Demystifying Android Non-SDK APIs: Measurement and Understanding

Shishuai Yang[1,2], Rui Li[1,2], Jiongyi Chen[3], Wenrui Diao[1,2(✉)], and Shanqing Guo[1,2]
[1]School of Cyber Science and Technology, Shandong University
{shishuai,leiry}@mail.sdu.edu.cn, {diaowenrui,guoshanqing}@sdu.edu.cn,
[2]Key Laboratory of Cryptologic Technology and Information Security of Ministry of Education, Shandong University
[3]National University of Defense Technology, jiongyi_chen@126.com

ABSTRACT

During the Android app development, the SDK is essential, which provides rich APIs to facilitate the implementations of functionalities. However, in the Android framework, there still exist plenty of non-SDK APIs that are not well documented. These non-SDK APIs can be invoked through unconventional ways, such as Java reflection. On the other hand, these APIs are not stable and may be changed or even removed in future Android versions, providing no guarantee for compatibility. From Android 9 (API level 28), Google began to strictly restrict the use of non-SDK APIs, and the corresponding checking mechanism has been integrated into the Android OS.

In this work, we systematically study the use and design of Android non-SDK APIs. Notably, we propose four research questions covering the restriction mechanism, the present usage status, malicious usage, and the API list evolution. To answer these questions, we conducted a large-scale measurement based on over 200K apps and the source code of three recent Android versions. As a result, a series of exciting and valuable findings are obtained. For example, Google's restriction is not strong enough and can still be bypassed. Besides, app developers use only a tiny part of non-SDK APIs. Our work provides new knowledge to the research community and can help researchers improve the Android API designs.

CCS CONCEPTS

• **Software and its engineering → Software reliability**.

KEYWORDS

Android Non-SDK APIs; API Design and Evolution

ACM Reference Format:
Shishuai Yang, Rui Li, Jiongyi Chen, Wenrui Diao, and Shanqing Guo. 2022. Demystifying Android Non-SDK APIs: Measurement and Understanding. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510045

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510045

1 INTRODUCTION

As the most popular mobile operating system, Android occupies more than 72% of the global smartphone market share [9]. At the end of 2020, close to 3.15 million apps are available for users to download in Google Play [11]. Such a large number of apps benefit from the Android SDK provided by Google. Developers can quickly build their apps through the rich APIs provided by the SDK and then distribute them on the app market.

During the app development, the APIs invoked by developers are derived from the `android.jar` library. On the other hand, at the app execution stage, the APIs referred by apps are dynamically linked to the implementations of the `framework.jar` library, a runtime library located in the Android OS. Though both `android.jar` and `framework.jar` are constructed based on the source code of AOSP (Android Open Source Project), a large proportion of APIs in `framework.jar` do not exist in `android.jar`, called *non-SDK APIs*.

Non-SDK APIs are not documented in the Android framework Package Index [12], and they could be internal APIs, restricted APIs, or hidden APIs (see Section 2.1 for details). These APIs are not stable and may be changed or even disappear in future Android versions, providing no guarantee for compatibility. Therefore, by design, non-SDK APIs cannot be invoked by developers directly. In practice, the well-known programming tricks for accessing them are using the Java reflection mechanism or replacing the official `android.jar` with a custom one containing the non-SDK APIs. Inevitably, using the unstable non-SDK APIs will cause compatibility issues, like app crashes and unexpected behaviors [32, 36, 45].

Obviously, the widespread use of non-SDK APIs is not a good phenomenon. Therefore, from Android 9 (API level 28), Google began to strictly restrict the invocations of non-SDK APIs and encouraged developers to use the public APIs provided by the SDK to build apps [15]. On the OS level, a new API invocation checking mechanism was introduced, and restricted non-SDK API lists were provided for reference officially. The current solution tries to push app developers to drop non-SDK APIs gradually, and the ideal situation is blocking the use of any non-SDK APIs in apps. However, such an ultimate aim cannot be easily reached, and there is still a lack of up-to-date knowledge on the evolution of Android non-SDK APIs. Their usage in the wild and the effectiveness of Google's restriction have not been systematically evaluated.

Our Work. In this work, we systematically investigate how developers use non-SDK APIs and the evolution of non-SDK APIs in the Android framework. Notably, we seek to answer the following research questions:

Usage of Developers:

⇒ **RQ1:** *Can Google's restriction on non-SDK APIs be bypassed?*

⇒ **RQ2:** *What is the present status of using non-SDK APIs in apps?*

Malware vs. Benign Apps:

⇒ **RQ3:** *What are the differences in using non-SDK APIs between malicious and benign apps?*

Evolution of Non-SDK APIs:

⇒ **RQ4:** *How did non-SDK APIs evolve in the Android framework?*

To answer these proposed research questions, we collected 226,209 apps from multiple app markets to analyze how and why developers use non-SDK APIs. We also constructed a malware dataset based on VirusTotal [43] to analyze the differences in using non-SDK APIs between malicious and benign apps. In addition, through analyzing the source code of recent Android versions (9, 10, and 11), we studied the evolution of non-SDK APIs in the Android framework. Also, to facilitate the measurement, we built a series of targeted lightweight analysis tools, such as `veridex++` for scanning non-SDK APIs used in apps.

Contributions. Here we list the main contributions of this paper:

- Large-scale Measurement. Based on over 200K APK files and the source code of three recent Android versions, we conducted a multi-dimension large-scale measurement on the Android non-SDK APIs in the wild.
- Systematic Study. We systematically studied the Android non-SDK APIs from the aspects of developers, malware, and system design. In particular, we proposed four significant research questions and answered them with enough supporting evidence. Here we give the corresponding short answers.
  (1) *The restriction of Google still can be bypassed through the double-reflection and call stack breaking techniques.*
  (2) *Non-SDK APIs are widely used in apps. The usage purpose is to achieve some app features not supported by the SDK.*
  (3) *Non-SDK APIs have been abused by malicious apps, for example, to dynamically load malicious code.*
  (4) *The adjustment of non-SDK APIs is mainly for security concerns and fine-grained functionality control.*

Both OS designers and the software engineering research community can benefit from this work. (1) For OS designers: Google can follow our guidelines to reformulate the policy of restricting access to non-SDK APIs, and change risky APIs with secure alternative ones. (2) For the research community: We provided multi-dimension large-scale measurement data on the Android non-SDK APIs. It reflects how app developers use APIs, and the analysis results could help design usable APIs in the future.

Roadmap. The rest of this paper is organized as follows. Section 2 provides the necessary background information to allow readers to understand this work better. Section 3 presents the experimental setup of this work. Section 4 details our empirical research to answer the above research questions. Section 5 discusses some mitigation measures to reduce the use of non-SDK APIs and the limitation of this work. Section 6 introduces related work, and Section 7 concludes this work.

2 BACKGROUND

In this section, we provide the necessary background of Android APIs and focus on non-SDK APIs.

2.1 Android APIs

API is short for application programming interface, which is a pre-defined function. Developers only need to call the corresponding API according to the convention defined by the interface without accessing the source code or understanding the details of the internal working mechanism. Android APIs provide developers with the ability to access system resources of Android devices. For example, they can access external storage through the `getExternalStorageDirectory()` API. The collection of these APIs constitutes the SDK. The Android SDK is almost completely closed and interacts with developers through APIs.

When developing apps, the APIs used by app developers are originated from `android.jar`[1] which is a library in the development environment. Corresponding to `android.jar` used for development, there is `framework.jar`[2], a runtime library in the Android system that contains more APIs, including non-SDK APIs. The relationship between `android.jar` and `framework.jar` is illustrated in Figure 1. Non-SDK APIs exist widely in the source code, usually in the following three forms, as shown in Listing 1. Internal APIs are not planned to be open to the outside world and are only for internal use by the system. Restricted and hidden APIs are to prevent developers from using some unstable APIs. These APIs may be removed from the Android framework or become public APIs in a specific Android version.

```
1  // Internal API
2  package com.android.internal.telephony.cdma;
3  public class SmsMessage extends SmsMessageBase
       {
4    @UnsupportedAppUsage
5    public static SmsMessage createFromPdu(byte[]
       pdu) { }
6  }
7  // Restricted API
8  public class Activity extends
       ContextThemeWrapper{
9    private void initWindowDecorActionBar() { }
10 }
11 // Hidden API
12 public class TelephonyManager {
13   /** Returns a constant indicating the state
       of the card apps on the default SIM card.
14   * @hide */
15   @SystemApi
16   public SimState int getSimApplicationState()
       { }
17 }
```

Listing 1: Example of non-SDK APIs.

- Internal APIs. These APIs are usually in the `com.android.internal` package. The source code of all classes under this package is invisible and only can be used by system apps.

[1]Location: `<SDK-dir>/platforms/android-X/android.jar`, `X` is the API level.
[2]Location: `/system/framework/framework.jar`

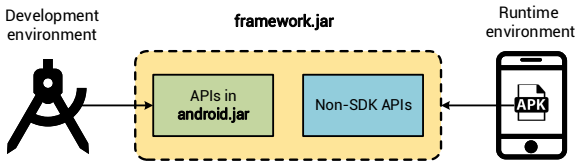

Figure 1: Relationship of `android.jar` and `framework.jar`.

This package provides the core functions of the Android system and can be used to access some sensitive resources.
- Restricted APIs. These APIs are labeled by the Java qualifiers `private` or `default`.
- Hidden APIs. These APIs are usually clearly marked with `@hide` in the Java doc.

As the Android OS evolves and new versions are released, each version is assigned a unique integer identifier, called the API level. Each Android device runs at exactly one API level, which precisely indicates the version of the API set that can be invoked by the apps running on the device. When an app is built, it contains the following API level information:

(1) `minSdkVersion` – the minimum API level required for the Android device to run this app. If the API level of the device is lower than the value specified by `minSdkVersion`, the device will prevent the user from installing this app.

(2) `targetSdkVersion` – the API level of the Android device at which this app expects to run. This information indicates that the app developer has tested this app against the target API level, and the system should not enable any compatibility behaviors to maintain this app's forward compatibility with the target version [17].

2.2 Restricted Non-SDK APIs

Access Method. Non-SDK APIs cannot be accessed directly through the SDK provided by Google. Before Android 9, when developing an app, the non-SDK APIs could be accessed through the following three ways:

(1) SDK replacement. Developers can use a custom `android.jar` (containing the non-SDK APIs implementations) to replace the default `android.jar`.

(2) Java reflection. Through Java reflection [18], developers can call the methods and properties of any object at runtime, even if these methods and properties are labeled by the Java qualifier `private`.

(3) Java Native Interface (JNI). In the Java layer, developers can call the native code by defining JNI functions. Further, they can access SDK and non-SDK APIs through specific APIs provided by NDK in the native code [31].

However, in Android 9, Google realized the problem of referencing a large number of non-SDK APIs in apps and began to restrict the use of non-SDK APIs [15]. The purpose of limiting the use of non-SDK interfaces is mainly to improve apps' stability further, prevent apps from crashing during runtime, and improve the experience of users and developers [6].

Table 1: APK dataset.

Type	Source	Amount (total)	Amount (filtered)
Benign Apps	APKPure, F-Droid, Anzhi, Baidu, Huawei, Xiaomi, ...	226,209	79,493
Malware	VirusTotal	10,029	10,029

Non-SDK API Lists. To minimize the impact of non-SDK APIs restrictions on development workflow, Google classified all APIs in the Android framework into four different lists and targeted different lists to enforce different restrictions [10], as follows:

- `blacklist` – *blocklist*. Regardless of the `targetSdkVersion` of the app, once the app tries to access the APIs in this list, a runtime crash will be triggered.
- `greylist-max-x` – *conditionally blocked*. The APIs in this list are usually expressed in the form of `greylist-max-x`, and `x` is the code name of Android. For example, `greylist-max-p` means that if the `targetSdkVersion` of the app is no more than Android P (i.e., API level 28), the app can still access the non-SDK APIs. However, if the `targetSdkVersion` is greater than Android P, a runtime crash will occur when the app accesses the non-SDK APIs.
- `greylist` – *unsupported*. The APIs in this list are currently not restricted and can still be used in apps.
- `whitelist` – *SDK APIs*. The APIs in this list can be freely used and are now supported as part of the officially documented Android framework.

The APIs in the first three lists (i.e., non-SDK APIs) are not officially supported and may be changed at any time without notice.

3 METHODOLOGY AND DATASET

To answer the proposed research questions, in this section, we illustrate our measurement approach and constructed datasets.

3.1 Methodology

As illustrated in Figure 2, on a high level, our measurement contains three main steps, as follows:

- APK Dataset Construction. First, we constructed the APK dataset used in the study, including benign apps and malicious ones.
- API List Generation. Based on the source code of AOSP, we built an automated app scanner `veridex++` and generated the non-SDK API lists.
- Multi-dimension Analysis. To answer the proposed research questions, we designed multiple kinds of targeted analysis based on the data constructed in the previous steps.

3.2 App Dataset Construction

In this work, the dataset used in our research is divided into two types, as shown in Table 1.

- Benign Apps. Since Google Play did not provide the app bulk downloading APIs anymore, we selected 18 popular app markets, including 9Apps, 2265, Anzhi, APKPure, Baidu, DownloadPCAPK, F-Droid, Gfan, Huawei, LapTopPCAPK,

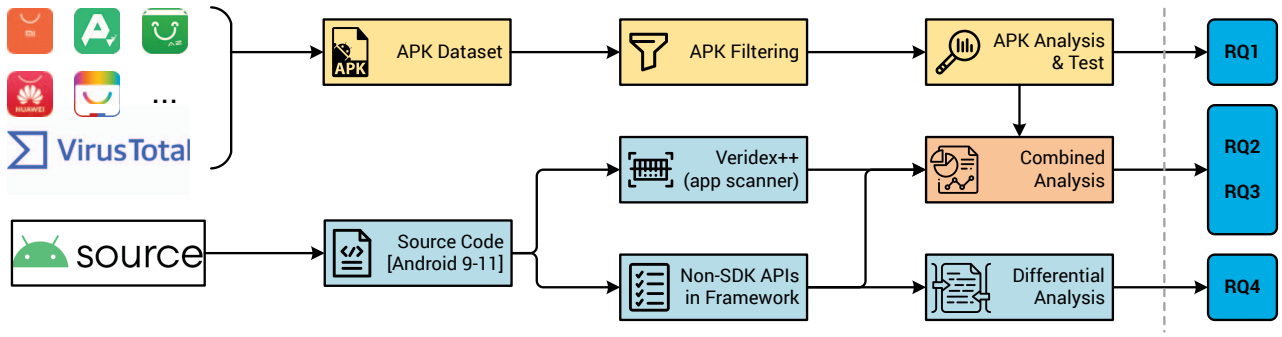

Figure 2: Overview measurement flow.

Lenovo, LePlay, Leyou, Flyme, PC6, Yingyongbao, Uptodown, and Xiaomi. After de-duplication, a total of 226,209 apps were obtained.

- Malware. We requested access to the VirusTotal dataset [43] and obtained a total of 10,029 malicious samples. VirusTotal claims that their provided malicious apps have been marked as "malicious" by at least 15 anti-virus engines.

Since we focus on the change before & after the launch of Google's restriction on non-SDK APIs (from API level 28), we filtered the apps with `targetSdkVersion` 27 and later versions. The filter implementation is based on Androguard [1]. Note that we did not filter malicious apps because most of their `targetSdkVersion` attributes are less than 28.

3.3 API List Generation

`Veridex++`. Starting from Android 9, in order to help developers detect non-SDK APIs used in their apps, Google provided a static detection tool – `veridex` [16]. This tool can be built by compiling the source code of AOSP. To suit apps with different `targetSdkVersion` attributes, we built three versions of `veridex` based on the source code of Android 9, 10, and 11 (corresponding to API level 28, 29, and 30).

In addition, we integrated these three versions of `veridex` into one tool and added the support of parallel execution for large-scale app scanning, called `veridex++`. We utilized this tool to scan apps to obtain the used non-SDK APIs. To further reflect the real intention of developers using non-SDK APIs, we excluded the non-SDK APIs in the official libraries used by apps. These libraries are support libraries required to run apps, usually containing specific characteristics, such as beginning with `com.google.*`, `com.android.*`, `androidx.*`, and `android.*`.

Non-SDK API Lists. Based on the source code of AOSP, through executing the corresponding source code compiling command[3], we obtained all non-SDK APIs corresponding to Android 9, 10, and 11 (API level 28, 29, and 30). For Android 12 (API level 31), the source code is currently not available, but Google provides independent non-SDK API lists for the Android 12 developer preview version.

[3]Command: `m out/soong/hiddenapi/hiddenapi-flags.csv`

3.4 Multi-dimension Analysis

To answer the proposed four research questions, we designed targeted analysis solutions. To RQ1, we tested various bypass approaches to validate their effectiveness and explored the restriction implementation. To RQ 4, the analysis is mainly based on comparing different versions of non-SDK lists. To RQ 2 and 3, we combined multiple data sources to measure the status. The detailed analysis methods are given by research questions in Section 4. Note that data measurement is only a part of our study, and we also investigated the fundamental causes behind these statistics results.

4 FINDINGS

In this section, we summarize our empirical research results on the research questions proposed in Section 1.

RQ1. Can Google's restriction on non-SDK APIs be bypassed?

As mentioned in Section 2.2, there exist some approaches (SDK replacement, Java reflection, and JNI) to access the non-SDK APIs. Through developing multiple Android apps, on Google Pixel 2 with Android 8, 9, 10, and 11, we tested the effectiveness of these approaches before/after Google launching the restriction.

Current Restriction. Based on our tests, the execution results of the mentioned bypassing methods on the `blacklist` are listed in Table 2. We can find that, before Google launching the restriction, these approaches can achieve access successfully. The difference is that only the Java reflection approach can access restricted APIs (the `private` and `default` APIs) because such an approach can access any object. After Google launching the restriction, all approaches cannot work and will trigger a runtime crash.

The execution results on the `greylist` and the `greylist-max-x` are consistent with Google's claim (as mentioned in Section 2.2). That is, there is no accessing restriction to the APIs on the `greylist`. When we tried to access the APIs on the `greylist-max-x`, if the `targetSdkVersion` of the app is less than or equal to the restricted level, these APIs can be accessed successfully. When the `targetSdkVersion` of the app is greater than the restricted level, it will cause a runtime crash.

Principle of Restriction. It seems that Google's strategy of restricting non-SDK APIs works well in Android 9 and later versions. We further investigated the implementation of such a restriction. It

Table 2: The effectiveness of accessing non-SDK APIs on the `blacklist`.

Approaches	Before Google launching restriction			After Google launching restriction		
	Internal APIs	Restricted APIs	Hidden APIs	Internal APIs	Restricted APIs	Hidden APIs
Normal Invocation	×	×	×	×	×	×
SDK Replacement	√	×	√	×	×	×
Java Reflection	√	√	√	×	×	×
Java Native Interface	√	×	√	×	×	×

is deployed on the Android Runtime level. Inside the ART virtual machine, each API has a set of `access_flags` flag bits to express the attributes of the API [4]. For example, setting `access_flags` as `0x2` is to indicate that the API is `private`. There are some reserved bits in `access_flags`, which were not fully utilized before Android 9. In Android 9, these reserved bits can be used to identify which list each API belongs. When an API call triggered by an app at runtime enters the ART virtual machine, the ART virtual machine will first identify the caller's identity. If the API call is made by system apps, it will not be restricted. If not, this API's restriction level (i.e., the list that this API belongs to) will be identified according to the value of `access_flags` to restrict non-SDK API calls.

In the above restriction, the ART virtual machine performs two checks to prevent non-SDK API invocations: (1) Check the caller's identity (system or third-party app). (2) Check the restriction level of the invoked API.

Bypass Restriction. Since the execution logic of Google's restriction needs to check the API caller's identity, ***it can still be bypassed by destroying the integrity of the caller's identity***. There are two ways for developers to achieve this goal:

(1) Double-reflection. This is a Java way. Using the system class to reflect, we can change the caller's identity to be the system [5]. We first leverage reflection to obtain the reflection API, called the meta-reflection API. This meta-reflection API is loaded by the system class. Then, we use this meta-reflection API to reflect the call to the non-SDK API. At this time, the call to the non-SDK API will be considered a system call. In addition, there is a `setHiddenApiExemptions()` API (a non-SDK API) under the `VMRuntime` class that can be used to exempt a non-SDK API from the restriction. Combining the double-reflection with `setHiddenApiExemptions()`, all non-SDK APIs can still be accessed through the previous approaches (i.e., SDK replacement, Java reflection, and JNI).

(2) Call Stack Breaking. This is a JNI way. By breaking the call stack of the API, the ART virtual machine cannot identify the caller [14]. Specifically, through creating a new native thread and then attaching the new thread to the ART virtual machine, this new thread will be on a new call stack. Therefore, within this new thread, when invoking an API, this API call will occur within the new call stack, and the ART virtual machine will recognize that this call is made by a system component, not a third-party app.

RQ1 Finding

In Android 9 and later versions, app developers can still use double-reflection and call stack breaking to bypass Google's restriction and access non-SDK APIs.

Table 3: Statistics of apps using non-SDK APIs.

Target SDK version	Total apps	Apps using non-SDK APIs	Percentage
27 (Android 8.1)	13,043	11,157	85.5%
28 (Android 9)	39,839	35,237	88.4%
29 (Android 10)	21,407	19,098	89.2%
30 (Android 11)	5,204	4,581	88.0%

Table 4: Statistics of average non-SDK APIs usage per app.

Target SDK version	Apps using non-SDK APIs	Used non-SDK APIs amount	Average usage /app
27 (Android 8.1)	11,157	184,491	16.5
28 (Android 9)	35,237	737,467	20.9
29 (Android 10)	19,098	633,697	33.2
30 (Android 11)	4,581	167,460	36.6

RQ2. What is the present status of using non-SDK APIs in apps?

Next, we measured the usage of non-SDK APIs in the wild. As mentioned in Section 3.3, we obtained the usage of non-SDK APIs in apps utilizing `veridex++`. Through comparing the usage in different app versions, we analyzed the reaction of developers to Google's restriction. Note that, since there is no official non-SDK APIs lists for API level 27 (before Google launching the restriction), we used `veridex++` configured for the API level 28 to analyze apps with `targetSdkVersion` 27.

Overall Statistics. As listed in Table 3, ***using non-SDK APIs is quite common***, with over 85% of apps. Also, the usage percentages in apps with different `targetSdkVersion` are stable, from 85.5% to 89.2%, with a slight increase. Table 4 reflects that, ***to a single app, more and more non-SDK APIs are used after Google launching the restriction***. On average, the apps with `targetSdkVersion` 27 use 16.5 non-SDK APIs, and this number increases to 36.6 on `targetSdkVersion` 30.

To explore the causes behind the above statistics results, we conducted reverse analysis on various apps, especially those with multiple versions. Specifically, we decompiled them using Apktool [3] and performed differential analysis on their smali files. Finally, we discovered three main reasons:

(1) With the evolution of the Android OS, some APIs used in apps were adjusted to non-SDK API lists, but developers did not replace these APIs used in their apps in time. For example, some apps end the call by using the `endCall()` API. This API is a public API in Android 9 and was moved to the `greylist` in Android 10.

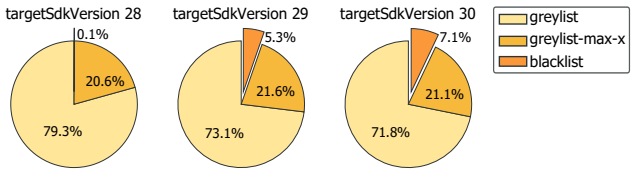

Figure 3: Distribution of non-SDK APIs referenced by apps.

Table 5: Statistics of unique non-SDK APIs used in apps.

Target SDK version	Total non-SDK APIs in lists	Unique used non-SDK APIs	Percentage
28 (Android 9)	141,735	4,346	3.1%
29 (Android 10)	285,463	4,398	1.5%
30 (Android 11)	315,872	2,194	0.7%

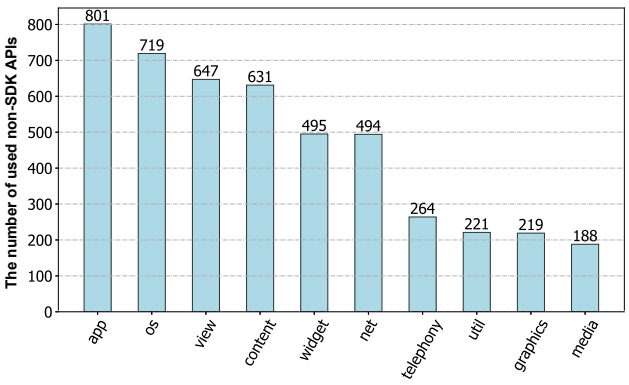

Figure 4: Top 10 belonged packages of non-SDK APIs.

(2) Some apps integrated third-party libraries containing non-SDK APIs. These libraries are updated with app updates, which may introduce new non-SDK APIs. For example, the app `Opera` integrates `Mintegral` [8], a third-party library for advertising SDK aggregation. Its updated version for `targetSdkVersion 29` uses the `getBatteryCapacity()` API (a non-SDK API) that does not exist in its previous versions.

(3) To increase the competitiveness of apps, the developer provides some unique functions through non-SDK APIs. For example, some developers utilize the non-SDK APIs under the `BatteryStatsHelper` class to monitor the battery usage of apps.

Usage by Lists. Next, we investigated what types of non-SDK APIs are used by app developers. We counted the number of non-SDK APIs referenced by apps on the `greylist`, `greylist-max-x`, and `blacklist` with `targetSdkVersion` 28, 29, and 30, as plotted in Figure 3. It shows that most of the used non-SDK APIs belong to the `greylist` because the `greylist` APIs have not been totally blocked and will not trigger runtime crashes. Some apps still use the APIs on the `blacklist`, and the proportion is gradually increasing.

As listed in Table 5, the proportion of non-SDK APIs used in apps accounts for a small part of the whole non-SDK APIs space (0.7% ∼ 3.1%), which means that ***developers are only interested in some specific functions of non-SDK APIs***.

Usage by Purposes. To further explore the developers' intentions of using non-SDK APIs, we located the belonged packages of these frequently used non-SDK APIs. According to the core keywords in the package names extracted by the NLP techniques, we filtered the top 10 belonged packages, as shown in Figure 4.

- `app`. This package contains high-level classes encapsulating the overall Android application model [21]. For example, developers can obtain the `ActivityThread` class instance through `currentActivityThread()` (`greylist`, 61,885 using times) in apps, which is the main thread of the current app. Then, they can further obtain the context, package name, and other app information through `ActivityThread`.
- `os`. This package provides essential system services, message passing, and inter-process communication [26]. For example, developers can use `getVolumeList()` (`greylist`, 17,024 using time) to return the path of all SD cards mounted on the device. However, the `getExternalStorageDirectory()` API provided in the SDK can not achieve this.
- `view`. This package provides the classes that expose the basic user interfaces handling screen layout and interaction with the user [29]. For example, the developer can count the number of user clicks in apps by hooking `mOnClickListener` (`greylist`, 8,355 using times) under the `mListenerInfo` (`greylist`, 13,604 using times) object.
- `content`. This package contains the classes for accessing and publishing data on the device [22]. For example, if developers need to implement the hot patch function, they can invoke `addAssetPath()` (`greylist`, 13,865 using times) to achieve the dynamic resource loading.
- `widget`. This package contains (mostly visual) UI elements to use on the app screen [30]. For example, when developers retrieve the width and height of a picture, if this picture does not set the width and height values, the values of `mMaxWidth` (`greylist-max-p`, 11,952 using times) and `mMaxHeight` (`greylist-max-p`, 11,895 using times) will be returned by default.
- `net`. This package assists network access, beyond the normal `java.net.*` APIs [25]. For example, developers can judge the current GPRS status through `getMobileDataEnable()` (`greylist`, 14,091 using times). If GPRS is turned on, they can perform the network traffic transmission tasks.
- `telephony`. This package provides APIs for monitoring basic phone information [27]. For example, developers can obtain the neighboring cell information of the device through `getNeighboringCellInfo()` (`greylist`, 3,796 using times) to locate the current device location.
- `graphics`. This package provides low-level graphics tools, such as canvases, color filters, points, and rectangles [23]. For example, developers usually use custom fonts in apps, and they need to load the font files and create fonts through `createFromFamiliesWithDefault()` (`greylist`, 26,019 using times).
- `util`. This package provides common utility APIs [28]. For example, suppose developers need to enable apps to adapt different device screens. In that case, they can use

`noncompatWidthPixels` (`greylist`, 20,251 using times) and `noncompatHeightPixels` (`greylist`, 18,472 using times) to obtain the device screen resolution.

- `media`. This package provides the classes that manage various media interfaces (audio and video) [24]. For example, if developers want to keep audio and video in sync, they usually need to use `getLatency()` (`greylist`, 2,724 using times) to obtain the track's estimated latency.

In summary, many popular app features cannot be supported by the Android SDK APIs, developers have to use non-SDK APIs.

RQ2 Finding

Using non-SDK APIs is very common in app developments, even after Google launching the restriction. App developers are only interested in a small part of non-SDK APIs, and the purposes are using unique features not supported by the SDK.

RQ3. What are the differences in using non-SDK APIs between malicious and benign apps?

The `targetSdkVersion` attributes of all malicious apps in our dataset are less than 28, so we used `veridex++` configured for API level 28 to scan them for generating the usage list of non-SDK APIs. The result shows that the usage of non-SDK APIs is widespread. In total, 61% of malware (6,150/10,029) used at least one non-SDK API.

Following the same approach, we scanned the benign apps with `targetSdkVersion` 28. We compared the results and selected the top 10 non-SDK APIs used in two kinds of apps for further analysis, as shown in Figure 5. It shows that the non-SDK APIs used by benign apps and malicious apps are quite different, and only four APIs (`currentActivityThread()`, `status_bar_height()`, `noncompatWidthPixels()`, and `noncompatHeightPixels()`) exist in both bar charts. Further checking shows ***malicious apps abuse non-SDK APIs to achieve malicious purposes***. Here we give two concrete cases.

Case Study 1. `getService()` is the most frequently used non-SDK API in malicious apps, which is designed for obtaining specific system services. In some usage cases, the malicious app invokes this API to obtain the `IPhoneSubInfo` interface [7], mainly responsible for querying SIM card information. Further, with `IPhoneSubInfo`, this app can use the hidden method – `getSubscriberId(int subId)` to obtain the IMSI (International Mobile Subscriber Identity) of the SIM card[4] according to `subId`. Finally, this app can guess the PIN (Personal Identification Number) code of the SIM card based on the IMSI. By default, it is usually the last six digits of the IMSI or 123456. The PIN code is essential and protects many significant functions of the SIM card, like network billing and internal information modification. With the PIN code, the malicious app can deduct fees without the user's consent. Part of the exploit code is listed in Listing 2.

```
1 public static String getPayPassword(String str,
      Context context){
2   if(!TextUtil.notNull(str)){
3     str = SIMUtil.getIMSI2(context);
4     }
5   return TextUtil.notNull(str)?str.substring(
      str.length()-6, str.length()):"123456";
6 }
```

Listing 2: Exploit case of PIN through `getService()`.

Case Study 2. Even if both benign and malicious apps use the same non-SDK API, their purposes are different. For example, `currentActivityThread()` can be used to obtain the hidden `ActivityThread` class instance. After an app gets `ActivityThread`, all APIs belonging to this class can be invoked by reflection, like the APIs for obtaining the process name and the app package name.

For the malicious usage (as demonstrated in Listing 3), an app can further obtain the `mPackages` instance in `ActivityThread` through the reflection and set its `mClassLoader` parameter to a custom `DexClassLoader` object referring to a malicious DEX file (`Line 9`). When this malicious DEX file is loaded, the app achieves dynamic malicious code loading to bypass the anti-virus detection.

For benign usage, the primary purpose is to improve the user experience. In Android 9, if an app uses non-SDK APIs, a dialog box will pop up at runtime to indicate that there exists an API compatibility problem. As a solution, the app can set the `mHiddenApiWarningShown` parameter in `ActivityThread` to `true` to prevent displaying this warning dialog box.

```
1  public void attachBaseContext(Context context){
2    ...
3    Class<?> cls = Class.forName("android.app.
       ActivityThread");
4    Method method = cls.getMethod("
       currentActivityThread", new Class[0]);
5    ...
6    DexClassLoader dexClassLoader = new
       DexClassLoader(String.valueOf(absolutePath2
       ) + "/code.dex", absolutePath, "/data/data/
       " + context.getPackageName() + "/lib/",
       context.getClassLoader().getParent());
7    Field declaredField2 = Class.forName("android
       .app.LoadedApk").getDeclaredField("
       mClassLoader");
8    declaredField2.setAccessible(true);
9    declaredField2.set(((WeakReference) ((Map)
       declaredField.get(invoke)).get(
       getPackageName()))).get(), dexClassLoader);
10 }
```

Listing 3: Malicious usage of `currentActivityThread()`.

RQ3 Finding

Non-SDK APIs have been abused for malware to achieve malicious purposes. Even for the same APIs, the usages of malicious and benign apps are usually different.

RQ4. How did non-SDK APIs evolve in the Android framework?

The above analysis shows there are still a large number of apps using non-SDK APIs, and intrusive changes to non-SDK APIs will cause extensive app crashes. Therefore, it is essential to understand

[4]If the current device is a dual-mode phone, it will return the IMSI of the main card.

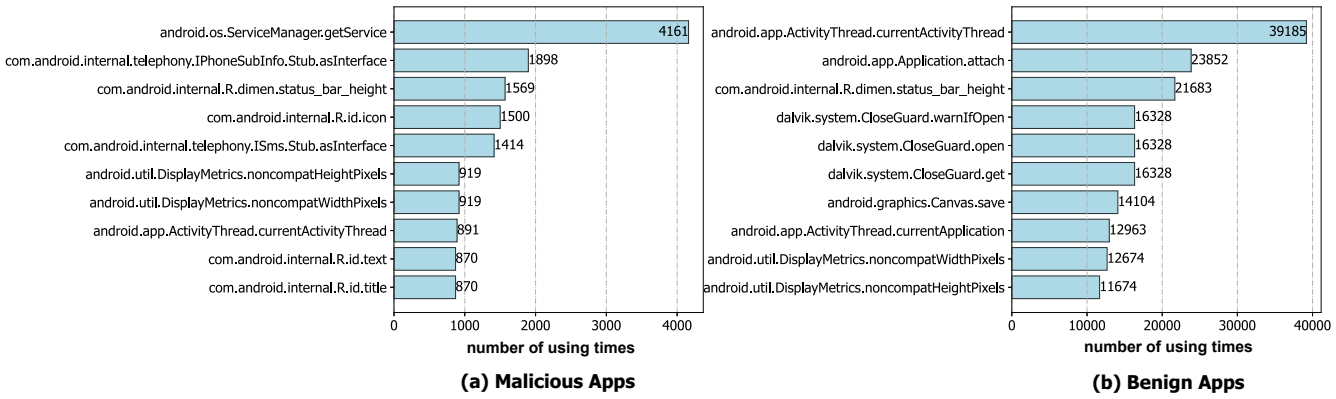

Figure 5: Top 10 non-SDK APIs used by malicious and benign apps.

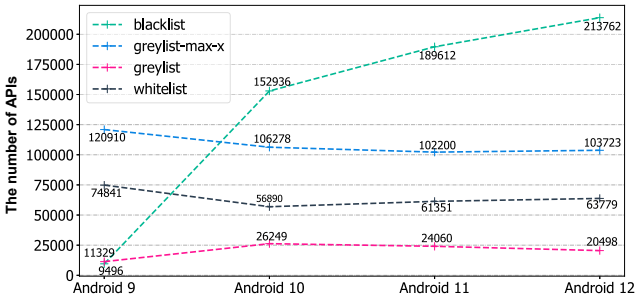

Figure 6: Changes in the amount of non-SDK APIs.

the evolution of non-SDK APIs in the Android framework. As mentioned in Section 3.3, we obtained the non-SDK API lists of Android 9, 10, 11, and 12 (API level 28, 29, 30, and 31). Further, we conducted a series of API list evolution analyses.

Restricted APIs Amount. The number of APIs in each list of the four versions is shown in Figure 6. It shows that Android 9 has fewer restricted APIs than other versions, especially the APIs on the `blacklist`. However, the APIs on the `blacklist` have increased significantly in later versions, especially from Android 9 to 10, which has increased by about 16 times. The number of APIs in other lists has remained relatively stable. Since Android 9 is the first version that Google began to implement restrictions on, if many APIs are directly added to the `blacklist`, it will cause API compatibility issues for a large number of apps.

API List Adjustments. To characterize the evolution of non-SDK API lists, we recorded the changes in the number of APIs in these lists between the adjacent Android versions, as shown in Table 6. In the adjustment of restricted API lists: (1) If a new non-SDK API is added to the Android framework, it will be highly likely added to the `blacklist` directly. The reason is that these newly added APIs have not been used by any apps, which can directly serve the purpose of restricting developers' access. (2) The APIs on the `blacklist` and `greylist-max-x` are more likely to be removed from the Android framework than the `greylist`. It is because these two types of non-SDK APIs are used relatively infrequently. Also, their functions are more relevant to user privacy. For example, `notifyLocationChanged()` is on the `blacklist` in Android 10 and was removed in Android 11. Using this API can notify the cell location change of the device. (3) Only a few APIs will be moved to the list with lower restriction levels or become public APIs. The APIs are usually moved to the list with the adjacent restriction level. For example, more APIs on the `greylist` are moved to the `greylist-max-x` rather than the `blacklist`. Since the APIs on the `greylist` are still used by apps, moving them directly to the `blacklist` will cause numerous app crashes at runtime. Moving them to the `greylist-max-x` not only ensure that apps using these APIs can still run on old devices, but also achieve the purpose of restricting the use of these APIs on the latest version of devices.

Reasons of Adjustments. We further explored that, in the list adjustment, why some APIs are further restricted (e.g., `greylist` → `greylist-max-x`), and some APIs are loosened to the lower restriction lists (e.g., `greylist` → `whitelist`). As a preliminary, we first investigated which kinds of non-SDK APIs were adjusted. Through filtering the keywords of names of packages that each non-SDK API belongs to, we obtained the most frequently adjusted features (packages) and the number of associated APIs in the process of list adjustment, as shown in Table 7. It shows that the APIs belonging to the Android telephony framework are more likely to be affected. Telephony is related to various system core services, such as CALL services, SMS services, and APN access points.

For the case of enhancing the restriction, the first reason is the ***security concern***. These APIs may bypass privacy protection and infringe user privacy. For example, `deleteMessageFromIcc()` is on the `greylist` in Android 10 and was moved to the `blacklist` in Android 11. Text messages are stored on the SIM card by default, not on the Android device. Using this API can delete all text messages stored on the SIM card, causing permanent loss of user data. Also, for the same reason, various newly added telephony-related APIs are directly added to the `blacklist`.

Besides, adding some APIs to the restricted list can achieve a more ***fine-grained access control of the API's functions***. For example, `finish()` is an overloaded function in the `android.app.Activity` class used to close the activity and has two declarations

Table 6: Changes of the amount of non-SDK APIs by list type between adjacent Android versions.

List Changes	Android 9 → Android 10	Android 10 → Android 11	Android 11 → Android 12
new → blacklist	145,103	53,802	38,954
new → greylist-max-x	5	27	28
new → greylist	16,966	126	30
blacklist → remove	1,847	17,979	14,790
greylist-max-x → remove	15,195	4,912	1,573
greylist → remove	1,478	488	506
blacklist → greylist-max-x	0	0	0
blacklist → greylist	0	2	36
blacklist → whitelist	0	119	97
greylist-max-x → blacklist	1	1	7
greylist-max-x → greylist	121	4	0
greylist-max-x → whitelist	111	83	23
greylist → blacklist	168	853	14
greylist → greylist-max-x	791	895	3,098
greylist → whitelist	197	98	10
whitelist → blacklist	15	1	1
whitelist → greylist-max-x	0	0	0
whitelist → greylist	467	7	0

†: new represents the number of APIs newly added to the list, and remove represents the number of APIs removed from the list.

Table 7: The most affected features during the list change process and the number of related APIs.

List Changes	Android 9 → Android 10	Android 10 → Android 11	Android 11 → Android 12
new → blacklist	(telephony, 25,218)	(telephony, 7,690)	(telephony, 6,759)
new → greylist-max-x	(telephony, 1)	(app, 5)	(view, 12)
new → greylist	(apache, 8,496)	(telephony, 60)	(libcore, 13)
blacklist → remove	(media, 395)	(telephony, 5,138)	(telephony, 2,727)
greylist-max-x → remove	(media, 2,124)	(net, 567)	(app, 234)
greylist → remove	(util, 351)	(telephony, 175)	(provider, 108)
blacklist → greylist-max-x	N/A	N/A	N/A
blacklist → greylist	N/A	(telephony, 1)	(bluetooth, 35)
blacklist → whitelist	N/A	(telephony, 45)	(telephony, 30)
greylist-max-x → blacklist	(system, 1)	(telephony, 1)	(media, 4)
greylist-max-x → greylist	(os, 53)	(telephony, 3)	N/A
greylist-max-x → whitelist	(media, 21)	(telephony, 29)	(graphics, 5)
greylist → blacklist	(hardware, 55)	(app, 127)	(graphics, 10)
greylist → greylist-max-x	(telephony, 125)	(R, 702)	(telephony, 915)
greylist → whitelist	(media, 75)	(util, 58)	(media, 3)
whitelist → blacklist	(os, 11)	(os, 1)	(telephony, 1)
whitelist → greylist-max-x	N/A	N/A	N/A
whitelist → greylist	(util, 107)	(provider, 7)	N/A

†: N/A means that no features are affected.

(with parameters or not), as shown in Listing 4. Note that `finish()` is a publicly accessible API. However, `finish(int finishTask)` belongs to the `greylist` before Android 12 and was moved to the `greylist-max-x` in Android 12. In fact, the internal implementation of `finish()` calls `finish(int finishTask)` and passes in a default parameter – `DONT_FINISH_TASK_WITH_ACTIVITY` (Line 4–6), which means closing the activity without closing the stack. When the activity to be closed is not at the bottom of the stack, if the app calls `finish(int finishTask)` rather than `finish()` and passes in the `FINISH_TASK_WITH_ROOT_ACTIVIT` parameter, this app will exit unexpectedly.

```
1 public static final int
      DONT_FINISH_TASK_WITH_ACTIVITY = 0;
2 public static final int
      FINISH_TASK_WITH_ROOT_ACTIVITY = 1;
3 public static final int
      FINISH_TASK_WITH_ACTIVITY = 2;
4 public void finish(){
5   finish(DONT_FINISH_TASK_WITH_ACTIVITY);
6   }
7 private void finish(int finishTask){ ... }
```

Listing 4: Example of the API being further restricted.

The restriction on `finish(int finishTask)` (and other similar APIs) is beneficial to less experienced developers, preventing them from passing in wrong parameters and causing unexpected consequences.

The main reason for loosening restrictions may be the requests from app developers. Since some developers must use the restricted APIs to implement specific functions, they applied to Google for releasing some APIs (through the Android Issue Tracker [13]). On the other hand, it is also related to the new features added in the new versions of the Android OS. For example, from Android 9 to 10, many media-related APIs under the `android.media.tv` package were moved from the `greylist` to `whitelist`. The corresponding is that, in Android 10, Google brought many improvements and changes to the Android TV, including security & privacy, media & graphics, dynamic partitions, and energy consumption/doze [2].

RQ4 Finding

In the evolution of Android OS, the non-SDK API lists were adjusted frequently. The reasons for enhancing API restrictions are mainly for security concerns and fine-grained control. The reasons for loosening API restrictions are mainly due to developer requests and newly added features.

5 DISCUSSIONS

In this section, we discuss some limitations of this work and give suggestions for reducing the use of non-SDK APIs and improving the efficiency of malware detection.

Limitations. In this study, our basic data of non-SDK APIs used in apps relies on the `veridex` tool provided by Google. However, `veridex` cannot detect JNI calls, and the detection of reflection is not 100% accurate. On the other hand, dynamic analysis is not a suitable solution for the massive number of apps in our dataset due to efficiency and coverage issues. Therefore, compared with the dynamic analysis approach, we developed `veridex++` based on `veridex`, a static analysis solution for the large-scale app scanning.

We used `veridex++` configured for API level 28 to scan apps with `targetSdkVersion` attributes less than 28. It may cause false positives because, for some non-SDK APIs in API level 28, when these apps used them, these APIs might still be SDK APIs. For the top 10 non-SDK APIs used by malicious apps, we manually excluded such false positives.

The number of apps with `targetSdkVersion 30` is relatively small (i.e., 5,204) because it is the latest Android API level (when conducting this work), and the mainstream Android devices on the market are still Android 10 (API level 29). Therefore, considering the future versions of released apps, some basic data may fluctuate, like Table 3, Table 4, and Table 5. Also, to further understand the developers' intentions of using non-SDK APIs, large-scale surveys/interviews on developers could be conducted.

Restriction Suggestions. According to the above analysis, most developers may ignore Google's restriction, resulting in an increase in using non-SDK APIs in recent versions of apps. Here we propose some suggestions for balancing the requirements of developers and app compatibility.

(1) For app development, when developers invoke non-SDK APIs, Android Studio should prompt a reminder about the consequence of accessing the API or provide an alternative to the public APIs with similar functions. Also, an integrity check mechanism of SDK should be added to Android Studio. So it can detect whether the `android.jar` library has been replaced and resets it to the official version.
(2) For app runtime, Google needs to further cut off the way to access non-SDK APIs, such as adding a mechanism to identify double-reflection calls. Also, at the native layer, apps should be prohibited from creating new native threads to prevent them from bypassing the restriction by breaking the function call stack.
(3) For API design, the widely used non-SDK APIs should be released or redesigned for public usage. Google could use the API usage data and the developer requests as references.

Malware Detection. Malware detection is a popular topic in the mobile security research community. The current mainstream malware detection technology is to build a classifier through machine learning based on a series of features extracted from apps, such as requested permissions and API calls [19, 34]. The accuracy of identifying malware basically depends on the extracted features. That is, the selected features should have significant differences between malicious and benign apps. This research shows that non-SDK APIs meet this requirement because there exist significant API usage differences (see Figure 5). In addition, the app behaviors are entirely different for the same APIs, and the corresponding API call sequences are also entirely different. Therefore, we recommend using non-SDK APIs as an essential feature when building malware classifiers to improve malware detection accuracy.

6 RELATED WORK

The Android API has been studied by plenty of previous work. However, most research focused on public APIs, and rare work noticed the non-SDK APIs in Android. In this section, we review the related work on Android APIs.

Non-SDK APIs. The most relevant work to non-SDK API was conducted by Li et al. [35]. They empirically investigated the evolution of Android internal APIs and hidden APIs from the aspects of significance, impact, and adoption. However, their work was based on early versions of Android ($\leq$ 6.0) before Google deployed the restriction on non-SDK APIs. The relevant knowledge needs to be updated to reflect the current status. Also, at that time, since Google did not provide the complete list of non-SDK APIs at each API level, the corresponding data analysis may not be accurate (may exist false positive or false negative).

Unlike the above research, in this paper, we mainly focus on the usage of non-SDK APIs in the Android framework after Google deployed access restrictions on non-SDK APIs. To the best of our knowledge, we are the first to conduct a systematic study of non-SDK APIs after Google implemented the restriction.

API Evolution. Various previous work studied the evolution of Android APIs. McDonnell et al. [42] found that Android is evolving fast at a rate of 115 API updates per month on average. However, developers cannot adopt these newly added APIs in

time. Linares-Vásquez et al. [38] analyzed the relation between the success of apps and the change- and fault-proneness of the underlying APIs. Their study shows that less fault- and change-prone APIs contribute more to apps' successes. In addition, they also investigated a relationship between API changes and developers' reactions [39]. Li et al. [37] performed an exploratory study of deprecated Android APIs based on a prototype research tool called CDA and discovered that the Android framework codebase is regularly cleaned up from deprecated APIs in a short period. More recently, Liu et al. [40] conducted an empirical study to characterize silently-evolved methods across ten versions of the Android API. These methods are functions whose behavior might have changed, but the corresponding documentation did not change accordingly.

Compatibility Issues. API evolution causes compatibility issues in Android apps [44]. Li et al. [36] proposed CiD to detect API-related compatibility issues based on an API lifecycle model. He et al. [32] studied compatibility problems induced by the evolution of Android OS. They developed IctApiFinder to discover incompatible API usage issues in Android apps based on inter-procedural data-flow analysis. Huang et al. [33] identified the callback compatibility issues induced by callback API evolution and devised Cider to detect this kind of issue. Cai et al. [20] empirically investigated the app incompatibilities that are actually exercised at runtime and found that runtime incompatibilities are mostly due to API changes during SDK evolution. Xia et al. [45] focused on the Android developers' reactions to evolution-induced API compatibility issues. Their research shows that developers do not want to provide alternative implementations for incompatible API invocations. More recently, Mahmud et al. [41] introduced ACID, which leverages API differences and static analysis of source code of Android apps to detect compatibility issues caused by API evolution.

7 CONCLUSION

Since Android 9, Google began to restrict access to non-SDK APIs to improve apps' stability. In this paper, we conducted the first large-scale study on the use and design of non-SDK APIs. We first explored the implementation of Google's restriction. Then, we investigated the present usage trend and purposes of non-SDK APIs. Next, we compared their usage in malicious apps. Finally, we characterized the evolution of non-SDK APIs in the Android framework. A series of exciting and valuable findings are obtained, which provides new knowledge to the research community and can help researchers improve Android APIs' design.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (Grant No. 61902148) and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] 2021. *Androguard.* Retrieved August 24, 2021 from https://github.com/androguard/
[2] 2021. *Android 10 on Android TV.* Retrieved August 24, 2021 from https://android-developers.googleblog.com/2019/12/android-10-on-android-tv.html
[3] 2021. *Apktool.* Retrieved August 24, 2021 from https://ibotpeaches.github.io/Apktool/
[4] 2021. *Dalvik Executable format.* Retrieved August 24, 2021 from https://source.android.com/devices/tech/dalvik/dex-format
[5] 2021. *FreeReflection.* Retrieved August 24, 2021 from https://github.com/tiann/FreeReflection/
[6] 2021. *Improving Stability by Reducing Usage of non-SDK Interfaces.* Retrieved August 24, 2021 from https://android-developers.googleblog.com/2018/02/improving-stability-by-reducing-usage.html
[7] 2021. *IPhoneSubInfo.* Retrieved August 24, 2021 from https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/telephony/java/com/android/internal/telephony/IPhoneSubInfo.aidl
[8] 2021. *Mintegral SDK.* Retrieved August 24, 2021 from https://www.mintegral.com/cn/sdk/
[9] 2021. *Mobile Operating System Market Share Worldwide - March 2021.* Retrieved August 24, 2021 from https://gs.statcounter.com/os-market-share/mobile/
[10] 2021. *Non-SDK API lists.* Retrieved August 24, 2021 from https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces#list-names
[11] 2021. *Number of Mobile Apps in Leading App Stores Worldwide in 2020.* Retrieved August 24, 2021 from https://financesonline.com/number-of-apps-in-leading-app-stores/
[12] 2021. *Package Index.* Retrieved August 24, 2021 from https://developer.android.com/reference/packages
[13] 2021. *Public Java APIs Requests.* Retrieved August 24, 2021 from https://issuetracker.google.com/issues?q=Public%20Java%20APIs%20Requests
[14] 2021. *RestrictionBypass.* Retrieved August 24, 2021 from https://github.com/ChickenHook/RestrictionBypass/
[15] 2021. *Restrictions on non-SDK interfaces.* Retrieved August 24, 2021 from https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces
[16] 2021. *Test using the veridex tool.* Retrieved August 24, 2021 from https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces#test-veridex-tool
[17] 2021. *Understanding Android API levels.* Retrieved August 24, 2021 from https://docs.microsoft.com/en-us/xamarin/android/app-fundamentals/android-api-levels/
[18] 2021. *Using Java Reflection.* Retrieved August 24, 2021 from https://www.oracle.com/technical-resources/articles/java/javareflection.html
[19] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 23-26, 2014.*
[20] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-Scale Study of Application Incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Beijing, China, July 15-19, 2019.*
[21] Google. 2021. *android.app.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/app/package-summary
[22] Google. 2021. *android.content.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/content/package-summary
[23] Google. 2021. *android.graphics.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/graphics/package-summary
[24] Google. 2021. *android.media.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/media/package-summary
[25] Google. 2021. *android.net.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/net/package-summary
[26] Google. 2021. *android.os.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/os/package-summary
[27] Google. 2021. *android.telephony.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/telephony/package-summary
[28] Google. 2021. *android.util.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/util/package-summary
[29] Google. 2021. *android.view.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/view/package-summary
[30] Google. 2021. *android.widget.* Retrieved August 24, 2021 from https://developer.android.com/reference/android/widget/package-summary
[31] Google. 2021. *JNI tips.* Retrieved August 24, 2021 from https://developer.android.com/training/articles/perf-jni
[32] 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 (ASE), Montpellier, France, September 3-7, 2018.*
[33] 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 (ASE), Montpellier, France, September 3-7, 2018.*
[34] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic Framework for Android Malware Detection Using Deep Learning. *Digital Investigation* 24 (2018), S48–S59.
[35] Li Li, Tegawendé F. Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *Proceedings of the 32nd IEEE*

International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, October 2-7, 2016.

[36] 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 (ISSTA), Amsterdam, The Netherlands, July 16-21, 2018.*

[37] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. CDA: Characterising Deprecated Android APIs. *Empirical Software Engineering* 25, 3 (2020), 2058–2098.

[38] 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 the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Saint Petersburg, Russian Federation, August 18-26, 2013.*

[39] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC), Hyderabad, India, June 2-3, 2014.*

[40] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John C. Grundy. 2021. Identifying and Characterizing Silently-Evolved Methods in the Android API. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE–SEIP), Madrid, Spain, May 25-28, 2021.*

[41] Tarek Mahmud, Meiru Che, and Guowei Yang. 2021. 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.*

[42] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), Eindhoven, The Netherlands, September 22-28, 2013.*

[43] VirusTotal. 2021. *VirusTotal.* Retrieved August 24, 2021 from https://www.virustotal.com/

[44] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, September 3-7, 2016.*

[45] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. 2020. How Android Developers Handle Evolution-induced API Compatibility Issues: A Large-scale Study. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE), Seoul, South Korea, 27 June - 19 July, 2020.*