Understanding Android OS Forward Compatibility Support for Legacy Apps: A Data-Driven Analysis

Shuang Li*†, Rui Li*†, Yifan Yu*†, Kailun Yan*†, Shishuai Yang*†, and Wenrui Diao*†(✉)
*School of Cyber Science and Technology, Shandong University
{lishuang128, leiry, yuyifan, kailun, shishuai}@mail.sdu.edu.cn, diaowenrui@link.cuhk.edu.hk
†Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University

***Abstract*—The update of Android OS constantly brings users various new features and enhances system security. On the other hand, the system and API modifications with the update may introduce the app compatibility issue. The app's SDK version may not align with the Android OS version, making apps not work adequately. This condition will inevitably damage the Android ecosystem. Thus, while developing Android OS, Google considered and deployed compatibility support. The software engineering research community also noticed the Android compatibility issue and conducted some investigations. However, most previous studies focus on apps' performance and solutions on compatibility (apps running on multiple OS versions). Rare work considers the Android OS side's forward compatibility implementations (supporting legacy apps running on the latest OS).**

This work systematically studied how Android OS implements forward compatibility for the apps developed with outdated SDKs, primarily focusing on the targetSdkVersion-based fine-grained control. Specifically, we propose three research questions, covering: 1) the forward compatibility support approaches; 2) the stability of forward compatibility support; and 3) the invocations of the APIs with forward compatibility support in third-party market apps. To address these questions, we conducted comprehensive measurements on Android's forward compatibility support, including its implementation, implications, and evolution. Our measurements were based on large-scale datasets covering the source code of Android 8.0∼13 and 130,461 apps. Finally, we provide rich data support and analysis to answer these questions. This study offers new insights into Android's forward compatibility support, helping the research community understand the evolution of Android's API design.

I. Introduction

Android is the most popular mobile operating system, with a market share of around 70% as of September 2023 [13]. It has continuously been evolving to provide users with a better experience. Each major Android update corresponds to a released SDK for app development. The API level uniquely identifies the framework API revision offered by a version of the Android platform [19]. On the one hand, app developers may not use the latest SDK due to the concern of development cost and cycle. On the other hand, Android devices may run various versions of Android OS since Google does not control the distribution of Android devices or software. As a result, the SDK version used for developing an app may not align with the Android OS version (i.e., the framework API level), causing the app compatibility issue.

Android app compatibility means an app can run properly on a specific version of the Android platform [5]. Failure to provide compatibility support will result in app crashes or uncertain behaviors [28]. Compatibility is a considerable challenge for app development and Android OS design. To solve this issue, in the app configuration, the `targetSdkVersion` attribute [19] is introduced to specify the SDK version that this app should run. Depending on `targetSdkVersion`, app compatibility can be further divided into *forward* and *backward compatibility* (F/B-compatibility for short). F-compatibility means that apps targeting low SDK versions can run adequately on Android systems with high API levels, and vice versa for B-compatibility. Google claims that apps are generally forward-compatible with the latest Android versions [7]. However, Android OS updates may introduce new security requirements or vulnerability fixes that could lead to attacks on legacy apps if forward compatibility is always maintained. For example, in apps targeting API level 21 or higher, passing an implicit intent throws a security exception [8], while for apps targeting low API levels, service hijacking vulnerabilities [16] still exist without warning due to compatibility support. Therefore, it is important to understand Android's F-compatibility implementations and whether they maintain the security risks of apps with low target APIs.

Previous works focused on the compatibility issues of Android apps [35], [28], [26], [36], [34], [23], [24], [32], [25]. These studies mainly focus on *app code*, such as how to detect or fix app compatibility problems. *Nearly no previous study focuses on how Android OS achieves F-compatibility support*. Our investigation shows that the F-compatibility of the Android OS is primarily concerned with method removal and method change. For the former, Android uses the `@Deprecated` annotation to caution developers against using the method that will be removed. For the latter, Android leverages the app's `targetSdkVersion` attribute, allowing precise control over methods with internal alterations. Li et al. [29] studied method removal. However, due to the complexity of method change, it has not yet been explored in previous work.

Our Work. In this study, we conducted a systematic analysis of F-compatibility implementations in Android OS. We focus especially on the more complex aspect of F-compatibility, namely method change. Through this, we shed light on the design strategy behind the Android API and quantify the impact of the F-compatibility implementation. To gain a comprehensive understanding of Android's forward compatibility, we analyzed its current status in the latest version, its evolution between two consecutive versions, and its usage in apps. This can be summarized into the following three research questions:

RQ1 ***How does Android maintain its availability for legacy apps?*** How does the latest Android version handle apps targeted at different API levels? Is it possible to loosen restrictions on legacy apps in certain scenarios, potentially leading to security risks?

RQ2 ***How stable are the F-compatibility implementations with the evolution of Android?*** Each update to Android OS brings various changes to fix bugs and improve user experience. What are Google's strategies for adjusting the existing F-compatibility implementations?

RQ3 ***What is the current state of third-party apps calling F-compatibility-related APIs?*** Google Play now requires apps to target API level 33 or higher, which means legacy apps will not be available on devices running new OS versions [18]. However, what is the current state of using APIs with F-compatibility support for apps on third-party markets?

To answer the above questions, we developed targeted research approaches to investigate the implementations, implications, and evolution of F-compatibility support on Android. Primarily, we focus on the `targetSdkVersion`-based fine-grained control. Our dataset contains multiple versions of the Android source code (8.0∼13) and 130,461 apps. Also, to achieve the measurement, we designed a static data-flow analysis approach to locate the F-compatibility implementations in the massive Android source code. Then, multi-dimensional targeted measurements are conducted to understand the Android compatibility strategies.

Contributions. We list the main contributions of this paper.

- *Large-scale measurement.* We constructed large-scale datasets for the empirical measurement study, covering multiple versions of the Android source code (8.0∼13) and 130,461 third-party market apps.
- *Systematic analysis.* We systematically studied how Android achieves F-compatibility on the code level. In addition, we designed three research questions to explore, as answered in the following.
  1) Android 13 has 2,376 F-compatibility implementations. It uses three ways to maintain its availability for legacy apps: allowing the execution of obsolete code, reducing checking conditions, and adjusting internal code logic. In some situations, these ways lead to loosening restrictions on lower-version apps, leading to potential security risks such as information disclosure.
  2) As Android OS evolves, its source code has an increasing trend of F-compatibility implementations. Android OS continuously maintains F-compatibility implementations for multiple versions, and removal cases are rare.
  3) Most third-party market apps (99.3%) target outdated API levels. Furthermore, 98.0% of market apps use APIs related to the deployed F-compatibility support of Android OS.

Data Availability. The code of our analysis tool and the raw measurement data for each research question are available at `https://doi.org/10.21227/gtme-x022`.

Target and Beneficiary. This study contributes new knowledge on how Android ensures forward compatibility support, helping researchers understand the evolution of Android's API design. Framework maintainers can refine strategies for F-compatibility support and reduce security risks through our data analysis. Our analysis encourages third-party markets to limit apps' target API levels. Developers should update apps' target API levels rather than relying on Android's forward compatibility. Our analytical approach is versatile and can also be implemented in Java-based development platforms.

II. Background

This section provides the necessary background on Android API levels and compatibility.

A. Android API Levels

The Android OS is constantly upgraded over time, and each significant change is associated with an API level, an integer uniquely identifying an Android version. For example, Android 11 is uniquely identified by API level 30 [4]. App developers can specify an app's API level requirements for the Android platform in its manifest file through the `uses-sdk` element [19]. There are three main API-level settings, as follows:

- `minSdkVersion`: the minimum API level of the Android platform that can run the app. Android checks the value of this attribute when installing an app and denies the installation if it exceeds the current system's API level.
- `targetSdkVersion`: the target API level at which to run the app and the level most suitable for running the current app. If the app does not declare this attribute, the default value equals `minSdkVersion`.
- `maxSdkVersion`: the maximum API level that supports the app. The Android system checks the value of this attribute when installing an app or re-verifying it after system updates. If the value of this attribute is lower than the current system's API level, the app will not be installed or be invisible to the user. Android does not recommend developers set this attribute, because it will block the app's deployment on Android's new versions.

Configuring the app with appropriate API levels can ensure users benefit from security, privacy, and performance improvements of the recent Android OS and avoid apps' installations on unsuitable Android platform versions [21].

B. Android Forward Compatibility

Due to the open nature and rapid evolution of the Android OS, a wide variety of Android devices are available on the market with varying OS versions. It causes compatibility issues for Android apps [35]. Android app compatibility means that the app can run properly on a specific version of the Android platform [5]. As illustrated in Figure 1, it can be divided into *backward and forward compatibility* based on the value of the

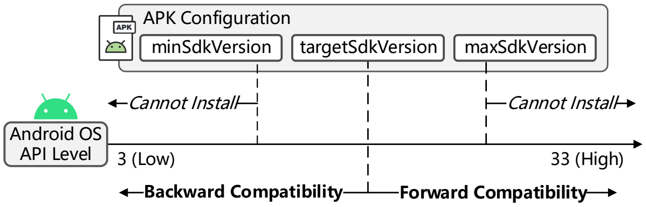

Fig. 1: Backward and forward compatibility.

app's `targetSdkVersion` attribute. Backward compatibility (B-compatibility) happens when `tagetSdkVersion` is higher than the current running Android system's API level [6]. On the contrary, forward compatibility (F-compatibility) occurs when `tagetSdkVersion` is lower than the system's API level [7].

We conducted extensive investigation and identified two scenarios for the Android OS to achieve forward compatibility. Specifically, every new Android version introduces changes to improve the user experience, security, and performance of the Android platform overall, including removing and updating existing framework methods and adding new ones. Android claims that apps are generally forward-compatible with the latest Android versions. That is, legacy apps can run properly on new versions [7]. To eliminate the impact of the method removal and change on apps' compatibility, Android performs the following strategies:

(1) *Method removal. Use `@Deprecated` annotations to mark the methods that are planned for removal, will soon be removed, and should no longer be used.* For example, as shown in Listing 1, in Android 5.0 (API level 22), the `getRecentTasks` method is deprecated because it can leak sensitive information to the caller and is no longer available to third-party apps [10]. Since removing this method directly would cause the legacy app (targeting API level 21 or lower) invoking it to crash at runtime after users update their phones to Android 5.0, Android marks this method with the `@Deprecated` annotation first, leaving developers a period to adjust their apps.

```
/** @deprecated As of {@link android.os.
    Build.VERSION_CODES#LOLLIPOP} (API level
     22) ...**/
@Deprecated
public List<RecentTaskInfo> getRecentTasks
    (...){ ... }
```

Listing 1: Example of using the `@Deprecated` annotaion.

(2) *Method change. Use app's `targetSdkVersion` attribute to perform a fine-grained control.* For instance, as shown in Listing 2, the `restorePermissionState` method is used to update the granted permission status requested during the app installation. Android 6.0 (API level 23) introduces the runtime permission model, and the permission granting rules are changed to make permissions more understandable, useful, and secure for users [14]. To avoid affecting the permission-related functions of the legacy app (targeting API level 22 or lower), the system distinguishes apps based on their `targetSdkVersion` attributes and performs permission granting for the legacy app.

```
private void restorePermissionState(...) {
    ...
 final boolean appSupportsRuntimePermissions
      = pkg.getTargetSdkVersion() >= Build.
      VERSION_CODES.M; // API level 23...
 if (...) { ... }
 else if (bp.isRuntime()) { ...
    if (appSupportsRuntimePermissions) { ...
         }
    else { ...
        if (! uidState.isPermissionGranted(
            bp.getName()) && uidState.
            grantPermission(bp)){ ... }
        ... }... }}
```

Listing 2: Example of using the `targetSdkVersion` attribute.

In the previous research, Li et al. [29] systematically analyzed the current situation of Android deprecated APIs, including their implementations and evolution. However, they only identified deprecation tags (i.e., `@Deprecated`) in the Android source code, leading to their analysis being conducted at the method level without going into the specific logic code inside the method. In comparison, our work focuses on understanding how Android achieves F-compatibility through `targetSdkVersion`-based fine-grained control at the code level, which the Android research community neglects. Specifically, we try to provide a comprehensive study by answering the research questions proposed in Section I.

III. Methodology

In this section, we present our methodology to answer the research questions proposed in Section I.

A. Overview

As illustrated in Figure 2, our analysis contains three main stages as follows:

Stage 1 *Dataset Construction.* We collected the source code for Android 8.0∼13 and built an APK dataset for analysis.

Stage 2 *F-compatibility Implementation Locating.* All our research questions require locating the Android's F-compatibility code that is implemented based on the app's `targetSdkVersion` attribute. We achieved this through data flow analysis.

Stage 3 *Targeted Measurement Analysis.* Against the proposed research questions, we performed multi-dimensional measurements and targeted analyses.

Note that, since Stage 3 is closely related to the content of the RQs, the corresponding analysis approaches are introduced in Section IV.

B. Dataset Construction

We need to construct two types of datasets for this study, as follows:

- *Android System Dataset.* We compiled the source code of AOSP (Android Open Source Project) Android 8.0∼13 (covering over 90% of devices [4]) and obtained their

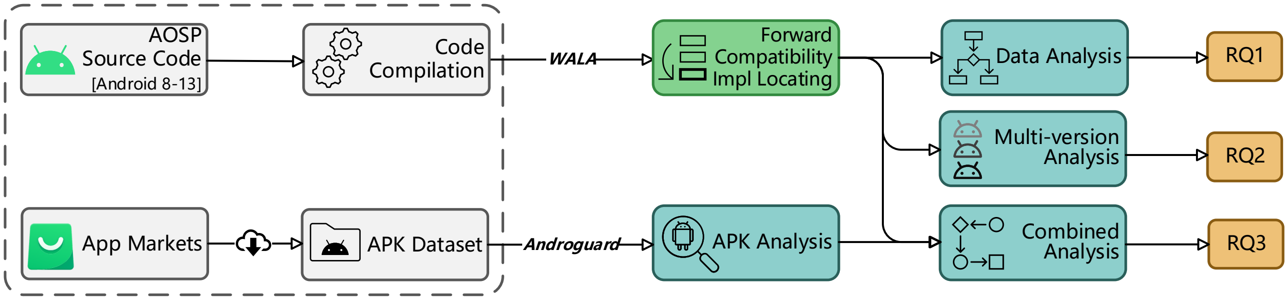

Fig. 2: Overview methodology flow.

/system directories, which contain the Android framework [17], for the subsequent analysis. The differences between different Android versions of the same API level are minor, so we only chose the latest source code version for each API level.

- *APK Dataset*. As mentioned in RQ3 of Section I, Google Play has set specific requirements for their apps. Currently, it requires apps to have a targetSdkVersion that is greater than or equal to API 33 [12]. Therefore, we selected four popular third-party app markets without such requirements to crawl apps in October 2023, including 2265, Lenovo, Leyou, and Mdpda. After deduplicating and keeping only the latest versions of apps with the same signature, a total of 130,461 apps were obtained.

C. F-Compatibility Implementation Locating

As an essential step for the subsequent analysis, we need to locate the F-compatibility implementations in Android, which perform different handling treatments for apps with different targetSdkVersion configurations. Therefore, it is necessary to identify the conditional statements related to apps' targetSdkVersion attributes, like if(targetSdkVersion <= Build.VERSION_CODES.Q). However, due to diverse implementations, it is not enough to look for such a conditional statement, of which condition is a constant comparison associated with the targetSdkVersion keyword. As we trace all F-compatibility implementations, three challenges affect accurate localization, as summarized below.

Challenge 1: Conditional Result Passing. As shown in Listing 3, targetSdk of the onApplyThemeResource method stores the value of app's targetSdkVersion attribute, and it is compared with Android version code Q (API level 28). The comparison result is saved in the targetPreQ variable. Without sequentially tracking targetPreQ, we would not know the exact location of the F-compatibility implementation. In other cases, the comparison result could be stored in a class field, and we also need to continue tracing this field.

```
// Class: Activity
protected void onApplyThemeResource(...) {
    ...
  final boolean targetPreQ = targetSdk <
      Build.VERSION_CODES.Q;
  if (!targetPreQ){...} ... }
```

Listing 3: Example of conditional result passing.

Challenge 2: Diverse targetSdkVersion Usages. In actual implementations, the usages of targetSdkVersion in the conditional statements are diverse. For example, as shown in Listing 4, the parseUsesStaticLibrary method invokes getTargetSdkVersion to obtain targetSdkVersion for further comparison instead of using it directly.

```
// Class: ParsingPackageUtils
private static ParseResult<ParsingPackage>
    parseUsesStaticLibrary(...){...
  if (pkg.getTargetSdkVersion() >= Build.
      VERSION_CODES.O_MR1) ... }

// Class: ParsingPackageImpl
public int getTargetSdkVersion() {
  return targetSdkVersion;}
```

Listing 4: Example of diverse targetSdkVersion usages.

Challenge 3: Diverse Variable Comparison. In the conditional statement, targetSdkVersion is compared with a variable rather than a constant. For instance, as shown in Listing 5, targetSdkVersion is compared with versionCode, which is the parameter of the isTargetSdkLessThan method. Without tracing the source of this parameter, we cannot confirm whether this passed integer is a constant for version comparison.

```
// Class: WifiPermissionsUtil
public boolean isTargetSdkLessThan(String
    packageName, int versionCode, int
    callingUid) { ...
  return targetSdkVersion < versionCode; }
```

Listing 5: Example of diverse variable comparison.

Due to the large amount of code and complex call chains in the Android source code, the static analysis approach is suitable for detecting it at scale. In the following, we present our data flow analysis solutions to locate the F-compatibility implementations and explain how to solve these challenges. At a high level, we trace the propagation of the targetSdkVersion attribute defined in the app manifest file and locate the relevant conditional statements. The analysis implementation is based on WALA [20] – a static analysis library for Java and related languages.

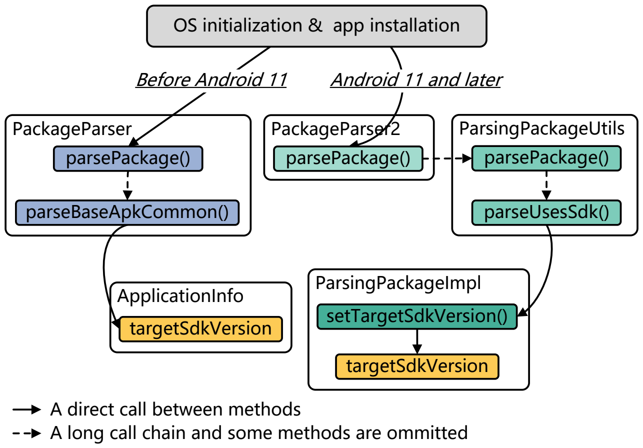

Fig. 3: Initial tracing variable locating.

Initial Tracing Variable Locating. First, we need to locate the initial definition held by the system that corresponds to the declaration of the `targetSdkVersion` attribute in the app manifest file. Therefore, we analyzed the app manifest parsing procedure of Android OS. As illustrated in Figure 3, the system parses the app manifest file during OS initialization or app installation. It puts the value specified by the `targetSdkVersion` attribute into the `targetSdkVersion` field of the `AppInfo` class (< Android 11) or the `targetSdkVersion` field of the `ParsingPackageImpl` class (≥ Android 11).

Subsequently, we trace the intra-procedural and inter-procedural propagation of the `targetSdkVersion` field (`AppInfo.targetSdkVersion` or `ParsingPackageImpl.targetSdkVersion`) to identify the related conditional statements of F-compatibility. Our analysis begins with the methods of accessing this field directly. In the intra-procedural analysis, we trace the propagation of it within a method. In the inter-procedural analysis, we trace its propagation among methods.

Intra-procedural Analysis. We have an entry tracing variable (ETV) passed into each method that needs to be analyzed. This ETV is determined in our inter-procedural analysis, associated with the `targetSdkVersion` field. Then we locate the block involving the ETV and iterate through all subsequent blocks in the CFG (Control Flow Graph) starting from it. Suppose that the ETV propagates to a new variable. In that case, we record the new variable, tracing it like the ETV to ensure that we do not miss any relevant conditional statements. If the ETV and other child tracing variables propagate to a return statement, a method invocation statement, or a field, we also record it and perform inter-procedural analysis.

Inter-procedural Analysis. This analysis can be classified into three cases.

(1) *Field propagation.* The tracing variable in method A is passed to a field. We search the method(s) that accesses this field, say B, and treat the variable assigned by the field assignment statement as the ETV of method B for intra-procedural analysis.

(2) *Parameter propagation.* The tracing variable in method A is passed to a parameter of method B. We treat this parameter as the ETV of method B for intra-procedural analysis.

(3) *Return value propagation.* The tracing variable in method A is passed as a return value of A. We search the caller(s) of method A, say B. In method B, we locate the assignment statement that invokes method A and assigns the return value to a variable. This variable will be treated as the ETV of method B for intra-procedural analysis.

One difficulty of inter-procedural analysis is Java's interface and inheritance mechanisms. Fortunately, WALA's CHA (class hierarchy analysis) solves this problem to some extent. Specifically, if the called method cannot be uniquely determined, WALA will parse all possible called methods. We build an inter-procedural call graph based on the CHA. If the tracing encounters a method call statement, we obtain all possible called methods and build a call relationship chain with the caller method.

Solutions to Challenges. Combining the above intra- and inter-procedural analysis approaches, we can address Challenges 1 and 2. In particular, when addressing Challenge 1, we notice a noteworthy situation. Some fields in the Android source code are marked with the `EnableSince` or `EnabledAfter` annotation, which was introduced in Android 11 [9]. `EnableSince` means "≥" and `EnabledAfter` means ">". We manually tracked the propagation of these fields and finally discovered this type of F-compatibility usage pattern. Since the call chain is complex, we use the example in Listing 6 to explain this pattern and only introduce the key propagation nodes here. The `REQUIRE_EXACT_ALARM_PERMISSION` field marked with `EnabledSince` is passed into the `isChangeEnabled` method as a parameter. Then, if the app's `targetSdkVersion` is not less than 31 (i.e., version code S), the `isChangeEnabled` method will return `true`, and vice versa is `false` (Line 7). That is, the return value of `isChangeEnabled` stores the comparison result of `targetSdkVersion` and a constant (API level), which also needs to be traced. Therefore, we scan the Android source code to obtain all fields with the `EnabledSince` or `EnabledAfter` annotation and search the methods that take one of these fields as a parameter and return a `boolean` type value. Then, we trace the propagation of these methods' return values. For Challenge 3, we trace back the variable compared with `targetSdkVersion` to confirm whether its source is a constant.

```
1 // Class: AlarmManager | Definition
2 @EnabledSince(targetSdkVersion = Build.
      VERSION_CODES.S)
3 public static final long
      REQUIRE_EXACT_ALARM_PERMISSION =
      171306433L;
4
5 // Class: AlarmManagerService | Use
6 private static boolean
      isExactAlarmChangeEnabled(String
      packageName, int userId) {
7   return CompatChanges.isChangeEnabled(
        AlarmManager.
```

TABLE I: Number of locations with F-compatibility support for specific API level in Android 13 (API level 33)† .

API Level Constant‡	3	4	7	8	10	12	13	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
General type	17	10	10	7	22	2	176	10	246	16	18	1	283	1	352	114	2	76	9	149	140	61	42	5	50
EnabledX type	-	-	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-	-	-	1	2	79	251	18	203

†: This table only includes API levels relevant to F-compatibility implementations.
‡: The constant that is compared with the value of the app's `targetSdkVersion` attribute.

```
        REQUIRE_EXACT_ALARM_PERMISSION,
        packageName, UserHandle.of(userId)); }
```

Listing 6: Example of EnabledX.

IV. Measurements and Findings

In this section, we provide a targeted data analysis of the proposed research questions and explain the reasons behind the results.

RQ1. How does Android maintain its availability for legacy apps?

Under this RQ, we focus on analyzing the conditional statements with respect to F-compatibility and the subsequent branch processing implementations in the Android source code. Also, we give an overview of current F-compatibility implementations in a recent Android OS version, Android 13.

Analysis Approach. Section III-C has described how to locate the F-compatibility implementation code. In practice, for each location, we record the related method data, conditional statement, Android API level used in this conditional statement, and processing statements in each branch of this statement.

Findings: F-compatibility Implementation Statistics. As listed in Table I, in Android 13 (API level 33), we identified 1,819 F-compatibility implementations for the general case (i.e., `targetSdkVersion` comparison) and 557 for the `EnabledSince` and `EnabledAfter` case.

In Table I, it can be noted that Android 13 supports almost all previous API levels with F-compatibility implementations. In particular, the comparison occurs 352 times on API level 22, which is the highest among all versions. The main reason is that Android 6.0 introduced the runtime permission mechanism [14]. That is, if the `targetSdkVersion` of an app is $\leq$ API level 22, Android will disable the runtime permission mechanism. In addition, API level 30 appears most frequently in `EnabledX`-related (`EnabledSince` or `EnabledAfter`) F-compatibility implementations and far exceeds the general implementations for the corresponding version. It is because a new compatibility testing tool marked with `EnabledX` was introduced in Android 11, and developers can use it to check whether the app is compatible with the behavior changes in the latest platform [9]. Specifically, developers can use the adb (Android Debug Bridge) tool to turn on or off a behavior change, so they do not need to change the `targetSdkVersion` of their apps to observe the impact of platform behavior changes.

The purpose of Android's F-compatibility implementation is to enable legacy apps to continue running in their previous state, but this practice comes with potential security risks. Mutchler et al. [33] studied the risky cases related to F-compatibility implementations. For example, apps targeting outdated SDKs can still invoke obsolete APIs with security risks while running on the latest Android OS due to compatibility issues. Therefore, we are curious about the number of F-compatible implementations in the Android source code that impose varying degrees of restriction on apps targeting different API levels. In other words, how many F-compatibility implementations loosen their restrictions on low-version apps (apps targeting old SDKs)?

Findings: F-compatibility Implementation Measures. By analyzing the branch code, we summarized and categorized the processes of Android OS to achieve F-compatibility, including three main treatment measures as follows.

(1) *Allowing the execution of obsolete code.* For high-version apps, the system prevents the execution of a method by throwing an exception or returning a void value. Still, for low-version apps, the system keeps the original code execution. For example, on Android, a service [15] is a component that can perform long-running operations in the background. Apps specify the services they want to interact with through Intents. Intent [11] can be divided into explicit and implicit. Explicit Intent specifies the desired service, while implicit Intent provides abstract info for the system to select the appropriate service. Using an implicit Intent can be unsafe as malicious apps can create services that match the Intent and perform service hijacking attacks [16], leading to security issues such as information disclosure. Therefore, for apps targeting API level higher than or equal to 21, security exceptions will be caused when implicit intentions are passed to service-related APIs such as `startService`, `bindService`, `stopService`, etc. As shown in Listing 7, these APIs call the `validateServiceIntent` method in the `ContextImpl` class. It throws an exception for the apps with `targetSdkVersion` higher than or equal to `LOLLIPOP` (API level 21), but only prints logs for other apps.

```
//ContextImpl.validateServiceIntent
if (getApplicationInfo().targetSdkVersion >=
     Build.VERSION_CODES.LOLLIPOP) {...
    throw ex;
} else {Log.w(...);}
```

Listing 7: Example of throwing exception.

(2) *Reducing checking conditions.* The system applies more checking conditions for high-version apps than for low-version apps, such as permission checking. As shown in Listing 8, if the app's `targetSdkVersion` is higher or equal to `P` (API level 28), the `uninstall` method in the `PackageInstallerService`

TABLE II: F-compatibility implementations by measure.

Type	Allow execution	Reduce checking	Adjust internal logic
General type	393	268	1,158
EnabledX type	102	57	398

class will check whether the `REQUEST_DELETE_PACKAGES` permission has been granted to it. This forward compatibility implementation will cause apps targeting low API levels to apply for deletion of apps without permission.

```
//PackageInstallerService.uninstall
if (appInfo.targetSdkVersion >= Build.
    VERSION_CODES.P) {
    mContext.enforceCallingOrSelfPermission(
        Manifest.permission.
        REQUEST_DELETE_PACKAGES, null);}
```

Listing 8: Example of additional checking condition.

(3) *Adjusting internal code logic.* In this case, the OS has no obvious tendency to loosen restrictions on low-version apps. For example, as shown in Listing 9, the `onKeyDown` method in the `Activity` class calls different methods for apps with the `targetSdkVersion` before and after `ECLAIR` (API level 5).

```
//Activity.onKeyDown
if (getappInfo().targetSdkVersion
        >= Build.VERSION_CODES.ECLAIR) {
    event.startTracking();} else {
    onBackPressed();}
```

Listing 9: Example of different execution logic.

Discussion. Our analysis data indicate that the latest version of Android OS includes forward compatibility implementations for apps targeting various API levels, even old ones. Furthermore, as listed in Table II, we counted the number of the above three ways to achieve F-compatibility, respectively. The first two ways relate to the loosening of restrictions on low-version apps. It shows that 34.5% of Android F-compatibility implementations (general and `EnabledX` methods) loosen restrictions on low-version apps, which could pose security risks. Android aims to maintain a large number of available methods instead of directly blocking invocations. However, implementing this measure could lead to potential security problems for apps in the long run. For F-compatibility, how to balance security and usability is still an open question.

Answers to RQ1

Android utilizes three methods to ensure that legacy apps remain functional: allowing the execution of obsolete code, reducing checking conditions, and adjusting internal code logic. However, its forward compatibility implementations have several instances of relaxing restrictions on apps targeting low API levels, which may lead to potential security risks.

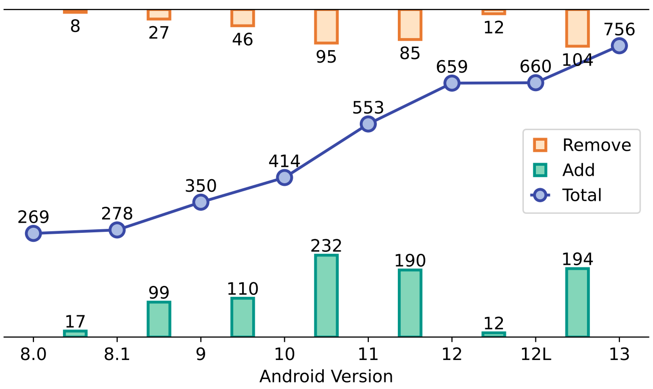

Fig. 4: Changes in numbers of TAC statements.

RQ2. How stable are the F-compatibility implementations with the evolution of Android?

Each Android version brings numerous updates that may affect existing F-compatibility implementations. Under this RQ, we measure the changes in F-compatibility implementations between two adjacent versions of Android 8.0∼13.

Analysis Approach. For a better understanding of the working practices of Android OS developers, we identify the F-compatibility implementations in each Android version by keeping a record of the F-compatibility conditional statements. For general type, the conditional statements include `targetSdkVersion` attributes and API level constants, and for EnabledX types, the conditional statements contain a method like `isChangeEnabled`. However, we no longer track the propagation of conditional results. To distinguish this situation from the F-compatibility implementations in other places of this paper, we call these locations the `targetSdkVersion` attribute conditional statements (*TAC statements* for short). Then, we combine these statements with their method signatures as the signature of F-compatibility implementations. The method signature consists of the method's name, arguments, return value type, and declaring class. Next, we compare the obtained F-compatibility signatures between two adjacent versions and manually check the discrepancies in the code implementations.

Findings: TAC Statements. Figure 4 shows the increase in F-compatibility conditional statements (general and EnabledX type) from Android 8.0 to 13. Since each Android version needs to implement the F-compatibility of apps with the previous version of the system, the increase in TAC statements is inevitable. We are more curious about the removed TAC statements and the reasons for removing them. Therefore, we checked the removed TAC statements and found three types of removals.

(1) *Statement removed.* The TAC statement has been removed in the method of the next version. For example, there is a method `setRequestedOrientation` in the `ActivityRecord` class. Due to the mistakes of Android OS developers, in Android 8.0, if the `targetSdkVersion` is > API level 26, the system will throw an exception and terminate the running app.

TABLE III: The numbers of removed TAC statements between adjacent Android versions.

Types	Android 8.0→8.1	Android 8.1→9	Android 9→10	Android 10→11	Android 11→12	Android 12→12L	Android 12L→13
Statement Removed	3	4	6	7	11	0	5
Method Removed	2	5	14	13	18	0	9
Method Changed†	3	18	26	75	56	12	90
Total Removed	8	27	46	95	85	12	104

†: Being recognized as "removal" due to the changed method signature, but the F-compatibility conditional statements have not been removed.

In Android 8.1, the developer removed this F-compatibility conditional statement to fix this bug [1]. However, this confuses many app developers as Android developers do not explain this in their documentation or source code [2].

(2) *Method removed.* The method of the previous version containing the TAC statement is removed in the next version. For example, the `isTaskForcedMaximized` method in the `TaskLaunchParamsModifier` class checks whether the `targetSdkVersion` of the app is less than 4. This method was removed in Android 11, and its caller did not invoke an alternative method with similar functionality. Android may consider that some F-compatibility implementations for very old versions are no longer necessary.

(3) *Method changed.* The signature of the method containing the TAC statement between the two versions has changed slightly, which is recognized as "removal". However, the corresponding method in the next version retains the TAC statement and its branch codes unchanged. Since the change in the next-version method is irrelevant to forward compatibility, it does not affect the original F-compatibility implementations.

Discussion. To summarize, Figure 4 indicates that only 11.8% of the total TAC statements were removed. Besides, Table III shows that the number of *method changed* accounts for 74.3% on average between versions. However, this type does not change the original TAC statement, so we believe the Android OS developers still maintain their F-compatibility. Therefore, only 3.0% of TAC statements have not been maintained after excluding the third type. This indicates that Android prefers to keep the previous F-compatibility implementations when updating the internal logic of the methods. It verifies a large amount of F-compatibility in the latest Android OS because Android rarely removes the forward compatibility already implemented. To shorten the period of security risks, we recommend that Android follow operations similar to the deprecate-remove cycle.

Answers to RQ2

With each evolution of the Android OS, there is a growing trend of implementing F-compatibility in its source code. Android OS continuously maintains F-compatibility implementations for a long time, and removal cases are rare.

RQ3. What is the current state of third-party apps calling F-compatibility-related APIs?

To ensure a safe experience for Android and Google Play users, Google Play now requires that newly submitted apps target API level 33 or higher, and legacy apps are not available to users on devices running new Android versions [18]. These requirements thus avoid app F-compatibility issues. However, for the third-party app markets without such requirements, what is the prevalence of legacy apps? What is the usage status of the F-compatibility-related APIs in these apps? In this RQ, we focus on analyzing apps in third-party markets calling F-compatibility-related APIs.

Analysis Approach. We used WALA's CHA (class hierarchy analysis) to obtain all public SDK APIs in android.jar of Android 13 that can reach F-compatibility implementations. They are treated as *SDK APIs with F-compatibility support* (APIs-FC). Also, we extracted the `targetSdkVersion` attributes and invoked SDK APIs from apps for comparative analysis.

(1) *Identify APIs-FC.* The F-compatibility implementations we obtained lie in various components of the Android system. Some of the methods to which they belong are internal and cannot be invoked by apps directly. Therefore, we need to find their top public APIs for apps on the call chain through backward searching.

Specifically, we obtained all the public APIs in android.jar as the initial API set. Then, based on CHA, we constructed the invocation relationship among methods. Thus, we can find the caller of a method and then follow the call chain with BFS (Breadth First Search). From each F-compatibility implementation location, if we encounter a public API, we will map it to this API and identify this API as API-FC. To avoid state space explosion, we empirically limited the number of a method's callers to 20 and the search depth to 20. Finally, we identified 3,807 APIs-FC.

(2) *App analysis.* For each app, we used androguard [3] to extact its `targetSdkVersion` attribute and called SDK APIs. To further reflect the real intention of developers using APIs, we excluded the APIs in the official libraries used by apps, such as beginning with `com.google.*`, `com.android.*`, `androidx.*`, and `android.*`. Then we filtered out the APIs-FC and compared `targetSdkVersion` with the maximal API level constant used in these APIs-FC.

Findings. As mentioned in Section III-B, we collected 130,461 APK files. The percentages of their `targetSdkVersion` attributes are plotted in Figure 5, revealing that most apps do not use the latest SDKs. Only 917 (0.7%) apps target API level 33 or higher, while 81,485 apps (62.5%) still target API level 26 or lower. This means that quite a number of third-

TABLE IV: Top ten most frequently used APIs with F-compatibility support in market apps.

Entry Point Method	API Level	App Amount
`Landroid/content/Context;->getResources()Landroid/content/res/Resources;`	28	119,609
`Landroid/app/AlertDialog$Builder;->create()Landroid/app/AlertDialog;`	32	111,471
`Landroid/content/Context;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;`	32	96,676
`Landroid/view/WindowManager;->getDefaultDisplay()Landroid/view/Display;`	30	89,180
`Landroid/os/Looper;->loop()V`	32	87,043
`Landroid/os/Environment;->getExternalStorageDirectory()Ljava/io/File;`	32	85,019
`Landroid/os/Environment;->getExternalStorageState()Ljava/lang/String;`	32	84,920
`Landroid/content/SharedPreferences$Editor;->commit()Z;`	32	82,907
`Landroid/app/Activity;->finish()V`	32	82,323
`Landroid/content/Context;->getSharedPreferences(Ljava/lang/String;I)Landroid/content/SharedPreferences;`	32	81,944

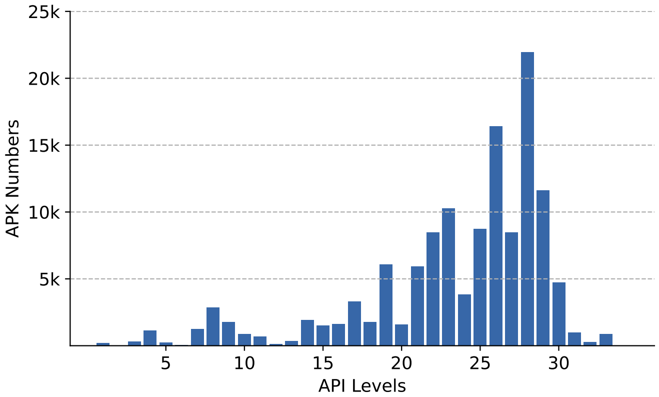

Fig. 5: Apps' `targetSdkVersion` attributes.

party market apps are developed with outdated SDKs. Further, we investigated why so many third-party apps target outdated SDKs by analyzing their update times. The data shows that after the release of Android 13 in May 2022, only 9.0% of apps (11,757) have been updated, with just 7.6% (897) developed for Android 13 or higher. Additionally, 127,853 (98.0%) apps call at least one API-FC with a `targetSdkVersion` attribute no greater than the maximal API level constant.

In addition, we are interested in understanding the reasons behind the most frequently used APIs-FC and the unused ones in apps. Table IV lists the top 10 most frequently used APIs-FC. Some belong to the system's essential functionalities and must be used during the app development. For example, the APIs-FC (`getResources()`, `getSystemService()`, and `getSharedPreferences()`) all belong to the `Context` class, and `Context` is an interface class for global information about the app environment. Other APIs-FC also invoke basic functionalities like creating dialog boxes or getting the default screen. For example, `create()` is used to create dialog boxes to interact with users.

On the other hand, 819 (21.5%) APIs-FC are not invoked by any app. There are three main reasons.

- *Newly added APIs.* Most of the apps in our APK dataset target API level 29 or lower, so some new APIs added in higher Android versions are never used, such as the `addOrUpdateStatus` API introduced in Android 12.
- *Rarely used functionalities.* Some APIs are outdated or only suitable for certain apps. For example, the `downloadMultimediaMessage` API is related to MMS (Multimedia Messaging Service). Currently, apps generally send messages over the Internet, not MMS.
- *APIs only for system apps.* Some APIs are designed for system apps, not third-party market apps. For example, invoking the `setCallComposerStatus` API requires the `MODIFY_PHONE_STATE` permission with a system signature, which third-party market apps cannot obtain.

Discussion. Many apps in third-party markets do not meet the API level requirements of Google Play. This is mainly due to two reasons. Firstly, some apps have not been updated for a long time. Secondly, even if some apps are updated regularly, they may not be developed for the latest SDK version. Since adapting to a new SDK version requires modifying the app's code, which needs extra development cost and time, developers prefer to rely on Android's forward compatibility rather than migrating their apps to the latest version. Besides, plenty of third-party apps use APIs-FC. Some APIs-FC are even necessary for nearly all apps. This leads to massive apps executing the code branches for lower versions in F-compatibility implementations, which may introduce security risks mentioned in RQ1 or prevent users from using new features provided in branches for high versions. Moreover, we found that some APIs-FC are not used due to the technical evolution. In such cases, these APIs could be removed from the framework or updated directly without considering the compatibility issue.

Answers to RQ3

Most third-party market apps (99.3%) target outdated SDKs (API level $\leq$ 32). 98.0% market apps use the deployed F-compatibility support of Android OS. Some APIs-FC are used widely among third-party apps, so their implementations would have a profound influence.

V. DISCUSSIONS

In this section, we discuss the potential limitations of our study and suggestions regarding forward compatibility issues.

Limitations. When constructing the Android system dataset, we only selected one release version for each Android API level. This is because the changes between different release versions at the same API level are small. However, using more release versions may be more precise.

Since static analysis does not execute the code, we cannot ensure that all of our discovered methods with F-compatibility support trigger F-compatibility control in practice.

During the measurement of the number of apps affected by F-compatibility implementation, we only considered the public SDK APIs in apps. Non-SDK APIs used by apps in special ways, such as double reflection [38], are neglected. Besides, when identifying APIs-FC, we restricted the number of a method's callers and the search depth to avoid state space explosion. Therefore, the affected app amount may not be entirely accurate.

Suggestions and Lessons Learned. After the above analysis, it is evident that Android's forward compatibility allows legacy apps to function appropriately. However, its implementation may also introduce potential security risks. Furthermore, Android's forward compatibility is essential for the functioning of massive apps on third-party markets, making its impact widespread. To address this issue, various parties must collaborate. In this regard, we suggest some recommendations for developers and third-party markets.

- For OS developers, F-compatibility implementations can be used as a transitional means to give app developers a period of time to adapt. After this period, system developers can delete or hide the F-compatibility implementations or rewrite the code of the lower version branch. Therefore, the functions that the legacy app intends to use will be invalidated, prompting app developers to use new features.
- For app developers, they should update their target API levels timely and implement backward compatibility in their apps if they want to run smoothly on all Android versions.
- For third-party app markets, we recommend that they follow the same requirements as Google Play, which restricts the `targetSdkVersion` attribute of apps submitted by developers and makes legacy apps invisible to users. This action can effectively eliminate legacy apps to avoid security risks from F-compatibility implementations.

VI. Related Work

Plenty of work studied the compatibility issue caused by the fragmentation of Android OS. However, most research focused on compatibility issues in apps, and rare work noticed the implementations of app F-compatibility guarantee provided by Android OS.

Android Forward Compatibility. As mentioned in Section II-B, Android ensures that legacy apps are forward-compatible with the new Android versions by two strategies: 1) using the `@Deprecated` annotation to mark the methods that are expected to be removed, and 2) using the app's `targetSdkVersion` attribute to perform a fine-grained control. For the first strategy, Li et al. [29] proposed CDA and applied it to multiple versions of Android framework code for characterizing how APIs are deprecated in practice. For the second strategy, Mutchler et al. [33] manually discovered several F-compatibility implementations based on the `targetSdkVersion` attribute while studying the security of legacy apps targeting outdated API levels. However, this strategy's overall status has not been extensively explored by the current Android research community, such as its implementation, implication, and evolution. Our work fills this research gap.

App Compatibility Issues. The first work studying the app compatibility problem caused by Android fragmentation was conducted by Wei et al. [35]. They manually analyzed 191 real-world compatibility problems in open-source apps and developed FicFinder to detect compatibility issues in apps automatically. Li et al. [28] proposed CiD to model the Android API lifecycle systematically and then analyzed the app bytecode to investigate compatibility issues due to Android evolution. He et al. [26] developed IctApiFinder to detect incompatible API usages in Android apps using a context-sensitive inter-process data flow analysis framework. Huang et al. [27] implemented an empirical study on callback compatibility issues and proposed Cider to detect this kind of issue in apps. Liu et al. [31] studied compatibility issues caused by silently evolving code in the Android source code. Other work related to app compatibility issue detecting includes [36], [34], [22], [32], and [30]. To fix app compatibility issues, Fazzini et al. [23] developed AppEvolve which can automatically perform app updates for API changes. Based on AppEvolve, Haryono et al. successively proposed CocciEvolve [24] and AndroEvolve [25] to conduct automatic deprecated-API usage update for apps. In addition, Xia et al. [37] combined static analysis and machine learning to develop RAPID, which examines developers' handling of evolution-induced API compatibility issues in Android apps on a large scale.

Unlike the above research focus on detecting and fixing apps' compatibility issues introduced by Android fragmentation, our work studies the implementation of F-compatibility promised by Android OS, especially through using the app's `targetSdkVersion` attribute.

VII. Conclusion

In this work, we systematically studied the forward compatibility implementations of the Android OS, with a primary focus on the `targetSdkVersion`-based fine-grained control. Specifically, we design three research questions covering the implementations, implications, and evolution of forward compatibility support in Android. We addressed these questions through analysis based on large-scale datasets and targeted approaches. This study offers insights into how Android ensures forward compatibility and traces the evolution of the Android framework design. It also quantifies the relaxation of restrictions for apps targeting lower versions in the context of forward compatibility. Furthermore, our findings can aid framework developers in refining their forward compatibility policies and assist third-party markets in evaluating the extent to which their apps leverage forward compatibility.

ACKNOWLEDGEMENTS

This work was supported by Taishan Young Scholar Program of Shandong Province, China (Grant No. tsqn202211001), Shandong Provincial Natural Science Foundation (Grant No. ZR2023MF043), and Xiaomi Young Talents Program.

REFERENCES

[1] (2017) Request: remove new restriction of Android 8.1 : "Only fullscreen activities can request orientation". [Online]. Available: https://issuetracker.google.com/issues/68454482?pli=1

[2] (2018) New fatal crash on ad display. [Online]. Available: https://groups.google.com/g/google-admob-ads-sdk/c/Yniv0UNCc74

[3] (2023) Androguard. [Online]. Available: https://androguard.readthedocs.io/en/latest/

[4] (2023) Android API Levels. [Online]. Available: https://apilevels.com/

[5] (2023) App compatibility in Android. [Online]. Available: https://developer.android.com/guide/app-compatibility

[6] (2023) Application back compatibility. [Online]. Available: https://developer.android.com/guide/topics/manifest/uses-sdk-element#bc

[7] (2023) Application forward compatibility. [Online]. Available: https://developer.android.com/guide/topics/manifest/uses-sdk-element#fc

[8] (2023) bindService. [Online]. Available: https://developer.android.com/guide/components/intents-filters

[9] (2023) Compatibility framework tools. [Online]. Available: https://developer.android.com/guide/app-compatibility/test-debug

[10] (2023) getRecentTasks. [Online]. Available: https://developer.android.com/reference/android/app/ActivityManager#getRecentTasks(int,%20int)

[11] (2023) Intent. [Online]. Available: https://developer.android.com/reference/android/content/Intent

[12] (2023) Meet Google Play's target API level requirement. [Online]. Available: https://developer.android.com/google/play/requirements/target-sdk

[13] (2023) Mobile Operating System Market Share Worldwide. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[14] (2023) Runtime Permissions. [Online]. Available: https://source.android.com/docs/core/permissions/runtime_perms

[15] (2023) Service. [Online]. Available: https://developer.android.com/reference/android/app/Service

[16] (2023) Service Hijacking. [Online]. Available: https://capec.mitre.org/data/definitions/499.html

[17] (2023) Standard partitions. [Online]. Available: https://source.android.com/docs/core/architecture/partitions

[18] (2023) Target API level requirements for Google Play apps. [Online]. Available: https://support.google.com/googleplay/android-developer/answer/11926878

[19] (2023) uses-sdk-element. [Online]. Available: https://developer.android.com/guide/topics/manifest/uses-sdk-element

[20] (2023) WALA. [Online]. Available: https://github.com/wala/WALA

[21] (2023) Why target newer SDKs? [Online]. Available: https://developer.android.com/google/play/requirements/target-sdk

[22] H. Cai, Z. Zhang, L. Li, and X. Fu, "A Large-Scale Study of Application Incompatibilities in Android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Beijing, China, July 15-19, 2019*, 2019.

[23] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage Update for Android Apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Beijing, China, July 15-19, 2019*, 2019.

[24] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example," in *Proceedings of the 28th International Conference on Program Comprehension (ICPC), Seoul, Republic of Korea, July 13-15, 2020*, 2020.

[25] S. A. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano, and G. Muller, "AndroEvolve: Automated Update for Android Deprecated-API Usages," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, Spain, May 25-28, 2021*, 2021.

[26] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and Detecting Evolution-Induced Compatibility Issues in Android Apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), Montpellier, France, September 3-7, 2018*, 2018.

[27] H. Huang, L. Wei, Y. Liu, and S. Cheung, "Understanding and Detecting Callback Compatibility Issues for Android Applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), Montpellier, France, September 3-7, 2018*, 2018.

[28] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Amsterdam, The Netherlands, July 16-21, 2018*, 2018.

[29] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising Deprecated Android APIs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden, May 28-29, 2018*, 2018.

[30] P. Liu, M. Fazzini, J. C. Grundy, and L. Li, "Do customized android frameworks keep pace with android?" in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, 2022.

[31] P. Liu, L. Li, Y. Yan, M. Fazzini, and J. C. Grundy, "Identifying and characterizing silently-evolved methods in the android API," in *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, 2021.

[32] T. Mahmud, M. Che, and G. Yang, "Android Compatibility Issue Detection Using API Differences," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, March 9-12, 2021*, 2021.

[33] P. Mutchler, Y. Safaei, A. Doupé, and J. C. Mitchell, "Target Fragmentation in Android Apps," in *Proceedings of the 2016 IEEE Security and Privacy Workshops, San Jose, CA, USA, May 22-26, 2016*, 2016.

[34] S. Scalabrino, G. Bavota, M. Linares-Vásquez, M. Lanza, and R. Oliveto, "Data-Driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study," in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR), 26-27 May 2019, Montreal, Canada*, 2019.

[35] L. Wei, Y. Liu, and S. Cheung, "Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, September 3-7, 2016*, 2016.

[36] ——, "PIVOT: Learning API-Device Correlations to Facilitate Android Compatibility Issue Detection," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE), Montreal, QC, Canada, May 25-31, 2019*, 2019.

[37] H. Xia, Y. Zhang, Y. Zhou, X. Chen, Y. Wang, X. Zhang, S. Cui, G. Hong, X. Zhang, M. Yang, and Z. Yang, "How Android Developers Handle Evolution-induced API Compatibility Issues: A Large-scale Study," in *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE), Seoul, South Korea, 27 June - 19 July, 2020*, 2020.

[38] S. Yang, R. Li, J. Chen, W. Diao, and S. Guo, "Demystifying Android Non-SDK APIs: Measurement and Understanding," in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, May 25-27, 2022*, 2022.