered *installation-time incompatible* if it could be successfully installed on at least one Android platform but not on another, as indicated by the return code or message of the installation process. Similarly, an app was regarded as *run-time incompatible* if it could run successfully, without producing any system error messages, exceptions, or crashes, on at least one Android platform but not on another. Differentiating these two classes of app incompatibilities allowed us to gain an in-depth understanding of the extent and phases of compatibility issues in Android apps.

Through our datasets and the two complementary studies focusing on installation-time and run-time compatibility issues, respectively, our study aims to assess the prevalence and distribution of these compatibility issues. We examine various symptoms and causes related to app incompatibilities, including the efforts made by app developers to achieve compatibility and the effects of incompatibilities—we immediately examine symptoms (i.e., effects) and then connect them to respective root causes based on official documentations that describe the root cause of each symptom [25, 33, 38]. Furthermore, we investigate the impact of key properties of apps, such as their age, properties of the Android platform, and release time, on app incompatibilities. By analyzing the differences in compatibility issues between benign and malicious apps, we also explore possible relationships between app compatibility and app security. Additionally, our study takes a longitudinal view to uncover the evolutionary patterns of app incompatibilities over time. We aim to understand the differences in compatibility issues between benign and malicious apps and reveal how these compatibility characteristics evolve in both groups.

In particular, we explore the following set of research questions. The first three concern the **installation-time incompatibilities** of Android apps:

- **RQ1**: *How prevalent are installation-time app compatibility issues in Android?* Prior research has indicated the potential presence of compatibility issues in Android apps, primarily attributed to specific factors like SDK evolution [40, 47, 54, 68]. However, in order to comprehensively understand compatibility issues arising from any possible cause, our study takes a different approach. We actually installed each app sample to directly observe and characterize the compatibility issues that arise during actual installation of apps on various Android platforms.
- **RQ2**: *How are the compatibility issues in Android apps distributed over major symptoms?* App incompatibilities manifest through noticeable symptoms, such as error logs during installation failures and crash traces during execution failures, which can be attributed to compatibility issues [22, 56]. By examining the factors that contribute to incompatibilities based on specific symptoms, we can gain insights into the underlying causes of these symptoms (again based on the official documentations about the links between the effects and causes). Particularly for answering this question, we conducted an analysis of the primary symptoms associated with installation-time incompatibilities, and investigated the distribution of installation failures across these symptoms.
- **RQ3**: *What are the main factors that contributed to the installation-time incompatibilities in Android apps?* The existing knowledge about Android app incompatibilities primarily attributes them to problems with the Android platform, such as fragmentation and SDK evolution [40, 68]. In order to enhance this understanding, we conducted additional research to investigate potential app properties that may contribute to compatibility issues. Specifically, we performed statistical analyses to identify possible correlations between installation failures and various app properties, such as the age of apps and their specification of the minimum SDK version.

And three similar questions concern the **run-time incompatibilities** of Android apps:

- **RQ4**: *How prevalent are run-time app compatibility issues in Android?* Similarly to RQ1, we aim to assess the prevalence of app incompatibilities. Yet differently from RQ1, we address run-time incompatibilities here, understanding what

actually happens when incompatible behaviors are exercised. We ran each installation-time compatible app in our RQ1 datasets for five minutes and observed manifestation of any of those behaviors.

- **RQ5**: *How are the run-time compatibility issues in Android apps distributed over major symptoms?* Similarly to RQ2, we further assess the effects and causes of app incompatibilities, yet differently we look at run-time incompatibilities here. We identify the run-time symptoms of incompatible behaviors based on the error messages, system logs, and exceptions/crashes that occur during app executions.
- **RQ6**: *What are the main factors that contributed to the run-time incompatibilities in Android apps?* Similarly to RQ3, we further examine the same relevant app properties as potential contributing factors, but now focusing on those to run-time incompatibilities. To that end, we performed similar statistical analyses to identify correlations between those factors and run-time failures induced by app incompatibilities.

Importantly, to address the security relevance of app incompatibilities, we aim to answer one additional, **cross-cutting question**: *How are benign apps and malware similar and different in terms of the prevalence, symptoms (and hence causes), and contributing factors with respect to incompatibilities that occurred in them?* We answer this cross-cutting question by investigating the common characteristics and differences between the two app groups[2] when answering the respective questions among the foregoing six (i.e., RQ1, RQ2, and RQ3 for installation-time incompatibilities, and RQ4, RQ5, and RQ6 for run-time incompatibilities).

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

(1) A significant portion (on average 15% in benign apps and 25% in malware) of our benchmark apps from different years experienced incompatibilities that prevented their installation on one or more of the ten Android platforms we studied. These issues were largely independent of app ages but significantly correlated with the *minSdkVersion* specified in the apps and the difference between this version and the API level of the platform where the app was being installed.

(2) Approximately 91% of the installation-time incompatibilities were caused by the use of native library functionalities in the apps that were not supported by the underlying hardware architecture. In some cases, the issues were primarily attributed to customizations made by device vendors to the Android system. Malware had much more diverse symptoms, with another primary cause (up to 20% in certain years) consisting in problematic APK construction.

(3) Run-time compatibility issues were even more prevalent (12-year average of 20.77%) in the benign apps, but notably less (14.48%) in malware, than installation-time issues. These issues were primarily caused by the API level of the Android platform. In particular, platforms of API levels prior to API 28 saw such issues more frequently than the newer API levels studied.

(4) Run-time compatibility issues were predominantly manifested as verify errors (56% on average) and null pointer dereferences (14% on average), which were primarily attributed to SDK/API changes during the evolution of the Android SDK. Not surprisingly, apps with a minimum SDK version that is more distant from the API level of a platform were more likely to exhibit run-time incompatibilities on that platform. Error-prone construction practice of malware was another main cause of their run-time failures.

(5) Both installation-time compatibility and run-time compatibility issues are primarily influenced by SDK version differences (between specified and that of the target platform) rather than chronological development timing. Installation failures show scattered patterns with complex interactions between SDK versions and API levels,

[2]Hereafter, we refer to benign apps and malware as the two **app groups** in our studies.

while run-time failures exhibit strong relationships with the gap between platform API versions and specified minimum SDK requirements. Malware demonstrates more systematic compatibility patterns compared to benign apps, with consistent behaviors across different types of failures. These patterns indicate fundamental differences between hardware and framework compatibility issues in the Android ecosystem, with hardware-related failures showing more concentrated patterns while framework issues are more widely dispersed.

Based on these observations and findings, we extensively discussed their implications and provided actionable insights and recommendations for developers and end users of Android apps regarding how to deal with incompatibilities and develop better defenses against malware. To ensure these findings and insights remain relevant with the latest Android platform versions, we also conducted an additional study examining compatibility patterns on Android 14 (API 34) and Android 15 (API 35). The results, while showing unique patterns, confirm that they are still in accordance with our key observations and conclusions about both installation- and run-time incompatibilities of applications in Android. This paper extends its preliminary version [18] in many ways, from which the key differences and new contributions are elaborated in Section 9.

All of the source code and datasets used in our study can be found at https://figshare.com/s/4b2a7aedd25f50894fbe.

2 BACKGROUND

In this section, we provide concise explanations of the basic concepts and terminology related to the Android system and Android apps. These descriptions are essential for readers to comprehend the subsequent sections of this paper.

2.1 Android Platform and SDK

The Android framework serves as an intermediary layer between the Android operating system (OS), which is based on a customized Linux kernel, and user applications. This framework is responsible for implementing the application programming interface (API) methods that enable user apps to access system services and utilize common functionalities associated with mobile devices. The API is typically bundled as part of the Android software development kit (SDK), which includes various tools to support app development.

In the framework-based development paradigm, Android apps are composed of different components. These components are the building blocks of an app and can be categorized into four types: *Activity*, which forms the basis of the user interface, *Service*, which performs background tasks, *Broadcast Receiver* (or simply *Receiver*), which responds to system-wide broadcasts, and *Content Provider*, which offers database capabilities.

2.2 Incompatibilities in the Android Ecosystem

In the Android ecosystem, it is common to encounter devices with different hardware configurations running the same version of the Android operating system. These hardware differences include, among others, variations in processor models, screen sizes, and the presence or absence of certain sensors. Additionally, different device manufacturers often customize the Android system to align with their specific mobile device products. This phenomenon of hardware and software diversity is known as *fragmentation* in the Android ecosystem [39]. The presence of fragmentation poses challenges for app developers, as it becomes difficult to ensure that an app functions properly on all devices and Android versions. This leads to compatibility issues that are specifically attributed to fragmentation [68].

To meet the diverse needs of its market, Android offers a wide range of OS and SDK versions, both of which undergo constant evolution. The SDK, particularly the API, tends to evolve at a faster pace [27]. Over the past 15 years, Android

has released more than 30 API versions, each associated with a specific *API level*. This API evolution introduces a significant factor contributing to incompatibilities in Android apps [54]. For instance, due to the API evolution, an app developed with one API level might not be installable or runnable with Android of different API levels.

In the context of Android, there are two types of compatibility: *device compatibility* and *app compatibility* [25]. Device compatibility refers to whether a mobile device is capable of running apps designed for the Android runtime. Only Android-compatible devices include the Google Play Store, which serves as the primary platform for users to install apps. Therefore, device compatibility is typically not a concern for app developers. On the other hand, app compatibility is of utmost importance to app developers, as it encompasses incompatibilities that may arise due to device configurations, the Android system (particularly the framework and API), or a combination of both. Therefore, in this study, our focus is primarily on app compatibility.

2.3 Compatibility Attempts in Android Apps

When developing an app, the developer has the option to specify the API level that the app targets and the minimum API level required for the app to function [74]. These API level specifications are typically recorded in the app's manifest file (AndroidManifest.xml)—part of the app package (i.e., APK)—as *minSdkVersion* (the minimum API level) and *targetSdkVersion* (the targeted API level).

According to the official Android Developer Guide (OADG) [28], it is recommended to always declare the *minSdkVersion* in the manifest file, or else it will default to 1 [35]. Since API level 4, apps can also declare the *targetSdkVersion* attribute, although it is optional. If unspecified, it will default to the same value of *minSdkVersion*. Additionally, starting from API level 4, apps can (albeit they are not recommended to) declare another API level number called *maxSdkVersion*, which specifies the maximum SDK version that the app is compatible with.

As per the OADG, an Android system does not allow the installation of an app if the system's API level is lower than the app's *minSdkVersion* or higher than its *maxSdkVersion*.[3] However, Android promises *backward compatibility* [33], meaning that an Android platform with an API level higher than the app's *minSdkVersion* allows the app to be installed and function as expected. If the platform's API level is higher than the app's *targetSdkVersion*, the system also enables compatibility behaviors to ensure the app functions correctly [35].

Nevertheless, between API level 4 and API level 6, the check for *maxSdkVersion* is enforced. If an app specifies a *maxSdkVersion* that is lower than the platform's API level, the installation will fail.

2.4 Repackaging in Android

Repackaging in Android refers to the process of modifying an existing Android application package (APK) file by replacing its original contents with malicious or unauthorized components. It involves extracting the APK file, modifying its code, resources, or other assets, and then reassembling the package with the malicious changes [48].

Repackaging can have significant implications for compatibility in Android apps. When an app is repackaged, its original code and resources are altered, which can introduce compatibility issues with the Android operating system, other apps, or specific device configurations. These compatibility issues arise due to the following typical reasons, among others:

- *Inconsistent signatures*: Repackaging an app typically involves modifying its digital signature, which verifies the authenticity and integrity of the app. If the app's signature is altered during repackaging, it will no longer

[3]The check against *maxSdkVersion* has been abolished by Android since its version 2.0.1 [35] (which corresponds to API level 6 [32]).

match the original signature. This can result in compatibility problems when the modified app interacts with other components that rely on the app's original signature for verification.

- *Modified functionality*: Repackaging can involve modifying the app's code or resources to alter its behavior or introduce malicious functionality. These modifications can lead to compatibility issues if the app relies on specific APIs, frameworks, or libraries that are no longer present or have been altered in the repackaged version.
- *Incompatible modifications*: Repackaging may introduce changes that are incompatible with the Android system or specific device configurations. For example, modifying the app's manifest file or permissions can cause conflicts with the system's security model or prevent the app from accessing necessary resources or services.
- *Version mismatch*: Repackaging can result in discrepancies between the app's declared version information and its actual code and resources. This can lead to compatibility problems when the app interacts with other components that rely on accurate version information for compatibility checks or feature compatibility.

Overall, repackaging can disrupt the intended functionality and behavior of an app, making it incompatible with the original Android system, other apps, or specific device configurations.

3 METHODOLOGY

This section provides an overview of our study process and outlines the dataset and tools utilized. We then introduce the main metrics and measures employed to quantify installation-time and run-time app incompatibilities, hence addressing our research questions.

3.1 Process Overview

Figure 1 illustrates the overall process of our study. We utilized both benign and malicious APKs as benchmarks to examine the differences in characteristics regarding app incompatibilities between these two groups. Our datasets encompassed samples developed over a span of 12 years (2010–2021[4]) to facilitate an evolutionary perspective in analyzing app incompatibilities. We considered a total of 10 different Android versions (API level 19 through 29 [5]), excluding API 20 which is specifically designed for wearable devices. The first 8 versions collectively account for 96.5% of the entire Android market share as of late 2018 [27].[6] Other versions were excluded due to their outdated status (released in 2012 or earlier) and negligible market share (1.5% or lower).[7]

We examine both installation-time and run-time incompatibilities of apps in our study. To **characterize installation-time incompatibilities**, we attempt to install the original APK of each app on an Android device for each studied Android version. We collect the installation logs and analyze them to determine whether the installation was successful or resulted in a failure. The logs also provide information to understand the effects of installation failures. We consider uninstallation success as part of installation success, so after successfully installing each app, we uninstall it as part of the installation test.

[4]We started our extension study in 2019 after we presented our preliminary (conference version) paper that year; collecting the additional two years of apps and their execution traces, especially those for the 12 years of malware samples, took us over four years.
[5]By summer 2019 when we started the extension study, the newest Android version that had been released was API 28. But there were very little market share of that version due to its short life then. Later, for the revision of this paper that started in late 2023, we further added API levels 28 and 29 to our studies—our studies are extremely costly, mainly for the run-time experiments; adding the two SDK versions plus two more years of sample apps, the additional experiments took 339 machine days. Such prohibitive costs prevented us from including even newer API levels and newer years of apps.
[6]This data was obtained in May 2019, the last time Google updated its Android Distribution Dashboard that includes the percentage of Android devices running each version of Android—after that, Google stopped doing so [37], making the official distribution no longer known ever since.
[7]Again, the market share data was up to May 2019, the last time such data was available according to Google's public release of the Android version distribution dashboard.

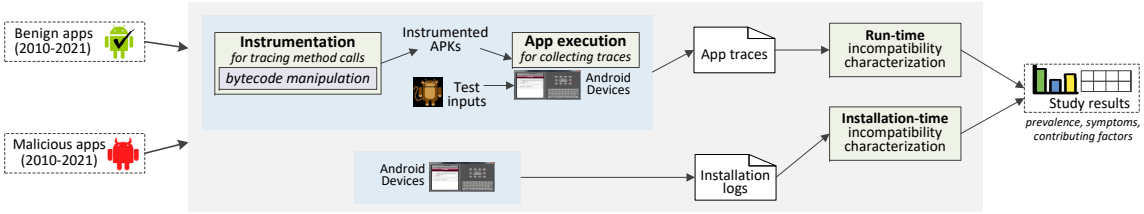

Fig. 1. Overview of the process flow of our app incompatibility study. The installation-time and run-time incompatibility characterization pipeline, represented by the grey box in the middle, is shared by the study on benign apps and that on malware.

To **characterize run-time incompatibilities**, we need to analyze app executions. We employ lightweight instrumentation and profiling techniques to differentiate two categories of run-time incompatibilities: (1) *incompatible launch*, where an app fails to launch successfully, and (2) *incompatible running*, where a launch-compatible app exhibits incompatibilities after running for some time. To achieve this, we instrument each app to trace all method calls using the Soot framework [46] for Dalvik bytecode manipulation using tools dedicated to dynamic Android app characterizations [15, 16]. We then run the instrumented app for five minutes on an Android device for each of the 10 Android versions, collecting the app trace and system log during its execution. By examining the trace, we can distinguish between the two run-time incompatibility situations: if the trace contains valid records of calls, we exclude the app from the *incompatible launch* category. We identify the effects of run-time incompatibilities by further analyzing the system log.

The outputs of our study pipeline are the characterization results for both types of incompatibilities. This pipeline was applied to benign apps in exactly the same way as it was applied to malware. For each app group, we examine three kinds of incompatibility results: the *prevalence* of incompatibilities, the *symptoms* (effects) of compatibility issues when they occurred, and the *contributing factors* to the incompatibility-induced (installation or run-time) failures, in accordance with the three research questions we aim to answer for each app group. Importantly, through the existing knowledge about the *root cause* underlying each of the incompatibility effects (symptoms) according to the official Android documentation [38], we are enabled to understand the root cause of each incompatibility instance observed in our study via the symptom the compatibility issue demonstrates.

In the next three subsections, we provide further details on the key elements of our study design: subject apps (Section 3.2), experimental procedure (Section 3.3), and measurements (Section 3.4).

3.2 Subject Apps

Table 1. Subject apps used in our study

Data Use	App Group	Number of Samples from Each Year (2010-2021)												Total
		2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	
Installation-time	Benign Apps	16,835	9,977	10,991	9,688	5,300	5,406	2,431	2,266	3,184	2,467	3,000	3,000	74,545
	Malware	2,140	12,451	3,841	11,079	5,229	5,823	2,940	2,075	3,100	2,241	3,000	3,000	56,919
													Total (Installation-time Incompatibility Study)	***131,464***
Run-time	Benign Apps	1,531	2,020	2,054	1,750	1,335	3,127	1,548	1,680	1,325	1,322	2,500	2,337	22,529
	Malware	1,316	1,303	1,303	1,305	1,351	1,307	1,301	1,308	1,303	1,318	2,878	2,682	18,675
													Total (Run-time Incompatibility Study)	***41,204***

Table 1 provides a summary of the subject Android apps used in our study, specifically 74,545 benign apps and 56,919 malicious apps. These apps were developed in different years from 2010 through 2021. In particular, the 2,266 apps from 2017 were directly downloaded from Google Play [34], while the remaining benign apps were obtained from AndroZoo [6], which is a diverse collection of apps from various sources. The malware subjects from 2013 through 2016 were downloaded from VirusShare [67], and the malware of all other years was downloaded from AndroZoo, where we set a threshold of 10 positive detections of VirusTotal [3] in each app's metadata recorded by Androzoo.

Despite the diversity of AndroZoo in terms of its data sources, we still tried further to diversify our study subjects by considering Google Play and VirusShare as additional sources. Including Google Play apps ensures that our dataset captures apps actively distributed on the most widely used official platform, thereby enhancing representativeness and avoiding biases that may arise from relying solely on AndroZoo. Additionally, VirusShare is a dedicated malware repository that complements AndroZoo's malware collection by providing distinct samples that might not be available elsewhere. Using these additional sources, we ensured greater diversity and comprehensiveness in our dataset, capturing broader behaviors and compatibility challenges for both benign and malicious apps. One likely attempt would be to have balanced numbers of samples across these years. However, this is not always feasible in practice—nor necessary since actual numbers of apps across years are not necessarily balanced.

During our data collection, we excluded corrupted APKs that could not be unzipped or were missing resource files. While these corrupted apps are not installable, they are not relevant to app incompatibilities. The *minSdkVersion* attribute of each app is needed for our study. Thus, for those apps that do not explicitly specify this attribute,[8] we defaulted it as 1 just as Android does [35]. For each app group, we ensured that all the 12 yearly datasets are mutually disjoint—there are no apps shared by any two of these 12 datasets. Ultimately, we utilized a total of 131,464 (74,545 benign + 56,919 malicious) apps for the installation-time compatibility study.

For the run-time compatibility study, we had to limit the number of samples due to the execution overhead (i.e., running each app for five minutes on each of the 10 Android versions). We initially selected the apps from the installation-time study for each year. However, since many of these apps failed to install on one or more of the Android versions, we continued selecting additional apps from the respective sources until we had at least 1,000 installable apps for each year. In total, we used 41,204 (22,529 benign + 18,675 malicious) apps for the run-time compatibility study.

3.3 Experimental Setup and Procedure

To ensure the scalability of our study and control study overheads, we utilized 10 Android virtual devices (AVDs) for our study. All AVDs were Nexus One devices with 2GB RAM and 1GB SD storage. However, they had different API levels, specifically 19, 21, 22, 23, 24, 25, 26, 27, 28, and 29. We chose these API versions based on their significant proportions in the market share distribution of different Android platform versions with respect to the timing and targeted range of app ages for our study, as justified earlier. We ran these AVDs using the Android emulator [29] provided with each corresponding Android version.

To generate run-time inputs for exercising apps hence profiling the app behavior—as needed for studying run-time incompatibilities, we employed the Monkey tool [31], which is included in the Android SDK. The Monkey tool generates random inputs for apps, which suffice for revealing incompatibilities, if any, in our study—previous works have used this tool for extensively characterizing the run-time behaviors of benign and malicious apps [10] and effectively detecting malware [13]; the tool itself also has an industrial level of strength and stability compared to many, more advanced

[8] 0.29-1.25% of the apps in our yearly datasets did not specify *minSdkVersion*.

research prototypes that are not as stable and widely adoptable. We serialized the app call traces and system logs using the Logcat tool [30], also part of the Android SDK. To instrument the apps and trace method calls, we utilized our Android characterization toolkit [15] and dynamic analysis utilities [14]. These tools allowed us to collect the necessary data for our run-time incompatibility study. To retrieve app manifest data, including the *minSdkVersion*, *targetSdkVersion*, and *maxSdkVersion*, we used the *apktool* [71]. For app installation and uninstallation, we utilized the *adb* tool [26], which is the Android debug bridge. These tools and utilities facilitated the data collection process for our study.

Using the study facilities mentioned above, we can now provide precise definitions for the terms we previously mentioned intuitively. For an app installation or uninstallation to a device A to be considered successful, the execution of the `install` (`uninstall`) command of *adb* on the app to device A must return a code explicitly indicating success. An app is considered installation-time compatible with device A if it can be installed to and then uninstalled from device A successfully. However, if the app fails to install or uninstall on device A, it can only *potentially* be considered installation-time incompatible with device A. The reason is that the failure to install an app to a device may not necessarily be due to (installation-time) compatibility issues. To determine if the failure is truly caused by compatibility issues with one device, we also tried installing it to another device (i.e., with a different Android version in our study) until it can be successfully installed on that device or all of the other 9 devices are exhausted. In the end, we consider that the failure is actually due to compatibility issues with device A (i.e., the app is indeed installation-time incompatible with A) if the app can be successfully installed to at least one different device among the 9. By doing so, we exclude apps that fail to install on a specific device due to reasons other than compatibility issues—i.e., no installation-time-compatible device can be found among the 10 Android versions we considered.

Additionally, Android does not allow apps with a *minSdkVersion* higher than the underlying platform's API level to be installed [35]. Therefore, we do not consider installation failures of such apps on those platforms as being induced by (installation-time) compatibility issues. Regarding apps with a *maxSdkVersion* lower than the underlying platform's API level, they are not allowed to install on Android versions of API level 4, 5, and 6. However, since our study focuses on Android versions of API level 19 or higher, the check policies against *maxSdkVersion* do not affect our results. [9]

In a similar manner, failure in executing an app on a device A, as indicated by execution error messages, exceptions, and crashes in the system log, may be attributed to reasons other than (run-time) compatibility issues, such as bugs in the app or invalid user operations. To exclude failures that are not induced by compatibility issues with a particular device, we ran the app on various devices with different Android versions (all for five minutes with Monkey inputs). If the app can successfully run on at least one different device without exhibiting any execution failure symptoms, but it fails during execution on device A (also for five minutes and with the same sequence of Monkey inputs used in the successful run), we consider the app as run-time incompatible with device A. Consequently, the error messages, exceptions, and crashes observed during the failing execution on device A are considered as run-time incompatibility symptoms. Of course, if an app can successfully run on a device A without exhibiting any execution failure symptoms, the app is simply considered run-time compatible with A.

Given the above process of determining app to be installation-time and run-time compatible or not, one of the challenges in our study was to identify and exclude irrelevant failures that were not induced by compatibility issues during app installation and execution. This process was time-consuming, especially when finding successful executions, due to the significant overhead of dynamic analysis, such as running each app for five minutes and analyzing traces and system logs for failure symptoms. However, since we installed and executed each sample on eight different Android

[9] In our initial datasets of year 2010 to 2019, a small percentage (0.52–3.6% with a mean of 2.1%) of apps had *maxSdkVersion* specified, and an even smaller number of (less than 5) apps of any year had *minSdkVersion* higher than 19.

versions, we were able to exclude irrelevant failures for each version by referring to the results obtained from the other seven versions. We would have needed to try additional devices only if the installation or execution of an app failed on all eight versions, which fortunately did not occur for any of the samples in our study.

3.4 Measurements

To address our research questions, we calculated several measures for both classes of app incompatibilities. For installation-time incompatibilities, we computed the *installation-time incompatible rate* (**IIR**) for each set of apps (apps from a specific year) with respect to an API level. This rate represents the percentage of apps that cannot be successfully installed on the AVD with that API level. We also computed the distribution of IIR over major installation-time incompatibility effects (i.e., symptoms) so as to understand the causes of the respective compatibility issues according to how we bridge the effects and causes as described earlier. Additionally, we examined the evolutionary pattern of installation-time incompatibilities by studying the changes in both the IIR and the distribution of IIR measures over time (years).

Similarly, for run-time incompatibilities, we calculated the *run-time incompatible rate* (**RIR**), but separately for the two subclasses: *incompatible launch* and *incompatible running*. We further analyzed the distribution of RIR based on major run-time incompatibility effects (symptoms).

In addition, we performed a series of statistical analyses to identify correlations between app incompatibilities and various properties of the Android platform and its apps. Specifically, we focused on compatibility-related platform properties such as *release year* and *API level*, which have been studied in prior works [40, 47]. Additionally, we investigated several app properties that could potentially impact app incompatibilities, including *minSdkVersion*, *app (creation) year*, *app lapse*, and *API lapse*[10]. We obtained the SDK release years according to the Android version history [32].

We proposed and studied the two derivative properties (app lapse and API lapse) because intuitively they can be used to examine the length of Android's forward and backward compatibility with apps [47]. To measure the correlations of interest, we calculated the Spearman's correlation coefficients [57] for relevant variables. We chose this method because it is a non-parametric correlation statistic that does not assume the normality of the underlying data points. In interpreting the correlation strengths, we referred to [5] and considered the absolute value of the coefficient. A coefficient range of $[0.0, 0.09]$ indicates a very weak or negligible correlation, $[0.1, 0.29]$ suggests a weak correlation, $[0.3, 0.49]$ represents a moderate correlation, $[0.5, 0.69]$ indicates a strong correlation, and $[0.7, 1.0]$ signifies a very strong correlation.

Next, we present the findings from our empirical studies on installation-time and run-time incompatibilities in Android apps. Our studies aim to examine the prevalence, contributing factors, effect (symptom) and cause distribution, and security relevance of these incompatibilities from an evolutionary perspective. To enhance clarity, we have conducted two separate studies: Study I focuses on installation-time incompatibilities (Section 4), while Study II focuses on run-time incompatibilities (Section 5). In both studies, we address three research questions and discuss the major findings pertaining to these questions. Within each of these six research questions, we address our *cross-cutting concern* regarding the security relevance of app compatibility/incompatibility by examining both the common characteristics and differences between the two app groups in terms of each of the three aspects (i.e., prevalence, effect/symptom distribution, and contributing factors). On top of these research questions, we also extend our investigation to examine

[10]Here, app lapse refers to the difference between the SDK release year and the app year, while API lapse represents the difference between the platform's SDK API level and the app's *minSdkVersion*. That is, app lapse = (SDK release year − app year) and API lapse = (platform's SDK API level − app *minSdkVersion*)

how incompatibility patterns observed in our main study translate to the newest Android versions. We randomly sample apps from all years in our dataset (2010-2021) and analyze both installation-time and run-time incompatibility issues. Thus, in total, we will have 13 answers to be presented as 13 major findings, as summarized in respective finding boxes hereafter.

4 STUDY I: INSTALLATION-TIME INCOMPATIBILITIES

As previously stated, an app is considered to be installation-time compatible when it can be both installed and uninstalled successfully. Commonly, if an app can be successfully installed on a device, it is expected that it should also be successfully uninstalled from the same device. Our study findings validated this expectation, as all the subject apps that were successfully installed to any device were also successfully uninstalled from that device.

4.1 RQ1: Prevalence of Installation-Time App Incompatibilities

Figure 2 illustrates the overall IIR of the studied benign versus malicious apps across the twelve-year period, as well as the collective IIR of all the benign and malicious apps considered in this study as an aggregated dataset (referred to as "all of 12 years"), despite the varying symptoms of installation failures. Note that the overall average is computed as a weighted mean with the weights being the numbers of benign/malicious apps of individual years used for the installation-time incompatibility study (see Table 1), in order to account for the uneven numbers of samples we have from different years.

Common characteristics. Regardless of being benign or malicious, a high-level *common* observation for apps of both groups reveals that, within each yearly dataset, the failure rate remained relatively consistent across almost all of the ten different Android versions for all the studied years, suggesting a generally negligible impact of the platform's API level on app installation compatibility. In fact, the combined results of all the apps confirmed this overall observation.

However, results for the latest two years (2020 and 2021) show a distinctive pattern: for benign apps, while the IIR on API 19 increased significantly (reaching 23.03% in 2021), the failure rates on newer API levels (27-29) dropped notably (around 6-7%). This represents a dramatic shift from 2019 (where API 28-29 showed approximately 40% IIR), suggesting a major change in development practices. This pattern differs from the previous years (2018-2019) where high IIRs were seen across multiple older API levels (19-23). The reason lies in developers' compatibility attempts: most newer apps specified a minSdkVersion targeting newer API levels, leading to installation failures on older platforms but ensuring better compatibility with newer ones. Notably, this strategic targeting has resulted in a clear three-tiered pattern in 2020-2021: very high IIR for API 19 (>20%), moderate for APIs 21-26 (7-10%), and low for APIs 27-29 (<7%). This implies that developers have become increasingly conscious of app compatibility requirements, particularly with newer Android platforms (API 24 and above).

Similarly, for malware, the pattern is also notably different in years 2020–2021, showing consistently high IIRs across most API levels (ranging from 33% to 39%), with API 19 showing particularly high rates (reaching 32.01% in 2021). This uniformity in high IIRs suggests systematic compatibility issues rather than targeted version support. An interesting observation is the steady increase in IIRs from 2019 to 2021 (38.94% to 32.20% overall), indicating growing installation challenges for malicious apps despite improved Android compatibility mechanisms—an interesting, natural defense line against malware.

Another consistent pattern across both benign and malicious apps is the notably lower IIR on API 27 compared to other API levels. This unique characteristic persists even in the newer API levels (28 and 29), which generally show

Fig. 2. Installation-time incompatibilities in terms of IIR in benign apps (y axis, top) and malicious apps (y axis, bottom) over the twelve studied years of 2010 through 2021 and the combined results for all of the 12 years (x axis).

similar IIR patterns to APIs 24-26. This "API 27 effect" is particularly pronounced in malware (11.57% compared to 28% for surrounding API levels), suggesting specific technical characteristics of this API version that affect installation compatibility differently. Looking at the overall trends across all years, benign apps show an average IIR of 14.12% while malware exhibits a significantly higher average of 25.29%, demonstrating that malicious apps continue to face greater installation-time compatibility challenges across Android versions.

The combined results across all twelve years (2010-2021) show that while there is some variation in IIR across different API levels, these differences are less pronounced than in individual years, suggesting that the Android ecosystem has generally evolved to provide more consistent compatibility behavior across versions, particularly for newer APIs (24-29). This evolution shows a clear pattern of convergence in benign apps' IIRs for newer APIs (standard deviation decreasing from 13.2% in 2019 to 2.1% in 2021 for APIs 27-29), while malware maintains more volatile patterns.

Notably, there was no clear or consistent association between dataset sizes and IIR, suggesting that the significant variations in dataset sizes did not substantially impact these general observations. This independence from dataset size holds true across both app types and all API levels studied.

> **Finding 1:** *Over the twelve years studied, installation-time app incompatibilities persisted independently of Android versions overall, yet Android API levels 19 and 27 had greater improvements in resolving app installation compatibility than other SDK versions.*

Benign apps versus malware. As discussed above, these two app groups have much in common in installation-time compatibility, including the evolutionary trends and the effects of Android platform versions (API levels). The differences are mainly twofold.

First, in terms of absolute numbers, malware continues to show higher installation failure rates than benign apps. Specifically, over the individual app years, the IIR of benign apps ranged from low rates in early years (0.01-0.45% on API 19 during 2010-2017) to higher rates in recent years (reaching 45.58% on API 23 in 2019). The aggregate IIR across all years for benign apps was 14.53%. In contrast, malware exhibited consistently higher incompatibilities, with IIR

reaching up to 45-46% across multiple API levels in 2019, and particularly high rates in 2020-2021 (peaking at 38.72% on API 24 in 2020). The aggregate IIR across all years for malware was 24.25%. Per our inspection, we found that the main reason for this persistent gap between malware and benign apps is that a large portion of the malware is repackaged. Repackaging is well-known as a common practice and productive way of producing malware [48], which tends to result in invalid APKs[11] and other complex compatibility issues. In fact, as we presented and discussed later in Section 4.2, our malware datasets have seen a higher presence of invalid APKs, especially in the malware from 2010 and 2017, as a more frequent installation failure symptom. In comparison, our benign apps have little inclusion of repackaged apps, which is consistent with earlier findings about repackaging in benign apps [17].

Second, the evolution of IIR patterns across years reveals distinct characteristics between the two groups. While benign apps show clear phase transitions over time (from consistently low rates in 2010-2017, a sharp increase in 2018-2019, to a more targeted pattern in 2020-2021), malware demonstrates a more gradually increasing trend. This is particularly evident in the progression of peak IIRs: malware shows steadily increasing maximum rates across years (from 8% in 2010 to 47% in 2019, maintaining 33-39% in 2020-2021), whereas benign apps show more abrupt changes (staying below 20% until 2017, jumping to 40% in 2018-2019, then dropping to selective high rates only for specific APIs in 2020-2021). These distinct evolutionary patterns suggest fundamentally different approaches to API compatibility between legitimate and malicious development – benign apps show evidence of adaptive responses to platform changes, while malware exhibits a pattern of consistently increasing installation challenges across the Android ecosystem.

A distinctive pattern emerges in 2020-2021: both groups show increased IIRs on API 19 (reaching 23.03% for benign apps and 32.01% for malware in 2021), suggesting that maintaining backward compatibility with very old Android versions remains challenging. However, the patterns diverge significantly for newer APIs, with malware showing consistently high IIRs (27-42% across APIs 21-26) compared to benign apps (7-15% for the same range). This divergence is particularly pronounced in the transition between API levels—while benign apps show sharp drops in IIR between certain API levels (e.g., 23.03% to 11.96% between API 19 and 21 in 2021), malware maintains more uniform failure rates across consecutive APIs, suggesting different compatibility management approaches between legitimate and malicious app developers.

> **Finding 2:** *Overall, malware continues to experience higher installation-time incompatibility issues than benign apps (with average IIR of 24.25% versus 14.53%). While benign apps show clear phase transitions in compatibility patterns over time, malware exhibits steadily increasing incompatibility rates across years, reaching and maintaining high IIRs across most API levels in recent years.*

4.2 RQ2: Distribution of Installation-Time App Incompatibility Symptoms

Figure 3 illustrates the distribution of installation failure effects due to compatibility issues across the 9 failure effects with non-trivial (at least 1%) app attribution in this study. These effects were error codes, such as `INSTALL_FAILED_INVALID_APK`, detected during installation using *adb*. Although we calculated the distribution for each Android version separately, the distributions were highly similar. Thus, we present and analyze the aggregate distribution, combining installation failure effects across all of ten Android platform versions for each app year.

[11] These invalid APKs are different from the corrupted ones we discarded during data collection as discussed in Section 3.2 that cannot be unzipped or have missing resource files—they are invalid mainly due to inconsistent code format and structure across different code regions within the Dex files, which cannot be readily observed until actually getting installed to a device.

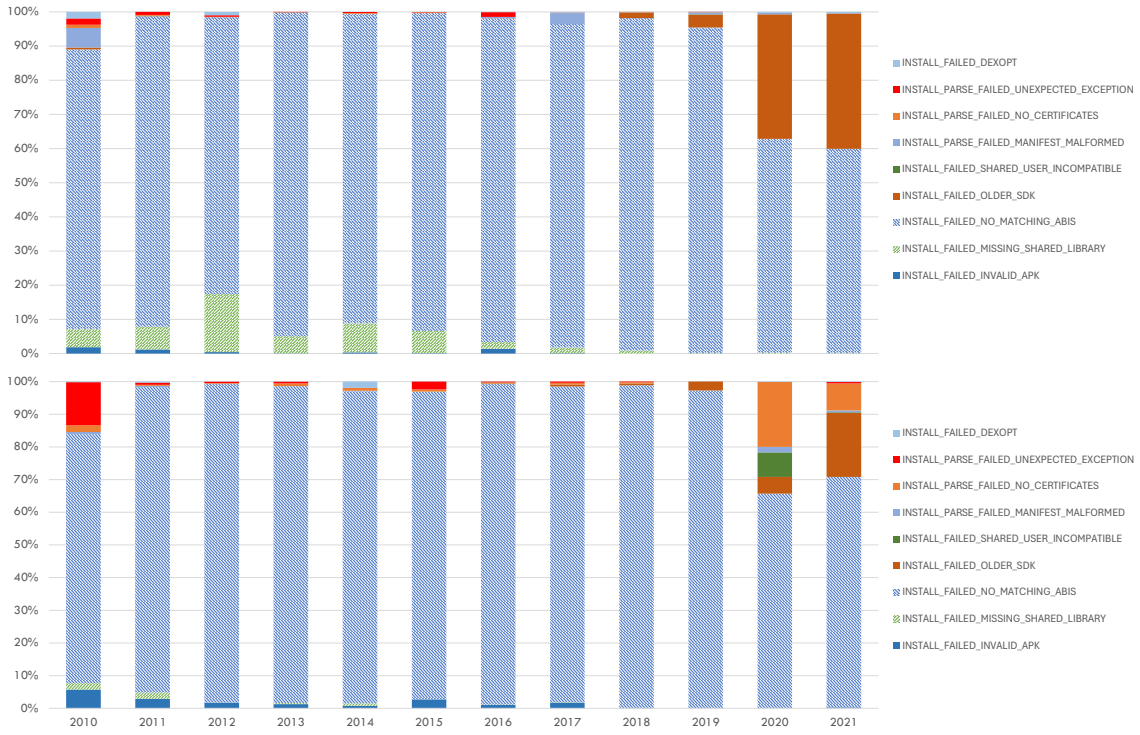

Fig. 3. Percentage distribution of installation-time incompatible benign apps (y axis, top) and malicious apps (y axis, bottom) across the ten studied years of 2010 through 2021 over the varied symptoms they exhibited as observed in their installation failures.

Common characteristics. Despite the diverse range of symptoms, it is clear that for both benign apps and malware there are only relatively a few noticeable types of symptoms, especially in benign apps. As a result, even after we dismissed the symptoms with a trivial app attribution, many of the (non-trivial) 9 symptoms still do have a readily visible representation on the charts. In particular, the single most (*commonly* between the two app groups) dominant symptom was `INSTALL_FAILED_NO_MATCHING_ABIS`. In fact, the majority of failed installations (86.38% in benign apps and 90.06% in malware across all years) were attributed to this error[12], which indicates that the apps utilized native libraries incompatible with the processor architecture. For instance, installing an app built for Intel x86 CPUs to a device with the ARM architecture would fail with this error.

A notable shift is observed in the latest years (2020-2021), where `INSTALL_FAILED_OLDER_SDK` emerged as a significant symptom in both benign apps (reaching 39.58% in 2021) and malicious apps (reaching 19.64% in 2021). This indicates an increasing trend of apps targeting newer SDK versions, leading to compatibility issues with older platforms in Android.

Another notable symptom common between the benign and malicious apps in earlier years was `INSTALL_FAILED_MISSING_SHARED_LIBRARY`. Specifically, this symptom showed higher prevalence in benign apps (averaging 4.50% overall), with peaks in certain years like 2012 (16.96%); for malware, while historically significant before 2014, its prevalence has decreased to zero in recent years after 2017. These failures were caused by the use of

[12] According to how we excluded non-compatibility-induced installation failures (§ 3.3), it was confirmed that these apps ran normally (did not crash nor encounter such errors) on at least one different Android version out of the ten studied, an essential part of our experimental methodology.

libraries that were missing from the underlying Android framework, hence symptomized as the error code of INSTALL_FAILED_MISSING_SHARED_LIBRARY. The situation often occurred due to the removal of those libraries during vendor customizations.

> **Finding 3:** *The primary cause of installation failures in apps was their dependency on architecture-incompatible native libraries, accounting for over 86% of cases in benign apps and 90% in malware across all years studied. This dominance persisted through 2019, but showed a notable decline in 2020-2021 where it dropped to around 60-70% due to the emergence of newer Android SDK versions incompatibilities with their older versions as another major cause (reaching 39.58% in benign apps and 19.64% in malware by 2021).*

Benign apps versus malware. For the benign apps, aside from INSTALL_FAILED_OLDER_SDK becoming significant in 2020–2021 (reaching 39.58%), no other substantially notable symptoms were observed beyond the common two (e.g., INSTALL_FAILED_NO_MATCHING_ABIS and INSTALL_FAILED_MISSING_SHARED_LIBRARY) that dominated their failure distributions. This suggests that benign apps were largely consistent in their compatibility behaviors, with newer apps experiencing issues primarily tied to outdated SDK requirements. The rise of INSTALL_FAILED_OLDER_SDK in recent years also reflects developers increasingly targeting higher API levels, potentially as a response to platform evolution and security updates.

In the malware group, however, the symptoms were clearly more diverse and exhibited significant year-to-year variations. In addition to the common symptoms, several notable issues emerged, such as INSTALL_PARSE_FAILED_NO_CERTIFICATES, which persisted as a secondary but consistent contributor to installation failures and peaked at 19.86% in 2020. Notably, while these issues were occasionally observed in benign apps, their prevalence in malware was significantly higher, suggesting a stronger association with poor or malicious APK construction practices.

Specifically, the symptom INSTALL_FAILED_INVALID_APK tends to be more malware-specific, as it is rarely present in benign apps. This partly confirms what we found during our further inspection when examining the generally much higher IIR in malware than in benign apps (Section 4.1). The widespread adoption of repackaging in malware construction has contributed significantly to this symptom. Repackaging often leads to improper or corrupted APK composition/format, rendering many malicious apps invalid for installation. This symptom was particularly prevalent in earlier years, such as 2010, 2011, and 2015, but has steadily declined in recent years, dropping to below 1% (hence invisible in the figure) in 2019–2021. This decline may reflect an evolution in malware construction techniques, with attackers adopting more sophisticated methods to bypass installation barriers.

Our further investigation revealed another two main reasons that led to so many invalid APKs. The first is a *signature mismatch*, where the APK file has an incorrect or mismatched signature—Android requires that APK files be signed with appropriate certificates and keys, a requirement that quite some malicious apps do not meet. If the APK file has been modified or is signed with a different certificate, the installation will fail with this symptom. This situation commonly occurs when attempting to install an updated or a modified version of an app, which apparently is again related to the high prevalence of repackaging in malware production—the repackaging process naturally results in the updates/modifications that cause certificate inconsistencies. The second is *insufficient permission*: If the APK file requires certain permissions that are not granted on the Android device, the installation can fail. A plausible further explanation for this cause of invalid malware APKs is that malware often asks for excessive permissions. Note that some other known reasons for this symptom, such as APK corruption and installation of apps with the same package

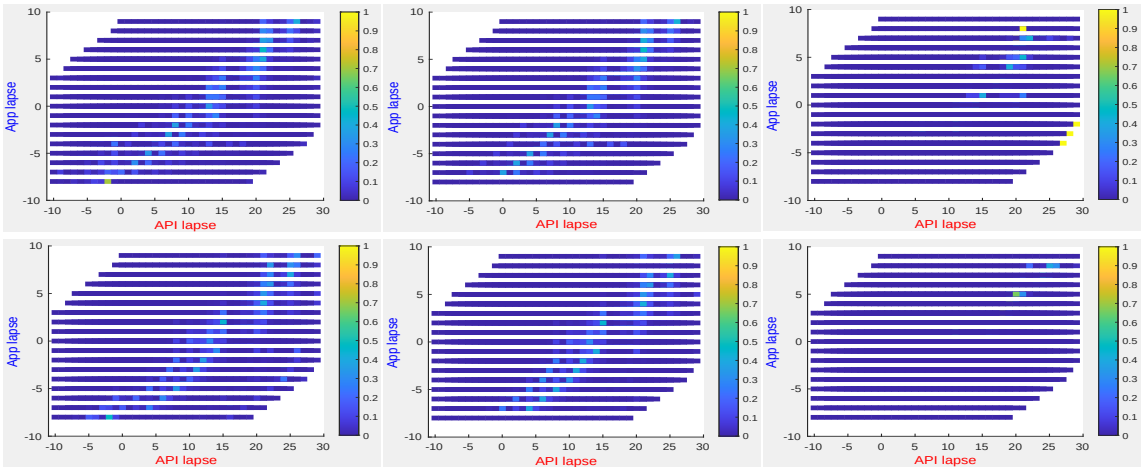

Fig. 4. Distribution of IIR (encoded by square color) over *app lapse* (y axis) and *API lapse* (x axis) between benign (top) versus malicious (bottom) apps, for all symptoms (leftmost) and the two dominating ones that are common to two app groups: `INSTALL_FAILED_NO_MATCHING_ABIS` (middle) and `INSTALL_FAILED_MISSING_SHARED_LIBRARY` (rightmost).

names, were not relevant here: the corrupted APKs (i.e., those that cannot be unzipped or have missing resource files) were ruled out during our data collection (Section 3.2), and we assured that each app installation was attempted on a clean device (i.e., in the fresh status without any app installed).

Another significant malware-specific symptom, `INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION`, while showing high percentages in earlier years (13.17% in 2010), has seen a decline in recent years (below 0.5% in 2020-2021). The main underlying reason relevant with respect to our experimental procedure is that the APK file has syntax errors or incorrect formatting in its manifest file or other essential components. The Android package installer expects the APK file to adhere to specific XML formatting rules. When there are any inconsistencies or errors in the APK's manifest or resource files, the APK will fail to be parsed with unexpected exceptions. Other reasons include conflicts with libraries or dependencies, especially those that are external, and resource/asset issues (e.g., those with incorrect file paths or unsupported file formats). Once again, these miscellaneous symptoms are more or less related to how the malicious apps were produced (e.g., through integrating heterogeneous components of incompatible/inconsistent sources, such as during repackaging).

Finding 4: *Compared to benign apps, malware has exhibited more diverse symptoms of installation failures. While invalid APKs and parsing failures were historically significant (up to 13% for unexpected exceptions), recent years (2020-2021) show new patterns with certificate issues (up to 19.86%) and SDK version incompatibilities (up to 19.64%) becoming dominant, in addition to the common symptoms that cover most of the benign apps' installation-time incompatibility issues.*

4.3 RQ3: Contributing Factors of Installation-Time App Incompatibilities

To gain further insights into the causes of installation-time incompatibilities and inform on developing strategies for addressing these issues, we analyzed the distribution of IIRs based on both app lapse and API lapse. This analysis is

visualized in the heatmaps shown in Figure 4, which includes the overall distribution as well as separate distributions for the two prevalent symptoms that were found (in Section 4.2) common to both the benign and malware app groups. In every heatmap, each row of color squares represents the various API lapse values (i.e., API level deltas) and each column the app lapse values (i.e., #years), forming a matrix. Since there are not always apps corresponding to each pair of API lapse and app lapse values, some elements in the matrix correspond to zero IIR values simply because there are no apps associated with the respective pair of lapse values—not because the apps had an average IIR of zero (i.e., having seen no installation-time compatibility issues at all).

Common characteristics. The overall distribution (left) reveals a more complex pattern of installation-time compatibility issues than previously observed. Instead of clear correlation with API lapse ranges, the heatmaps show scattered hotspots of higher IIRs across various combinations of app and API lapses. These hotspots appear as localized regions of elevated installation failures, suggesting that compatibility issues have become more nuanced and less predictable based on version differences alone.

A notable pattern emerges in the form of diagonal clusters of higher IIR values, particularly visible when API lapse values range between 15–25. This diagonal distribution suggests that installation failures are influenced by the interplay between chronological development timing (app lapse) and SDK version differences (API lapse), rather than either factor in isolation. These findings indicate that apps targeting platforms with a substantial API lapse—while maintaining a moderate app lapse—face increased challenges due to evolving platform requirements and developer adaptation strategies in the Android ecosystem.

The impact of app lapse has evolved to show more significance than previously observed, particularly in negative app lapse regions (where apps are older than their target SDKs). This indicates that apps developed before their target SDK's release face distinct compatibility challenges, potentially due to mismatches in API evolution and app features. However, these challenges manifest differently across API lapse values, suggesting a multi-dimensional relationship between development timing and API level (SDK version) differences.

When examining symptom-specific distributions, distinct patterns emerge for `INSTALL_FAILED_NO_MATCHING_ABIS` (middle) versus `INSTALL_FAILED_MISSING_SHARED_LIBRARY` (rightmost). ABI-related failures show more concentrated hotspots, indicating that hardware compatibility issues cluster around specific app-age and SDK-version combinations. In contrast, library-related failures exhibit a more dispersed pattern, suggesting that framework compatibility issues are less tied to specific combinations and are more broadly distributed across different app and API lapse values. This broader distribution reflects the varied nature of shared library dependencies and the complexities of addressing them across diverse Android devices.

These patterns suggest that installation-time compatibility has evolved into a multi-dimensional challenge where successful installation depends on intricate interactions between development timing, SDK version differences (between app specification and the hosting platform), and specific types of compatibility requirements. The emergence of diagonal IIR clusters and symptom-specific behaviors highlights the need for developers to adopt more sophisticated testing and compatibility strategies, targeting specific app and API lapse interactions to mitigate failure risks.

The Spearman correlation coefficients presented in Table 2 reveal several moderate or stronger relationships common to both app groups, underscoring the role of technical factors in installation compatibility. For benign apps, we observe moderate correlations between ABI-related failures and both API lapse (0.34) and minSdkVersion (-0.30). Similarly, malware exhibits slightly stronger correlations with API lapse (0.38) and minSdkVersion (-0.34) for ABI failures—and much stronger over all symptoms (0.45 and 0.43 for API lapse and minSdkVersion, respectively). These consistent patterns

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

App group	Symptom	app lapse	API lapse	minSdkVersion	SDK API level	app year
Benign	overall	-0.02	**0.31**	-0.28	0.06	0.06
	ABI	0.04	**0.34**	**-0.30**	0.14	0.04
	LIB	0.19	0.24	0.17	0.24	-0.10
Malware	overall	-0.07	**0.45**	**-0.43**	0.08	0.12
	ABI	-0.06	**0.38**	-0.34	0.12	0.13
	LIB	0.15	0.15	-0.11	0.13	-0.11

highlight the influence of SDK version differences, with API lapse positively correlated and minSdkVersion negatively correlated with installation failures. This suggests that larger gaps in API lapse or poorly aligned minSdkVersion values increase the likelihood of ABI-related compatibility issues.

Generally, the correlation between IIR and the apps' *minSdkVersion* is similar to that with API lapse. Considering that API lapse is essentially based on the minimum SDK version specified in apps, the similarly strong correlation here indicates that *minSdkVersion* contributes significantly to the correlation strength observed between IIR and API lapse.

Both groups show weak correlations between installation failures and chronological factors, such as app lapse (-0.02 for benign, -0.07 for malware), SDK API level (0.06–0.08), and app year (0.06–0.12). These results indicate that temporal aspects of development, including when an app was created relative to its target SDK, play a minimal role in determining installation compatibility. Instead, compatibility challenges are driven more by SDK version (API level) mismatches than by the timing of development (either that of the app and the Android platform/SDK).

Library-related failures (LIB) exhibit weak correlations across all factors for both groups, with coefficients generally below 0.24. This suggests that library compatibility issues are likely governed by more complex interactions, such as dependency management and platform-specific behaviors, which are not fully captured by simple metrics like API lapse or minSdkVersion specifications.

These patterns indicate that installation-time compatibility issues in Android apps, regardless of being benign or malicious, are primarily influenced by technical version differences (API lapse and minSdkVersion). The weak influence of chronological development factors underscores the fundamental role of version alignment in the Android ecosystem. Developers and security practitioners should focus on addressing version-related compatibility challenges, especially for ABI-related issues, to improve installation success rates and enhance defenses against malware.

Finding 5: *Installation-time compatibility shows moderate correlations with both API lapse and minSdkVersion, with malware exhibiting stronger relationships than benign apps. Rather than a clear "safe range," compatibility issues manifest in scattered hotspots across version differences, suggesting that installation failures are influenced by complex interactions between SDK targeting and platform versions. Hardware-related incompatibilities (ABI) show more concentrated patterns of failure compared to the more dispersed framework customization issues (LIB).*

Benign apps versus malware. While both app groups show similar overall patterns of installation-time compatibility issues, there are notable distinctions in how version-related factors affect their installation success. The correlation analysis reveals consistently stronger relationships in malware compared to benign apps——for both API lapse (0.45 vs. 0.31) and *minSdkVersion* (-0.43 vs. -0.28). This indicates that SDK version differences more strongly influence installation compatibility for malicious apps, likely due to their frequent use of repackaging or incomplete adherence to

platform standards. That the slightly stronger correlations observed in malware highlight its sensitivity to API lapse and minSdkVersion mismatches also suggests malware's reliance on compatible API configurations to ensure successful installation as facilitation of defense against malware.

The heatmap visualizations further underscore these distinctions. Malware exhibits more concentrated hotspots of installation failures, while benign apps show more dispersed patterns. This suggests that malware's installation-time compatibility issues are comparatively more predictable based on version relationships, possibly due to their systematic targeting of specific platform versions or configurations to maximize impact. For ABI-related failures, malware shows a slightly stronger correlation (0.38) compared to benign apps (0.34), reflecting a more systematic dependence on version alignment. On the other hand, library-related failures show weak correlations for both groups (0.15–0.24), indicating that such issues may arise from more varied and less version-dependent factors, such as specific device environments or library dependencies.

Interestingly, the diagonal patterns of high IIR values are more pronounced in malware, highlighting the combined effect of app lapse and API lapse in determining installation success. This diagonal clustering suggests that malicious apps face heightened compatibility challenges when targeting platforms with significant API differences, especially when their development timing more distantly misaligns with the platform's SDK release. This may reflect different development practices in malware creation, such as a greater reliance on automation or less rigorous testing, compared to the more structured development processes typically seen in benign apps.

Finding 6: *Malware incompatibilities exhibit systematically stronger correlations with version-related factors compared to benign apps, particularly for factors such as API lapse and* minSdkVersion*. While framework compatibility issues remain similarly dispersed across both app groups, hardware compatibility issues show more concentrated patterns in malware.*

5 STUDY II: RUN-TIME INCOMPATIBILITIES

We present our empirical findings on run-time app incompatibilities, following a similar structure to our study on installation-time incompatibilities (Section 4). Per our qualification of run-time compatibility issues, two categories of such issues are anticipated in general: *incompatible launch* and *incompatible running*. Thus, we initially intended to study both categories. However, our analysis revealed that none of the apps in our datasets were associated with incompatible launch issues. Therefore, in the subsequent sections, we focus exclusively on the second category of run-time incompatibilities—–specifically, incompatible running.

5.1 RQ4: Prevalence of Run-Time App Incompatibilities

In a similar visual format to Figure 2, Figure 5 showcases the overall RIR (Run-time Incompatible Rate) of our benchmark apps across each yearly dataset and the combined RIR of all benchmarks (referred to as "all of 12 years"). The numbers in this combined group are the weighted averages of per-year RIRs across all the 12 years studied for each platform version (API level), where the weight for each year is the number of benign/malicious apps we sampled from that year as used for run-time incompatibility study (see Table 1).

Common characteristics. A notable pattern emerges across both benign and malicious apps: API level 26 consistently exhibits high RIRs, ranging from approximately 40–63% for benign apps and 32–52% for malware in recent years. This persistent pattern highlights the challenges posed by API 26, likely due to significant changes introduced in this

Fig. 5. Run-time incompatibilities in terms of RIR in benign apps (y axis, top) and malicious apps (y axis, bottom) over the twelve studied years of 2010 through 2021 and the combined results for all of the 12 years (x axis).

version, such as mandatory background execution limits and stricter permission controls, which require substantial app adjustments. In contrast, newer APIs (28 and 29) demonstrate significantly lower RIRs, typically below 5%. This stark improvement underscores substantial advancements in handling run-time compatibility issues in the latest Android versions, reflecting both platform maturity and developer adaptation.

Older APIs, such as API 19 and API 23, also display distinctive patterns. API 19, as the oldest version studied, consistently shows elevated RIRs, reaching a peak of 70.41% for benign apps in 2016. API 23, which introduced the runtime permission model, presents similarly high RIRs, such as 67.71% for benign apps in 2017. These high rates highlight the significant compatibility challenges associated with major platform changes, particularly during their initial years of adoption [20].

A clear temporal trend is evident: apps from 2014–2017 exhibit notably higher RIRs (often exceeding 50% on specific API levels in benign apps), compared to more recent years (2018–2021), where RIRs generally remain below 30%, with the exception of API levels 26 and 27. This temporal decline indicates that newer apps have benefited from improved development practices, enhanced testing tools, and better alignment with evolving platform requirements. These findings suggest that developers have adapted to platform changes more effectively over time, leading to fewer compatibility issues at runtime.

Interestingly, the explicit specification of higher *minSdkVersion* in recent years (2018–2021), while increasing installation-time compatibility issues as discussed earlier, appears to have brought significant benefits for run-time compatibility. This is particularly evident for newer API levels (28 and 29), where RIRs consistently remain below 5%. This indicates that developers' deliberate attempts to align their apps with newer platform features and requirements [74] have successfully reduced run-time issues, even at the cost of introducing installation barriers.

Finding 7: *Run-time incompatibilities show distinct patterns across API levels: older versions (19-23) exhibit high RIRs (up to 70% in some years), while the newest versions (28-29) maintain consistently low rates (below 5%). API 26 stands out with persistently high RIRs (40-63%) across all years, while temporal trends show decreasing incompatibilities in apps from 2018-2021 compared to earlier years.*

Benign apps versus malware. While the general pattern of RIR variation across API levels shows similarities between the two app groups, notable distinctions emerge in their overall incompatibility rates and specific API-level behaviors. On average across all studied years, benign apps exhibit a higher overall RIR of 20.77% compared to malware's 14.48%. This disparity is particularly evident during specific periods—most notably from 2013–2017, when benign apps consistently showed elevated RIRs above 23% (peaking at 28.85%), while malware maintained comparatively lower rates between 11–15%. These differences suggest that benign apps faced greater challenges adapting to evolving platform requirements during this time, potentially due to a wider variety of app features and use cases.

API-specific patterns further highlight these distinctions. For benign apps, run-time incompatibilities are particularly pronounced for API 26, where RIRs reached up to 63.60% in 2021, and API 23, which peaked at 67.71% in 2017. These peaks reflect significant challenges introduced by the corresponding platform changes—–API 26's introduction of background execution limits and stricter permission handling, and API 23's implementation of the run-time permission model [20]. These platform changes likely required developers to make substantial adjustments, contributing to the higher RIRs observed in benign apps.

In contrast, malware exhibits a slightly different pattern. While malware also shows elevated RIRs for API 26 (ranging from 40–52%), its rates are generally lower across other API levels. This includes notably better compatibility on newer APIs (28–29), where RIRs remained consistently below 7%. This improved compatibility in newer APIs may indicate that malware authors are increasingly optimizing their apps to align with modern platform requirements, potentially to evade detection or improve installation success across a broader range of devices.

Interestingly, benign apps demonstrate a higher sensitivity to platform changes, with larger fluctuations in RIRs across API levels and over time. Malware, on the other hand, shows more stable patterns, suggesting a more deliberate focus on maintaining compatibility with specific platform configurations. This difference may stem from the fact that benign apps often integrate a wider variety of features and dependencies, leading to more complex compatibility challenges. Malware, by contrast, tends to focus on fewer, more targeted functionalities, enabling better alignment with platform requirements.

Another key observation is the declining RIRs for both app groups in newer APIs (28 and 29). For benign apps, this reflects improved development practices and better adaptation to platform updates. For malware, this trend highlights an evolution in construction practices, where compatibility optimization is likely becoming a priority for maximizing reach and avoiding runtime issues that could expose malicious behavior.

To understand this persistent difference, we looked into the run-time behaviors of these studied benign apps versus malware from the perspective of code structure in terms of calling relationships, especially those among the three typical code layers of an Android app: user code, SDK, and third-party libraries [17]. We found that the studied (1) benign apps generally have much higher percentages of calls within the SDK (i.e., calls to one SDK API from another SDK API) than malware, whereas (2) the malicious apps generally have much lower percentages of calls from third-party libraries (noted as *3rdLib*) to SDK. These findings are highly consistent with those from an earlier study comparing the run-time behaviors between benign apps and malware from the same code-structure/cross-layer-calls perspectives [10], where the differences between the two app groups in terms of (1) and (2) were both found statistically significant (in terms of p values) and large (in terms of effect sizes), although we did not use the same dataset as in that prior study.

Here the SDK-to-SDK calls represent the behaviors of the Android framework itself, while the 3rdLib-to-SDK calls represent the behaviors of third-party libraries with respect to their use of the framework capabilities[13]. Thus, the

[13]Note that Android apps, benign or malicious, are known as heavily framework-based [10, 17] in general, meaning that in both app groups the majority of app functionalities are provided by the framework, rather than being developed by the app developers specifically for the apps.

higher RIR in benign apps suggests that run-time incompatibilities seen by Android apps may be considerably attributed to the incompatibility between the Android framework and the underlying Android device [68–70]. This is consistent with the results from our study on installation-time incompatibilities (i.e., Section 4), where we discovered that the primary reason for the studied apps failing in that first line of compatibility was their dependency on native libraries that are incompatible with the (CPU) architecture of the Android device that the apps run on.

Meanwhile, our closer examination also revealed that (3) the studied benign apps have notably higher percentages of calls from user code to SDK. Keeping this in mind while putting together the higher RIR of benign apps than malware and their difference in terms of (2), we see that the various third-party libraries used in Android apps seem to have done a better job than the app-specific user code in using the Android SDK APIs in a compatible manner. And this further difference in terms of (3) between the two app groups provides another point of explanation for the higher RIRs seen by the benign apps than those by the malware among our studied apps. In fact, a number of prior peer studies have found that incompatible use of SDK APIs is a major reason for widespread compatibility issues in Android apps, referred to as *API/evolution-induced incompatibilities* [40, 43, 47, 50, 53, 60, 61, 72].

> **Finding 8:** *Once successfully installed, malware shows lower overall run-time incompatibilities than benign apps (14.48% versus 20.77% on average), with particularly better compatibility on newer APIs (28-29). While both groups exhibit high RIRs on API 26 (40-63%), malware maintains more consistent failures rates across API levels. These contrasts may be explained by the run-time behavior differences between the two app groups.*

5.2 RQ5: Distribution of Run-Time App Incompatibility Symptoms

Figure 6 illustrates the distribution of app execution failures across various run-time incompatibility symptoms. These symptoms are represented by 11 keywords (as listed in the legend), which are the most frequently appeared patterns in the traces of failed app executions. Similarly to how the aggregated distribution of installation-time incompatibility symptoms for each app year (that encompasses all Android versions) is presented in Figure 3, here we show the aggregated distribution of run-time incompatibility symptoms for each app year over all the Android versions given the similarity of per-version distributions.

Common characteristics. For both benign and malicious apps, the three dominant symptoms across all app years were *verify error*, *null pointer* (dereference), and *native crash*, listed in descending order of dominance. Despite fluctuations in their relative proportions, these symptoms consistently remained significant over the years.

The prevalence of *verify error* strongly suggests that SDK/API changes were likely the primary cause of these incompatibilities. Apps compiled against older SDKs and executed on newer ones often experience inconsistencies in bytecode verification, leading to *verify error*. Such inconsistencies are a major cause of *verify errors* [36, 63], especially when platform changes introduce stricter validation rules.

Null pointer exceptions, in this context, often appear as derivative symptoms resulting from underlying issues caused by *verify error* or *native crash*. These compatibility-induced run-time conditions cascade into unexpected null dereferences. Therefore, SDK/API changes are considered the primary cause of run-time incompatibilities in both benign apps and malware.

Native crash errors, which are attributed to issues in the Android native (C/C++) code layer, further compound these compatibility challenges. Bugs in the Android Support Library [62] and limited API support––less than 23% of newly introduced APIs were supported [40]––are notable contributors to this symptom. Although intended to alleviate

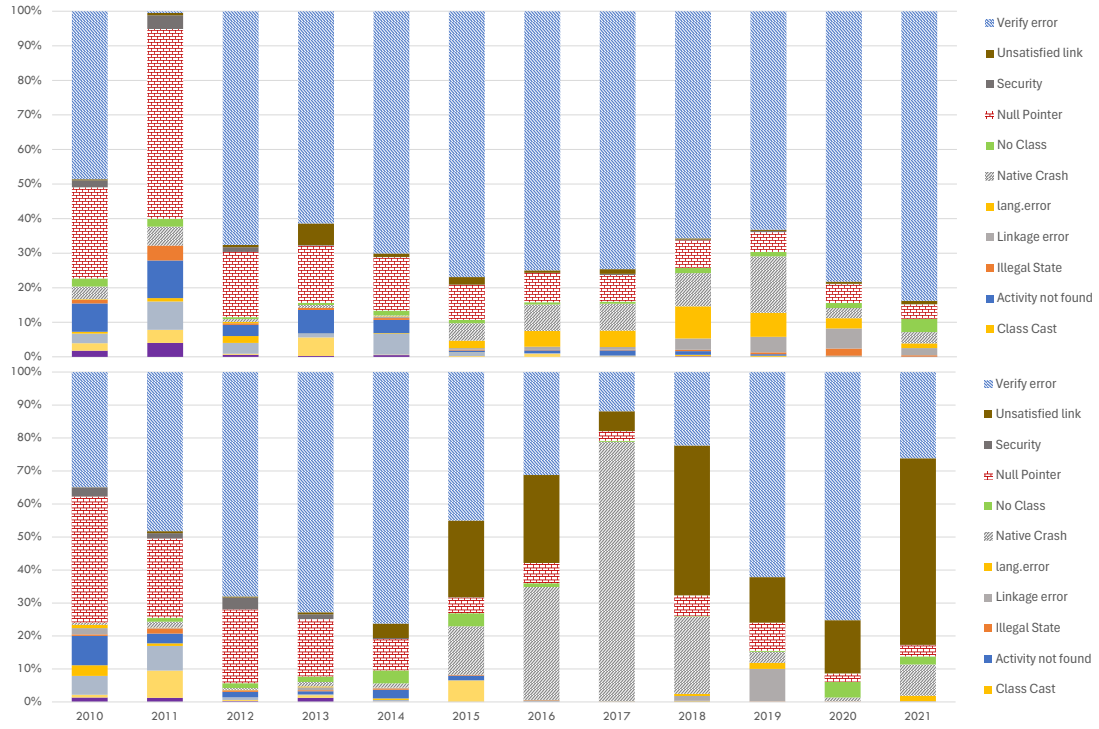

Fig. 6. Percentage distribution of run-time incompatible benign apps (y axis, top) and malicious apps (y axis, bottom) across the ten studied years of 2010 through 2021 over the varied symptoms they exhibited as observed in their run-time failures.

compatibility issues, the Support Library often became a source of additional run-time failures due to its incomplete API coverage.

Beyond these prominent symptoms, two additional observations are common to both app groups. First, for apps of any given year, a variety of other symptoms are notably represented. These include *Unsatisfied link* errors––caused when the definition of a native method is missing; *Null pointer* exceptions; *lang.error* (i.e., `java.lang.Error`, representing serious and unrecoverable runtime errors); and *Activity not found*––which occurs when a call to `startActivity` is invoked against a nonexistent Activity component. While these symptoms are not as prominent as the top three, they collectively form a significant portion of the run-time error landscape.

Second, despite the variety of symptoms observed, there was no consistent pattern in how their proportions among the eleven main categories evolved over the twelve years of this study. Both malware and benign apps exhibited fluctuations in symptom distributions without any clear trends. This indicates that while compatibility issues remain a persistent challenge, their specific manifestations vary depending on the interplay of app characteristics, API changes, and platform evolution.

Finding 9: *The predominant symptoms of run-time incompatibilities that were common to both benign and malicious apps were verify errors (on average 56%) and null pointer (on average 14%). These symptoms were primarily caused by changes in the SDK/API during the evolution of Android. Meanwhile, there were various other symptoms almost as prevalent overall but not as commonly predominant in both app groups, and there were also no consistent common patterns of changes in these symptoms' strength of presence over time.*

Benign apps versus malware. While both app groups share the three prominent symptoms of *verify error*, *null pointer*, and *native crash*, benign apps exhibit a greater prevalence of these symptoms overall. Specifically, the average percentage of the *verify error* symptom over the 12 years was 63.8% for benign apps, compared to 47.9% for malware. Similarly, *null pointer* errors occurred at an average rate of 15.2% in benign apps versus 12.2% in malware. These findings highlight that benign apps are more susceptible to run-time compatibility challenges related to API evolution.

Also, benign apps and malware as studied have exhibited different dominating symptoms of run-time incompatibilities beyond the three shared ones. Notably, malware exhibited a significantly stronger presence of the *Unsatisfied link* error, with an average ratio of 16.14% over 12 years, compared to just 1.3% for benign apps. This stark contrast underscores a fundamental difference in the underlying causes of run-time failures between the two app groups.

The causes of these symptoms reveal distinct patterns of incompatibility. For benign apps, run-time incompatibilities often stemmed from evolution-induced API misuses, such as the use of deprecated or older APIs on newer, incompatible platforms. This can be attributed to a well-documented phenomenon in the Android development community, where developers tend to lag behind in updating their apps to align with the latest Android platforms and SDKs [54, 66]. As a result, benign apps are more likely to encounter compatibility challenges when executed on newer devices.

In contrast, malware's run-time failures were more often caused by issues related to undefined native code or severe and unrecoverable problems within the Android runtime. This difference can be explained by the frequent use of repackaging in malware development. Repackaging, a common technique used to alter legitimate apps for malicious purposes, is prone to introducing semantic corruption, such as missing the definition of native methods called by the app. These errors lead to a higher prevalence of *Unsatisfied link* errors and other critical run-time failures in malware.

Furthermore, malware's reliance on repackaging also explains why its run-time errors are more concentrated around severe system-level issues, whereas benign apps display a broader spectrum of errors related to API evolution and compatibility. This distinction between the two app groups highlights the need for tailored approaches to addressing run-time incompatibilities: improving API migration support for benign apps and enhancing detection mechanisms for semantic corruption in malware.

Finding 10: *In comparison between the two app groups, run-time incompatibilities in benign apps are more often due to API evolution and usage of outdated/incompatible APIs, while run-time incompatibilities in malware are more often due to the problematic/error-prone construction practice in their creation.*

5.3 RQ6: Contributing Factors of Run-Time App Incompatibilities

Similar to Study I, we analyzed the distribution of RIR across the same possible contributing factors. Figure 7 presents the distribution based on *app lapse* and *API lapse*, both overall and separately for the top two symptoms common between benign apps and malware. The plot, in a format same as Figure 4, consists of data points (squares) each representing the RIR of all apps with specific app lapse and API lapse values, regardless of their creation years.

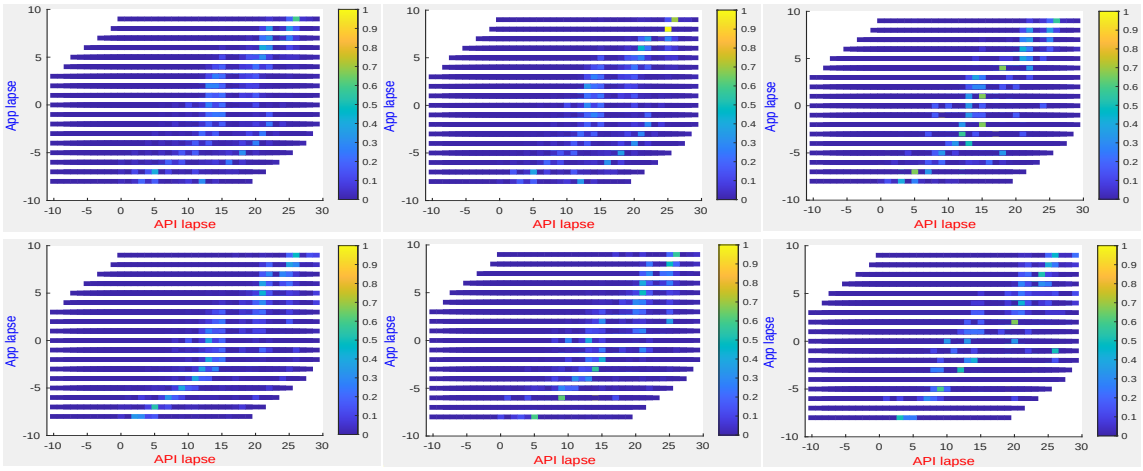

Fig. 7. Distribution of RIR (encoded by square color) over *app lapse* (y axis) and *API lapse* (x axis) between benign (top) versus malicious (bottom) apps, for all symptoms (leftmost) and the two dominating ones that are common to two app groups: `verify error` (middle) and `null pointer` (right).

Common characteristics. In the overall distribution (left), the RIRs were generally lower compared to the per-symptom distributions. This is because the overall distribution includes all our benchmarks regardless of their app lapses and API lapses. The two per-symptom distributions, on the other hand, focus only on apps that experienced run-time incompatibilities during execution (i.e., with RIRs greater than zero).

The distribution patterns reveal a complex relationship between run-time compatibility issues and version differences measured via the two lapse metrics. Figure 7 shows scattered hotspots of higher RIRs appearing in diagonal patterns across the version space, suggesting that run-time failures are influenced by the combined effect of chronological and SDK version gaps rather than either factor alone.

Examining the symptom-specific distributions, verify errors (middle) show more concentrated patterns of failures, while null pointer errors (right) exhibit a more dispersed distribution. This difference suggests that verification-related compatibility issues are more predictable based on version relationships, while null pointer errors may arise from more complex interactions between app implementation and platform evolution. For these null pointer errors, the dispersed distribution aligns with the varied and often complex interactions between app-specific implementations and evolving platform behaviors.

A notable pattern emerges where higher RIRs tend to cluster around moderate API lapse values (10–20), particularly when combined with certain app lapse ranges. This indicates that run-time compatibility issues are most prevalent when there's a moderate gap between the app's minimum SDK requirements and the platform version, rather than at extreme version differences. Interestingly, the diagonal distribution of hotspots suggests that compatibility issues are more likely to occur when the sum of app lapse and API lapse reaches certain thresholds. This implies that the total version distance, combining both chronological development timing and SDK version differences, may be a better predictor of run-time compatibility issues than either measure alone.

These patterns indicate that Android's compatibility mechanisms operate most effectively within certain bounds of combined version differences, beyond which *both forward and backward compatibility become less reliable*. This

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

App group	Symptom	app lapse	API lapse	minSdkVersion	SDK API level	app year
Benign	overall	-0.03	**0.41**	**-0.43**	-0.01	0.03
	null pointer	0.07	0.25	-0.28	-0.07	-0.10
	verify error	-0.08	**0.36**	**-0.38**	-0.01	0.08
Malware	overall	-0.06	**0.45**	**-0.44**	0.08	0.10
	null-pointer	-0.01	**0.40**	**-0.38**	0.12	0.07
	verify error	-0.01	**0.40**	**-0.38**	0.12	0.06

challenges the traditional notion of straightforward backward compatibility and suggests a more nuanced reality where successful run-time behavior depends on the interplay between development timing and API evolution.

The visual patterns indicating correlations between RIR and app/API lapses were largely supported by the Spearman correlation coefficients presented in Table 3. Overall, both API lapse and minSdkVersion demonstrate significant correlations (of moderate strength) with RIR for both benign apps (0.41, -0.43) and malware (0.45, -0.44), suggesting these version-related factors are key determinants of run-time compatibility regardless of app type.

For both groups, the correlations maintain similar patterns across different symptoms. Verify errors show moderate correlations with API lapse (0.36–0.40) and *minSdkVersion* (-0.38), while null pointer errors exhibit slightly weaker but consistent relationships (0.25–0.40 for API lapse and -0.28 to -0.38 for *minSdkVersion*). The uniformity of these correlations suggests common underlying mechanisms affecting run-time compatibility across all apps, likely tied to how SDK/API updates interact with app behavior at runtime.

Notably, both app groups show consistently weak correlations (below 0.1) between RIR and app lapse, SDK API level, or app year. This indicates that chronological factors, such as the timing of app development or the release year of the SDK, have relatively minor direct influence on run-time compatibility issues, regardless of whether the app is benign or malicious. Instead, the correlations highlight the importance of the gap between minimum and actual (hosting device's) SDK versions (API lapse) and the targeting choices made by developers (*minSdkVersion*) as primary factors influencing run-time compatibilities.

These statistical results reinforce that run-time compatibility in Android is primarily governed by SDK version differences rather than temporal development factors, a characteristic that holds true across both benign and malicious applications. The strong influence of API lapse and *minSdkVersion* suggests that developers can mitigate run-time compatibility issues by carefully aligning their apps' minimum SDK requirements with the expected platform versions, regardless of the app type (i.e., app group).

Finding 11: *Run-time compatibility issues in both benign apps and malware show significant correlations with API lapse and minSdkVersion, while chronological factors such as app lapse show minor influence—suggesting that SDK version gaps, not development timing, are the primary determinants of run-time compatibility issues.*

Benign apps versus malware. While both app groups show moderate correlations between run-time compatibility and version-related factors, malware exhibits more consistent and slightly stronger relationships, as visually observable from Figure 7. This reflects the structured and systematic impact of version differences on malware compared to the broader and more varied compatibility challenges seen in benign apps.

This difference is even clearer in Table 3, where for malware, both verify errors and null pointer errors show identical correlations with API lapse (0.40) and minSdkVersion (-0.38). In contrast, benign apps exhibit more varied correlations across different symptoms, with API lapse ranging from 0.25 to 0.36 and minSdkVersion ranging from -0.28 to -0.38. This variation indicates that benign apps are influenced by a wider range of factors beyond strict version differences, potentially due to their more diverse feature sets and API usage patterns.

The overall correlation pattern is stronger in malware (API lapse: 0.45, minSdkVersion: -0.44) compared to benign apps (API lapse: 0.41, minSdkVersion: -0.43), suggesting that run-time compatibility in malicious apps is more systematically tied to version differences. This systematic relationship likely reflects differences in development practices or constraints in malware creation, such as the reliance on repackaging or targeting specific platform versions for exploitability.

Looking at the heatmap distributions in Figure 7, malware shows more concentrated patterns of high RIR values, while benign apps exhibit more dispersed patterns across minimal SDK and platform version combinations. This concentrated pattern suggests that run-time compatibility issues in malware are more predictable based on such version relationships, as malicious apps may target a narrower range of platform configurations to ensure functionality. Conversely, the more dispersed patterns in benign apps reflect a greater variety of run-time challenges, likely stemming from their broader API usage and feature implementations.

Additionally, the uniform correlations in malware highlight its dependence on specific platform features or APIs, making it particularly sensitive to version-related discrepancies. On the other hand, benign apps, with their varied correlations and broader heatmap distributions, encounter compatibility challenges that are influenced by a mix of platform evolution and complex app-specific factors.

Further analysis of the diagonal patterns reveals that the hotspots (high RIR areas) are more prominent for benign apps compared to malware. For benign apps, the concentrated hotspots in the verify error distribution suggest that compatibility mechanisms introduced in certain API levels (e.g., API 26) are more likely to trigger verification failures when used with apps developed significantly earlier. In contrast, malware hotspots are comparatively sparser and exhibit smaller RIR clusters for verify errors, potentially due to more targeted API usage or limited feature sets in malicious apps. For null pointer errors, the broader spread of null pointer errors in benign apps could indicate their reliance on a more diverse range of APIs, making them susceptible to unexpected runtime behaviors as platform features evolve. Conversely, malware's narrower spread suggests a focus on a smaller subset of API features, reducing its exposure to such compatibility challenges.

Finding 12: *Malware demonstrates more consistent and stronger correlations between incompatibility-induced run-time failures and version-related factors compared to benign apps. While both groups show moderate correlations, malware exhibits identical patterns across different types of failures, suggesting more systematic relationships between SDK version differences and compatibility issues in malicious apps.*

6 STUDY III: INCOMPATIBILITIES ON LATEST ANDROID VERSIONS

To ensure our findings remain relevant with the latest Android platform versions, we conducted an additional study examining compatibility patterns on Android 14 (API 34) [1] and Android 15 (API 35) [4]—the two latest Android API levels by the end of year 2024. From each of the 12 yearly (2010–2021) datasets described earlier, we randomly selected apps until 50 can be successfully installed hence used for run-time compatibility testing on each of these two additional target API levels. This sampling approach mirrors that adopted for the main study while providing a reasonable number

of data points to enable observing compatibility patterns. We also follow the same methodology to characterize the installation- and run-time compatibilities of these sampled apps.

During our experimentation, we encountered a significant platform policy change [2] starting with API 34. Beginning with this version, Google has implemented stricter installation requirements that mandate a minimum *targetSdkVersion*. Specifically, API 34 requires apps to target no lower than SDK version 23, while API 35 further raises this requirement to target SDK version 24 or higher. These policy changes directly impacted our installation testing results. For API 34, we observed that all apps developed before 2016 consistently failed to install, resulting in an IIR of 100% for these older apps. The installation failures uniformly produced the symptom `INSTALL_FAILED_DEPRECATED_SDK_VERSION: App package must target at least SDK version 23, but found` x, where x represents the app's actual *targetSdkVersion*. This outcome aligns with historical context, as API 23 was released in late September 2015, making it unlikely for apps developed before 2016 to target this version. Similarly, for API 35, all apps created before 2017 failed installation with the symptom `INSTALL_FAILED_DEPRECATED_SDK_VERSION: App package must target at least SDK version 24, but found` x. This again corresponds with the timeline of API 24's release in August 2016, meaning most pre-2017 apps would not have targeted this API level.

However, `adb` also provides an option `bypass-low-target-sdk-block` that allows circumventing these restrictions. This enabled us to proceed with our compatibility study by installing apps that would otherwise be blocked due to their lower target SDK versions. By utilizing this bypass option, we were able to complete our installation tests and proceed with run-time compatibility analysis, ensuring continuity with our methodology from the main study.

After examining compatibility patterns on API 34 and API 35, we found both consistencies with our prior findings and several noteworthy new patterns. These observations provide valuable insights into how compatibility issues evolve with the latest Android platforms. Next, we present main results in our additional analysis.

Installation-Time Incompatibility. As seen in Figure 8, for installation-time incompatibility, malware continues to exhibit higher incompatibility rates than benign apps across both API 34 and 35, with average IIRs of 30.3% and 30.5% for malware compared to 21.7% and 21.3% for benign apps, respectively. This *reinforces our earlier finding (Finding 2)* that malicious apps generally face greater installation barriers than legitimate applications.

Fig. 8. Installation-time incompatibilities in terms of IIR in benign apps (y axis, top) and malicious apps (y axis, bottom) over the twelve studied years of 2010 through 2021 and the combined results for all of the 12 years (x axis).

Meanwhile, we observed a dramatic shift for apps developed in 2020, which showed near-zero IIRs (below 1%) on both API 34 and 35. This suggests that recent app development practices have significantly improved alignment with newer platform requirements, likely due to developers' increased awareness of compatibility requirements and Google's stricter enforcement policies. However, malware from 2021 exhibited substantially higher IIRs (54% on API 34 and 53% on API 35) compared to benign apps from the same year (1% on both APIs). This growing disparity suggests that while legitimate developers have adapted to platform evolution requirements, malware creators may be deliberately using outdated development approaches or repackaging techniques that result in greater incompatibilities with newer Android versions. Despite these trends, the dominant failure symptom across both app groups remains `INSTALL_FAILED_NO_MATCHING_ABIS`, *consistent with our Finding 3*. This suggests that even with platform policy changes mandating minimum target SDK versions, hardware architecture mismatches continue to be the primary barrier to installation compatibility.

Run-Time Incompatibility. As Figure 9 shows, for run-time incompatibility, we again observed that malware experiences higher incompatibility rates than benign apps, with average RIRs of 32.7% and 33.9% for malware versus 28.3% and 24.8% for benign apps on API 34 and 35, respectively. This further *validates Finding 8*, indicating that even successfully installed malware faces distinct execution challenges compared to benign apps.

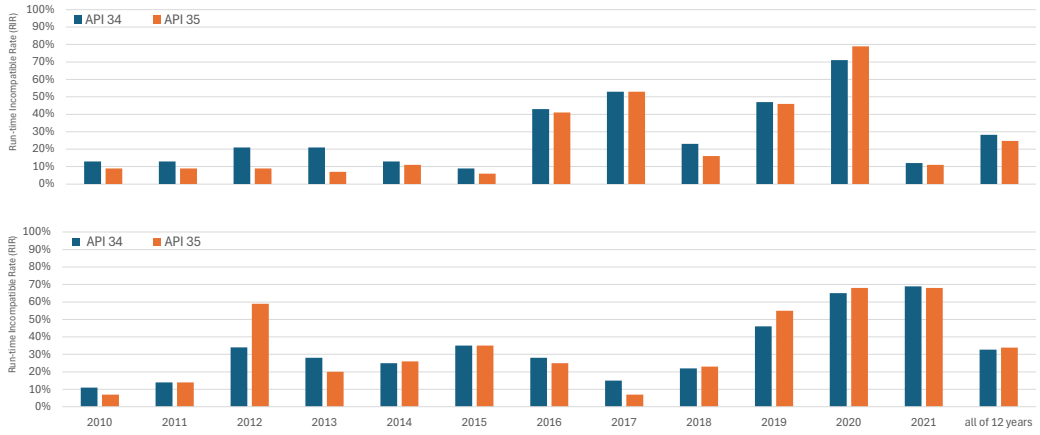

Fig. 9. Run-time incompatibilities in terms of RIR in benign apps (y axis, top) and malicious apps (y axis, bottom) over the twelve studied years of 2010 through 2021 and the combined results for all of the 12 years (x axis).

A notable temporal pattern emerged in run-time compatibility for both app groups. Benign apps show a steady increase in RIRs from 2018 (23% and 16%) to 2020 (71% and 79%), followed by a substantial decrease in 2021 (12% and 11%). This pattern suggests that while 2018-2020 apps encountered significant run-time challenges on newer platforms, apps developed in 2021 have considerably better run-time compatibility, reflecting improved adaptation to platform evolution. Malware exhibits an even more pronounced upward trajectory in RIRs from 2018 through 2021, reaching alarmingly high values of 69% and 68% by 2021. This consistent increase suggests that malicious apps face increasing execution barriers on newer Android versions, potentially due to enhanced security measures in these platforms or incompatible exploitation techniques. On the other hand, the dominant run-time failure symptom across both APIs and app groups remains `java.lang.VerifyError`, *confirming our Finding 9* that SDK/API changes during Android evolution continue to be the primary cause of run-time incompatibilities.

These observations on API 34 and 35 largely reinforce our earlier findings while revealing new evolutionary patterns in Android compatibility. They confirm that both installation-time and run-time compatibility issues persist in the latest Android versions, with distinct manifestations between benign apps and malware. Importantly, these additional study results suggest that conclusions derived from our original dataset (covering up to 2021 apps and API 29) might sustain their relevance as newer Android versions continue to be released.

> **Finding 13:** *Testing on the latest Android versions (API 34 and 35) confirms our previous findings while revealing new patterns. Both platforms enforce stricter installation requirements that block apps with lower target SDK versions, yet they maintain similar incompatibility trends.*

7 THREATS TO VALIDITY

Internal validity threats. A potential threat to the *internal validity* of our study results is the presence of errors in the implementation of our experimentation utilities, including tools and scripts. To mitigate this threat, we took several measures. First, we conducted a thorough code review of our own toolkit and scripts. We ensured that they were implemented correctly and aligned with our experimental design. Additionally, we manually verified the functional correctness of these utilities through selected benchmark validations. It is important to note that other tools utilized in our study are widely used within the Android SDK by both researchers and developers, which adds to their reliability.

Another potential internal threat relates to test flakiness in Android app executions. Despite using fixed seeds for test inputs and running each app across multiple Android versions, the generally possible non-deterministic nature of mobile app executions implies that even the same test inputs might trigger different behaviors across runs on multiple SDK versions. While our methodology of requiring successful execution on at least one version helps mitigate misclassification of flaky failures as compatibility issues, we cannot completely eliminate the impact of test flakiness on our results. Our approach of running each app on 10 different SDK versions provides some robustness against flakiness as we did not observe such cases happen during our manual verification, but this remains a potential source of inaccuracy of our methodology for determining incompatibility-induced installation/run-time failures in general.

External validity threats. The main concern regarding the *external validity* of our study results relates to the selection of benchmark apps. Despite our efforts to utilize a substantial collection of apps from diverse sources, the vast number of Android apps available on various app markets means that our selection represents only a relatively small subset. Consequently, the apps studied from each of the 12 years may not provide a fully representative sample of all Android apps during those years. As a result, our findings and conclusions are limited to the specific apps included in our study. This concern is particularly pronounced in our extended analysis on API 34 and 35, where we sampled only 50 apps per year. This limited sample size for the newest Android versions may not fully capture the diversity of compatibility issues in the broader app ecosystem, potentially restricting the generalizability of our observations about compatibility patterns on these latest platforms.

The effectiveness and quality of dynamic analysis results are influenced by the coverage of run-time inputs. In our study, the dynamic analysis approach used was relatively simple, and the identification of run-time incompatibility effects relied on the execution of app behaviors during a five-minute runtime. However, there is a possibility that the random inputs generated by Monkey might have missed certain execution paths and, consequently, specific incompatibility effects of the apps. Furthermore, although we maintained consistent test inputs for each app across the ten runs (each corresponding to a different Android version), the coverage of the app's behavior may have varied due

to changes in the Android platform across different versions. This introduces a threat to the *external validity* of our study results concerning run-time incompatibilities.

Another concern regarding *external validity* relates to the multi-APK phenomenon in Android. Developers often upload multiple APKs of an app to support different devices, and Google Play ensures that users receive the appropriate APK based on their device characteristics such as hardware, vendor, and Android version. To preliminarily assess the impact of this phenomenon on our results, we randomly selected 10 apps that were identified as incompatible with API 21 in our study. We attempted to manually install and run these apps on a real Samsung Galaxy S4 phone (API 21) directly from Google Play. Our findings indicate that all 10 apps remained incompatible, suggesting that multiple APKs may not exist for these particular apps. However, a comprehensive examination of all our benchmarks would be necessary to thoroughly evaluate the impact of the multi-APK phenomenon. Additionally, we did not consider the possibility that apps included in our benchmarks, which were identified as incompatible, may have received updated versions that resolved the compatibility issues at a later time. These factors may contribute to a potential overestimation of incompatibilities in our study.

Construct validity threats. The primary concern regarding *construct validity* pertains to the metrics and measurement procedures employed to assess the extent and distribution of app incompatibilities. It is possible that there are other measures and metrics that we did not consider, which might have better supported or further substantiated our conclusions. To mitigate this threat, we employed a diverse set of measures to characterize compatibility issues in Android apps from various perspectives. We also employed different measurement approaches, including group statistics to understand overall characteristics (e.g., aggregate RIR) and statistics of yearly subsets to uncover relevant evolutionary patterns. Additionally, our study did not specifically address app incompatibilities in relation to the *targetSdkVersion* specified in apps (e.g., IIR of apps on platforms with API levels equal to, smaller than, or greater than the app's *targetSdkVersion*). Exploring this aspect could potentially impact our overall conclusions regarding installation-time incompatibilities, particularly those related to Android's forward and backward compatibility. Furthermore, it's important to note that the ten Android Virtual Devices (AVDs) used in our studies were all based on x86/x86_64 processors. However, some of our benchmarks may have been developed for hence meant to be used on ARM architectures. In such cases, the installation incompatibilities observed, indicated by the "INSTALL_FAILED_NO_MATCHING_ABIS" errors, could be attributed to the apps' architecture preferences. Running these benchmarks on devices with ARM processors could yield different IIR results, potentially leading to changes in our current conclusions based on such results.

Additionally, our use of the `-bypass-low-target-sdk-block` option in `adb` to circumvent installation restrictions on API 34 and 35 also introduces a potential threat. This approach allowed us to study apps that would otherwise be blocked by Android's security mechanisms, creating an artificial testing environment that may not reflect real-world usage scenarios—ordinary users (installing apps via GUI operations) may not bypass these restrictions.

Conclusion validity threats. The heavy overhead of our study necessitated certain limitations. We focused on a specific set of hardware configuration parameters for all ten devices corresponding to the Android versions under investigation. Consequently, there is a potential threat to the *conclusion validity* of our results, as they may not generalize to all possible hardware configurations and API levels of Android devices in use. To address this concern, we selected the most prevalent API levels utilized by various Android devices with respect to the time range of our subject app collection. However, it is important to note that our conclusions should primarily be interpreted within the context of the device API levels and configurations employed in our study. Considering the dominant symptoms and underlying causes of the observed incompatibilities, as well as the case study conducted on the Samsung phone, it appears that the

incompatibilities are attributed to the broader Android ecosystem rather than the specific Nexus One device utilized. Consequently, we anticipate that our results would yield similar findings if the study were conducted on a different device, such as a Samsung phone. Nevertheless, to achieve more comprehensive and conclusive results, experimentation with multiple hardware devices would be necessary.

8 DISCUSSIONS

Building upon our empirical findings presented earlier, we have derived additional insights into installation-time and run-time app compatibility issues in Android. We aim to discuss the implications of our results, lessons we learned from this research, and provide practical recommendations to both app developers and end users on how to effectively address respective problems and challenges.

In particular, we start with discussing how to deal with benign-app incompatibilities according to our study (Section 8.1). These discussions are mainly based on the results on benign apps—we do not intend to make malware more compatible; in fact, if an app is detected as malware, it would not be attempted for installation or execution hence making installation-time and run-time compatibility issues irrelevant. Then, drawing on our comparative results between benign apps and malware, as well as the unique characteristics of malware with respect to our study results, we also discuss how the security property may be relevant to app incompatibilities (Section 8.2). These discussions are mainly poised to help/inform defenses against malware in Android.

8.1 Dealing with Benign-App Incompatibilities

Explicitly attempting for installation-time compatibility. Regarding installation-time compatibility issues, our findings suggest that these issues seem to be closely tied to the age of an app (see Figure 2). However, the actually critical factor affecting (strongly correlated with) such incompatibilities is the installation-time compatibility *attempt* of developers via the *minSdkVersion* attribute they specified in the apps. When installing an app on an Android version where the delta between the specified *minSdkVersion* and the platform API level is within 7 to 10, compatibility concerns at installation time are relatively minor. However, if the gap exceeds 12, there is an increased risk of installation failure.

Thus, to ensure successful installation of an app on a device, developers should leverage the *explicit* compatibility attempt by exercising caution when specifying the *minSdkVersion* attribute. It is important to strike a balance and ensure that the specified *minSdkVersion* is not too far from the API level of the targeted Android platform. Meanwhile, neglecting to specify this attribute poses a risk, as it defaults to 1, resulting in a potentially large API lapse, particularly with newer Android versions. Thus, the bottom line is to explicitly make that attempt.

Proactively mitigating compatibility-induced installation failures. Per our results in Section 4.2, the most dominating symptom of installation-time incompatibilities was `INSTALL_FAILED_NO_MATCHING_ABIS`. Thus, in cases where installation fails, it is typically due to the app utilizing native functionalities that are not supported by the targeted hardware architectures—particularly the processor (CPU) architectures. App developers need to be aware of the architectural support of any native capabilities that the app depends on. Avoiding or at least reducing such dependencies should be considered, when possible (e.g., by utilizing hardware-architecture-independent alternatives).

Additionally, using libraries that are not available in a vendor-customized Android framework was observed as another cause of installation failures. To mitigate such failures, developers should verify whether the targeted devices support all the app's functionalities in terms of any Android customizations introduced by device vendors, in addition

to the hardware architecture of the devices. This assessment is crucial to ensure (installation-time) compatibility hence a smooth installation process.

Pragmatically, it is recommended that app developers conduct installation testing of their apps on various CPU architectures and against different Android customizations before releasing the apps. From the end users' point of view, it would be very helpful if the developers make clear about results of such testing in the app descriptions, including the specific hardware architectures and customizations that have been tested.

Also, while the Android platform has made improvements in accommodating apps in terms of installation-time compatibility (especially starting with Android 7.0), the general likelihood of encountering installation failures on Android devices did not appear to consistently correlate with the SDK API level of the Android platform running on those devices. That is, older or newer Android platforms did not necessarily exhibit better or worse compatibility with older or newer apps. Thus, it primarily relies on app developers to make explicit and particular efforts to mitigate installation-time compatibility issues, such as checking for hardware-incompatible and vendor-absent functionalities and making clear compatibility attempts.

Cautiously choosing target Android platforms (SDK API levels). Beyond installation-time incompatibilities, run-time compatibility issues also pose a significant challenge across the Android ecosystem, with distinct patterns across different API levels. While newer APIs (28-29) show improved compatibility with consistently low RIRs, API 26 exhibits notably high incompatibility rates, suggesting that careful consideration is needed when targeting different Android versions. Once an app is successfully installed on a device, the specified *minSdkVersion* becomes less of a concern as other factors more significantly impact whether the app can run normally on the device.

Our findings indicate that compatibility between the SDK an app is built against and the SDK it runs on remains crucial, particularly due to API changes. The stark contrast in RIRs between API levels suggests that developers need to be particularly cautious when targeting certain Android versions. Furthermore, apps developed closer to a platform's release year showed different compatibility patterns than those developed several years before or after, indicating that temporal proximity to platform release doesn't necessarily guarantee better compatibility.

Thus, to minimize run-time compatibility issues, developers should consider several strategies: prioritize testing on API 26 due to its notably high incompatibility rates; leverage the better compatibility demonstrated by newer APIs when possible; and thoroughly review API changes between versions. More generally, for reducing run-time issues, minimizing API lapses and aligning the app's *minSdkVersion* with the targeted platform's SDK version is a critical step. There have been a number of techniques/tools that aim to assist developers with this task by identifying the specific APIs that need to be updated [55] and further automatically upgrading/updating the APIs used in given apps [66]. Developers of Android apps should leverage these tools while paying particular attention to SDK versions that are known to have higher incompatibility rates.

Purposely diagnosing run-time failures using error messages/logs. From our study results presented in Section 5.2, run-time incompatibilities are most often symptomized as verify errors and native crashes. We identified these symptoms by actually running the sample apps and observing their execution traces. Moreover, we summarized these symptom codes explicitly from the error messages in the run-time traces and logs.

Thus, when encountering run-time failures, app developers should pay close attention to error messages and logs related to verify errors and native crashes, as these can provide valuable insights into potential compatibility issues. For instance, tracking from where these errors were manifested back to the likely error-originating locations in the apps would help developers understand the specific root causes and identify fixing strategies.

Timely updating Android platforms/SDK APIs on user devices. Our results pointed to changes in the evolution of Android platforms, especially that of the SDKs/APIs, as the underlying reasons for the dominating run-time incompatibility symptoms like verify errors and native crashes. While app developers may take proactive actions to prevent these issues from happening and take postmortem steps to fix such issues during app maintenance and evolution, end users of the apps also have an important role to play in the battle against those compatibility-induced run-time failures.

Specifically, for app users, updating to the latest Android versions (e.g., APIs 28-29) could help reduce compatibility issues, as these more recent versions show consistently lower RIRs. However, users should be aware that certain versions, particularly API 26, demonstrate significantly higher incompatibility rates regardless of app age. This suggests that blindly updating to a newer version may not always improve compatibility. The common bottom line is that end users should make informed decisions about platform updates based on their specific app usage needs. Before updating, users should consider checking their critical apps' compatibility with the target Android version, particularly when updating to known problematic API levels.

8.2 Security Relevance of App Incompatibilities

Platform improvement in app compatibility not forming a concomitant defense. According to our results in Section 4.1, like benign apps, installation-time incompatibilities in malware were mostly independent of the version of the Android platform they are installed to. This observation implies that any of the improvements made in the Android platform/framework for better accommodating app compatibility did not affect much how easily/hard malware can be installed and hence penetrate/propagate successfully. From the overall declining IIRs in the recent (last three) years, such improvements did happen over the years.

While for benign apps these platform-wise improvements are encouraging and beneficial, they are also concerning with respect to the fact that malware also benefits the same from those improvements. Overall, it is clear that the platform improvement in app compatibility has not been forming a concomitant, implicit defense against the dissemination of malicious apps. Even worse, the latest improvements, as noted above, apparently facilitated malware installation hence dissemination. This observation raises significant security concerns about the relationship between app compatibility and app security. To the best of our knowledge, such a security warning has not been raised before. The implications are particularly concerning given that newer Android versions, initially expected to provide better security through improved platform features, may actually propel easier malware distribution.

In response, a practical, multi-layered defense strategy is needed. First, static installation-time malware detection should be strengthened to prevent malware from being installed as a first line of defense, especially for apps targeting newer API levels where malware shows better compatibility. Second, given that malware shows distinctly different compatibility patterns from benign apps (particularly on newer APIs), these patterns could potentially be leveraged as additional signals for malware detection systems.

Repackaging as a two-edged sword. Our results in Section 4.1 also suggest that malware tended to be subject to serious (a substantially higher degree of, than benign apps) installation-time compatibility issues. Our further investigation revealed that this major difference can be attributed to the widespread practice in malware production—repackaging. Indeed, per earlier studies, most (e.g., 80.6% [75]) of the malware in Android is produced through repackaging, especially piggybacking popular benign apps with malicious payload [48]. As we reported in Section 4.2, beyond the use of

hardware/CPU-architecture-incompatible native capabilities as the predominating causes common to both app groups, the next dominant cause of installation-time incompatibilities in malware is invalid APKs—mainly a result of repackaging.

On the one hand, repackaging has become a cheap productive way of developing malware; on the other hand, this practice led to greater installation-time incompatibilities in malware, potentially counteracting the intention (e.g., wide and easy malware propagation) of mass malware production through repackaging. Thus, from a mixed standpoint of app security and app development productivity, repackaging is a two-edged sword for Android malware.

Our results in Section 4.2 further revealed that among the benign apps across the 12 years studied, when an app failed to be installed, the failure symptoms were almost always limited to the two common, dominating ones—`INSTALL_FAILED_NO_MATCHING_ABIS` and `INSTALL_FAILED_MISSING_SHARED_LIBRARY`. In the malware, however, the failure symptoms were much more diverse. From a perspective of defending against malware, if an arbitrary Android app fails at installation time while exhibiting symptoms other than the two common dominating ones, it is more likely to be malicious than benign—a potential assistive decision factor to be considered in developing a static malware detection technique.

Leveraging SDK version difference and compatibility influence factor patterns as security indicators. Our results in Section 5.3 reveal that run-time compatibility issues in malware are more systematically tied to SDK version differences, as evidenced by their consistent and stronger correlations with API lapse and *minSdkVersion*. This more systematic compatibility-related malware behaviors contrast with the more varied patterns observed in benign apps, making SDK version difference patterns a potential distinguishing factor that may be leveraged for identifying malware. For example, apps that exhibit concentrated run-time incompatibility hotspots for moderate API lapse values (10–20) and predictable relationships with *minSdkVersion* may warrant closer inspection. These behaviors align with the narrower compatibility focus of malware, which often targets specific platform configurations to maximize reach or exploitation of security vulnerabilities.

Moreover, distinctive behavioral patterns, such as uniform correlation strengths across multiple symptoms (e.g., *verify errors* and *null pointer errors*), can serve as additional indicators of malicious intent. Unlike benign apps, which tend to display more diverse symptom distributions, malware's more uniform behavior provides an opportunity to detect potential threats based on their compatibility footprint. Similarly, patterns of statistical correlation with various compatibility influence factors may be leveraged as malware detection features too. As shown in Table 3, run-time compatibility symptoms in malware are clearly most uniformly correlated with different influence factors (e.g., API lapse and *minSdkVersion*) than in benign apps.

Strengthening dynamic malware detection. As per our results presented in Section 5.1, benign apps show notably higher RIRs overall than malware. That is, once passing the installation bar, malware in Android tended to be alarmingly easier to run through without any compatibility-induced problems than benign apps. This is very concerning because it implies that once an arbitrary Android app gets successfully installed, it would have a less chance to fail hence more easily cause harms if it is malicious indeed. Therefore, it is vital to deploy malware detection at runtime, despite the well-known challenges with utilizing dynamic malware detection techniques (e.g., the more hassles for setting up, the greater overhead during app usage) [13].

Particularly, malware demonstrates consistently high RIRs on API 26 and better compatibility on newer APIs (28-29), suggesting more reliable execution on modern platforms. This is particularly concerning as it implies that malware, once installed, can more predictably execute its malicious behaviors on certain API levels. Thus, robust dynamic malware detection is especially essential on these platforms where malicious apps face fewer execution barriers. Also, the pattern

of *varying* RIRs across API levels further suggests a need for targeted defense strategies that consider platform-specific (dynamic) detection features.

Moreover, our investigation as discussed in Section 5.1 revealed distinct behavioral patterns between benign apps and malware in terms of their SDK interactions and third-party library usage. First, the benign apps have much higher percentages of calls to SDK functionalities from both within the SDK and user code than malware. Second, the malware has much notably lower percentages of calls to SDK from various third-party libraries than the benign apps. These code structural differences not only explain the compatibility patterns but also offer potential signals for malware detection. Thus, future research on mobile app security should further explore how to leverage these differences, particularly on API levels where malware shows better compatibility, for effective dynamic malware detection.

Paying attention to older apps/platforms for compatibility and newer ones for security. Our results reveal that different API levels present varying levels of compatibility barriers against malware execution. API 26 shows consistently high RIRs for both benign and malicious apps, while newer APIs (28-29) demonstrate better compatibility overall. This suggests, for dealing with incompatibilities in Android apps, a need for attention to the older apps that tend to target older platforms, and that the fixing strategies need to be tailored to specific API levels rather than assuming uniform behavior across platforms.

On other hand, concerning security, particularly noteworthy is the incompatibility pattern on newer APIs where malware maintains more consistent RIRs compared to benign apps. This indicates that malware authors may be specifically targeting these platforms for better reliability. Therefore, enhanced security measures should be deployed on these newer API levels where malware demonstrates more predictable execution patterns. Additionally, special attention should be paid to apps showing unusually good compatibility across multiple API levels, as this might be an indicator of potentially malicious behavior—legitimate apps tend to show more variation in their compatibility patterns across different Android versions.

9 RELATED WORK

Studies on app issues relevant to incompatibilities. In previous research, there has been a notable focus on the compatibility issues in Android apps. To gather empirical evidence regarding the Android fragmentation phenomenon, Han et al. [39] specifically examined bug reports submitted by users of HTC and Motorola devices. Through the application of topic models and topic analysis techniques, they were able to identify evidence related to the Android fragmentation issue. The primary objective of their study was to enhance understanding of the fragmentation problem itself, rather than specifically investigating the compatibility issues that resulted from this problem. In contrast, our study immediately addresses app compatibility issues.

Several previous works investigated crashes of Android apps to understand their causes [22] and to reproduce the crashes [56]. However, it is important to note that the crashes studied in these works were not necessarily tied to app incompatibilities. In our study, we specifically focused on crashes that were caused by compatibility issues in order to characterize app incompatibilities; yet our study is not limited to such issues symptomized as crashes.

Research on API/evolution-induced app incompatibilities. Recent studies [40, 72] specifically examine compatibility issues arising from SDK evolution and API changes. These studies investigate the impact of these changes on compatibility and explore strategies employed by developers to address these issues. In [43], the authors particularly address such kinds of compatibility issues that are caused by the evolution of callback APIs used in Android apps. When the Android framework evolves, those APIs often get updated as well from one framework version to another.

Accordingly, several techniques [47, 53, 69] have been developed to detect these kinds of app compatibility issues by analyzing API changes during the Android framework evolution. Liu et al. developed a dedicated technique for identifying the APIs that cause compatibility issues [51]. Mobilio et al. developed FILO [55], a tool to help app developers diagnose backward compatibility issues due to the evolution of the Android framework. These kinds of techniques have also been empirically evaluated and compared [50, 60, 61]. In addition, instead of focusing on application-level compatibility issues, the approaches proposed in [19, 44] target application libraries in general, which may be applied to resolve SDK/framework-level incompatibilities in Android.

In contrast, our study is not limited to a specific cause and has a much larger scale compared to previous studies. We investigate compatibility issues at different phases, including installation and execution, based on actual observations during app executions, rather than relying solely on static code analyses. The detection approaches used in prior studies are predictive in nature and may suffer from false positives, as demonstrated in their results [40, 47]. In comparison, all the compatibility issues we studied are true-positive issues.

Furthermore, previous studies did not specifically address installation-time compatibility issues nor examine the evolution of app incompatibilities. However, their findings complement our research by covering developers' practices in preventing and fixing incompatibilities. Additionally, our study results can be used to improve incompatibility diagnosis techniques and provide valuable insights beyond that scope.

Other compatibility testing techniques. In [68], Wei et al. studied 191 instances of fragmentation-induced compatibility (FIC) issues in five Android apps and analyzed the causes of these issues. Based on their findings, they developed a tool for automatically detecting FIC issues. While Android fragmentation is a significant factor contributing to these issues, it is not the sole cause that we are concerned about in our study. Zhang et al. [73] proposed an approach for testing app compatibility with a particular aim to reduce testing costs. Their approach focuses on evaluating the compatibility of apps with different configurations. In comparison, our study examines app incompatibilities broadly, not specifically concerning one cause or another.

Mimic [45] is a tool for testing the UI compatibility of Android apps through differencing *UI* behaviors between apps or app versions across different devices. Our study implicitly covers, but is not limited to, compatibility issues manifested via app UI differences. Fazzini et al. [23] developed an automated tool for detecting inconsistent behaviors of apps across different Android platforms. While incompatibilities can contribute to the detected behavioral differences, our work primarily focuses on the incompatibilities themselves and their effects.

Characterization of the evolution of compatibility issues. A few studies on Android have applied an evolutionary lens to examine specific characteristics of apps. McDonnell et al. [54] investigated the evolution of the Android API prior to 2013 by analyzing the API update rate. They found that the API evolution was faster than the adoption rate of clients. In [7, 8, 64], the authors focused on the evolution of malware, specifically the effectiveness of anti-malware analysis tools, rather than studying incompatibilities in malware.

Furthermore, Zhang et al.[74] particularly investigated developers' intentions regarding app incompatibilities, which provides another perspective for understanding the causes of incompatibilities. However, this investigation is limited to installation-time issues, although an evolutionary view was also applied.

In comparison, our study focuses on characterizing incompatibilities in both benign apps and malware. We also explore the evolution of app incompatibilities, rather than solely examining the API evolution as a cause of incompatibilities. Additionally, our study spans a much longer period of ten years compared to previous relevant evolution studies.

Comparison to preliminary work. This work is an extension of the preliminary version of our study on Android app incompatibilities [18], which only addressed benign apps of eight years (2010 through 2017) without considering malware, and considering eight Android SDK versions. The extension mainly consists of four aspects: (1) we expanded the evolutionary lens from eight years to 12 (2010 through 2021), hence increasing the size of subject benign app set from 62,894 to 74,545 for the installation-time incompatibility study and from 15,045 to 22,529 for the run-time incompatibility study; (2) we broadened our motivation to cover the need for understanding the incompatibilities in malware as opposed to those in benign apps, hence the security relevance of app incompatibilities in Android; (3) in accordance with (2), we essentially replicated (i.e., doubled the scope of) our preliminary study by examining the same research questions against malware from the same 12 years, including 56,919 and 18,675 subjects for the installation- and run-time incompatibility study, respectively; (4) in addition to examining the prevalence, causes and effects, and contributing factors of the two classes of compatibility issues for malware as for benign apps, we also extensively examined the differences between these two app groups in the same three angles; (5) we increased the number of Android platform/SDK versions from eight to ten (adding API 28 and API 29) for all the studies; (6) we substantially expanded our discussions on the implications of our results and actionable recommendations accordingly to the previous four aspects of extensions; and (7) we updated our discussion on related works to the latest relevant literature and polished the writing of the entire manuscript as a new paper.

10 CONCLUSION

We conducted a comprehensive study on app compatibility issues in the Android ecosystem. Our investigation covered both installation-time issues in 131,464 apps and run-time issues in 41,204 apps. We analyzed the prevalence, distribution, and evolutionary patterns of these compatibility issues over a 12-year period of time from 2010 to 2021. To understand the factors contributing to app incompatibilities, we examined the relationships between these issues and the API levels specified in the apps and used by the devices. We also introduced two app properties, app lapse and API lapse, and investigated their correlation with compatibility issues. Additionally, we explored the impact of other individual app and platform properties on compatibility issues. We particularly looked into both these compatibility characteristics that are common to benign apps and malware, as well as those that differentiate between the two app groups.

Our study design allowed us to uncover new insights and draw novel lessons regarding compatibility issues in Android apps. Based on our findings, we provide practical recommendations for effectively addressing these issues for benign apps. We also provide insights into the security relevance of app compatibility issues, as well as actionable recommendations on how to enhance defenses against malware in Android by utilizing our findings and insights on the compatibility issues in the malware studied across 12 years.

One immediate step for future work is to leverage our study results on installation-time incompatibilities to develop automated tools that detect and even repair those compatibility issues. These tools need to be efficient (e.g., running much faster than actually installing apps to Android devices) and they will complement existing tools that focus on detecting/repairing API/evolution-induced incompatibilities that may occur during app executions. Meanwhile, such existing tools and the like may be improved in efficacy by utilizing our results on run-time incompatibilities (e.g., generating inputs towards covering calls to updated APIs to exercise those issues). Another promising direction is to develop prioritization strategies in new dynamic malware detection approaches as informed by our insights into the security relevance of app incompatibilities.

ACKNOWLEDGMENT

We thank our associate editor and reviewers for insightful and constructive comments. This work was supported in part by the Open Technology Fund (OTF) under Grant B00236-1220-00 and by the U.S. Office of Naval Research (ONR) under Grant N000142212111.

REFERENCES

[1] [n.d.]. Android 14 | Customizable, Accessible & Protective | Android. https://www.android.com/android-14/.
[2] [n.d.]. Behavior Changes: All Apps | Android Developers. https://developer.android.com/about/versions/14/behavior-changes-all.
[3] 2024. VirtusTotal. https://www.virustotal.com/
[4] 2024. What's New in Android 15, plus More Updates. https://blog.google/products/android/android-15/.
[5] Haldun Akoglu. 2018. User's guide to correlation coefficients. *Turkish journal of emergency medicine* 18, 3 (2018), 91–93.
[6] 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
[7] Haipeng Cai. 2018. A preliminary study on the sustainability of android malware detection. *arXiv preprint arXiv:1807.08221* (2018).
[8] Haipeng Cai. 2020. Assessing and improving malware detection sustainability through app evolution studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–28.
[9] Haipeng Cai. 2020. Embracing mobile app evolution via continuous ecosystem mining and characterization. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 31–35.
[10] Haipeng Cai, Xiaoqin Fu, and Abdelwahab Hamou-Lhadj. 2020. A study of run-time behavioral evolution of benign versus malicious apps in android. *Information and Software Technology* 122 (2020), 106291.
[11] 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)*. 265–269. https://doi.org/10.1145/3196398.3196433
[12] Haipeng Cai and John Jenkins. 2018. Towards sustainable android malware detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 350–351.
[13] 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
[14] 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
[15] 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
[16] 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
[17] Haipeng Cai and Barbara Ryder. 2020. A longitudinal study of application structure and behaviors in android. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2934–2955.
[18] 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*. 216–227.
[19] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.
[20] 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
[21] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding Android Security. *IEEE Security & Privacy* 1 (2009), 50–57.
[22] 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
[23] 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
[24] Xiaoqin Fu and Haipeng Cai. 2019. On the deterioration of learning-based malware detectors for android. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*. 272–273.
[25] Google. 2019. Android Compatibility. https://developer.android.com/guide/practices/compatibility.html. Last accessed: May 20, 2019.
[26] Google. 2019. Android Debug Bridge. https://developer.android.com/studio/command-line/adb.html. Last accessed: May 20, 2019.
[27] Google. 2019. Android Developer Dashboard. http://developer.android.com/about/dashboards/index.html. Last accessed: May 20, 2019.
[28] Google. 2019. Android Developer Guide. https://developer.android.com/guide. Last accessed: May 20, 2019.

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

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

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

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

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

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

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

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

[37] Google. 2020. Android Stopped releasing the distribution of devices running different versions of Android. https://www.gizmochina.com/2020/04/11/google-takes-the-android-distribution-chart-off-the-web-but-heres-the-latest-data/. Last accessed: July 13, 2023.

[38] Google. 2023. Android Developer Reference. https://developer.android.com/reference. Last accessed: July 13, 2023.

[39] 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

[40] 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*. 167–177. https://doi.org/10.1145/3238147.3238185

[41] 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.

[42] 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.

[43] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.

[44] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 86–98.

[45] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2019. Mimic: UI compatibility testing system for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 246–256.

[46] 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

[47] 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*. 153–163. https://doi.org/10.1145/3213846.3213857

[48] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. 2017. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology* 32 (2017), 1108–1124.

[49] 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

[50] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. 2022. Automatically detecting API-induced compatibility issues in Android apps: a comparative analysis (replicability study). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 617–628.

[51] Pei Liu, Yanjie Zhao, Mattia Fazzini, Haipeng Cai, John Grundy, and Li Li. 2023. Automatically detecting incompatible android apis. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–33.

[52] 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

[53] Tarek Mahmud, Meiru Che, and Guowei Yang. 2021. Android Compatibility Issue Detection Using API Differences. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 480–490.

[54] 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

[55] Marco Mobilio, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2020. FILO: FIx-LOcus localization for backward incompatibilities caused by Android framework upgrades. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1292–1296.

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

[57] 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

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

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

[60] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 288–298.

[61] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Valentina Piantadosi, Michele Lanza, and Rocco Oliveto. 2020. API compatibility issues in Android: Causes and effectiveness of data-driven detection techniques. *Empirical Software Engineering* 25, 6 (2020), 5006–5046.

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

[63] 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.

[64] 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

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

[66] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automated deprecated-API usage update for android apps: How far are we?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 602–611.

[67] VirusShare.com. 2019. VirusShare. https://virusshare.com/. Last accessed: May 20, 2019.

[68] 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

[69] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 878–888.

[70] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1176–1199.

[71] 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.

[72] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 886–898.

[73] 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

[74] Ziyi Zhang and Haipeng Cai. 2019. A look into developer intentions for app compatibility in android. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 40–44.

[75] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy*. 95–109.