Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Scalable Online Vetting of Android Apps for Measuring Declared SDK Versions and Their Consistency with API Calls

Daoyuan Wu · Debin Gao · David Lo

Received: March 2019 / Accepted: September 2020

Abstract Android has been the most popular smartphone system with multiple platform versions active in the market. To manage the application's compatibility with one or more platform versions, Android allows apps to declare the supported platform SDK versions in their manifest files. In this paper, we conduct a systematic study of this modern software mechanism. Our objective is to measure the current practice of declared SDK versions (which we term as `DSDK` versions afterwards) in real apps, and the (in)consistency between `DSDK` versions and their host apps' API calls. To successfully analyze a modern dataset of 22,687 popular apps (with an average app size of 25MB), we design a scalable approach that operates on the Android bytecode level and employs a lightweight bytecode search for app analysis. This approach achieves a good performance suitable for online vetting in app markets, requiring only around 5 seconds to process an app on average. Besides shedding light on the characteristics of `DSDK` in the wild, our study quantitatively measures two side effects of inappropriate `DSDK` versions: (i) around 35% apps under-set the minimum `DSDK` versions and could incur runtime crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above; (ii) around 2% apps, due to under-claiming the targeted `DSDK` versions, are potentially exploitable by remote code execution, and half of them invoke the vulnerable API via embedded third-party libraries. These results indicate the importance and difficulty of declaring correct `DSDK`, and our work can help developers fulfill this goal.

Keywords SDK Version · API Call · Android Fragmentation · App Analysis

Daoyuan Wu
E-mail: dywu@ie.cuhk.edu.hk
Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong.

Debin Gao
E-mail: dbgao@smu.edu.sg
School of Information Systems, Singapore Management University, Singapore.

David Lo
E-mail: davidlo@smu.edu.sg
School of Information Systems, Singapore Management University, Singapore.

1 Introduction

In recent years, we have witnessed the extraordinary success of Android, a smartphone operating system owned by Google. At the end of 2013, Android became the best-selling phone and tablet OS. As of 2015, Android evolved into the largest installed base of all operating systems. Over these years, Android keeps leading the global smartphone market share at over 80% [6]. Along with the fast-evolving Android, its fragmentation problem becomes more and more serious. Although new devices ship with the recent Android versions, there are still huge amounts of existing devices running old versions of Android [13].

To better manage the application's compatibility across multiple platform versions, Android allows apps to declare the supported platform SDK versions in their so-called "manifest" app configuration files (manifest afterwards). We term these declared SDK versions as `DSDK` versions. The `DSDK` mechanism is a modern software mechanism with which, to the best of our knowledge, few systems are equipped until Android. Nevertheless, it receives little attention so far, and few understandings are known about the effectiveness of the `DSDK` mechanism.

In this paper, we aim to conduct a systematic study on the Android `DSDK` mechanism. Specifically, our objective is to measure the current practice of `DSDK` versions in real apps, and the (in)consistency between `DSDK` versions and their host apps' API calls. To make our measurement results representative, we select popular apps with at least one million installs each on Google Play as the dataset. More specifically, we have collected a large-scale dataset with 22,687 popular apps (570.8GB in total, with an average app size of 25MB), which covers 90.2% of all such apps (both free and paid ones) available on Google Play. Furthermore, our study utilizes the latest Android API evolution and covers all 28 versions of Android SDKs or API levels[1].

After collecting the dataset and building the API-SDK mapping, we perform a systematic `DSDK` and API call analysis of each individual app. We design our approach scalable and robust so that it can be readily deployed by online app markets (e.g., Google Play) to timely notify developers of the `DSDK` inconsistency in their apps. Given this objective, dataflow-based analysis is not suitable because existing Android dataflow analyses (notably FlowDroid [15] and Amandroid [43]) are expensive even when analyzing medium-sized apps, e.g., requiring ∼4 minutes for the 8MB Nextcloud app[2] [27]. Moreover, they need to first transform or decompile Android app bytecode into an intermediate representation (e.g., Soot Jimple or Java bytecode), the process of which is not fully accurate [38] and often leaves some apps unanalyzable in many previous studies [53] [17] [34] [39].

In our approach, we thus operate on the original Android bytecode level and employ a lightweight *bytecode search* for app analysis. Specifically, we retrieve `DSDK` versions and API calls *directly* from each app without decoding the manifest file and without transforming app bytecode, which enables robust processing of all 22,687 popular apps. We also handle multidex [5], a special Android bytecode mechanism that is often skipped by prior works but common in modern apps — 5,008 apps in our dataset split their bytecodes into multiple files. With the correctly extracted app bytecodes, we then search these bytecode texts to obtain

[1] The latest Android version at the time of our writing is Android 9 (API level 28).

[2] `https://f-droid.org/en/packages/com.nextcloud.client/`

valid API calls that are not guarded by `VERSION.SDK_INT` checking (developers can use such `if` statements to invoke an API only in certain Android platforms) and are also not in uninvoked third-party libraries. In this way, our approach preserves the scalability and makes itself suitable for online vetting: the median and average time for analyzing an app in our dataset is only 4.75s and 5.39s, respectively.

Theoretically, our lightweight approach is less accurate than dataflow-based approaches. This is because we did not perform (the expensive) flow tracking, and false positives certainly appear. Fortunately, this limitation would not affect the real usage of our approach, since in our objective, the approach is used by online app markets for checking apps uploaded by developers. In other words, we can ask developers to manually check the inconsistency warnings in their apps. Moreover, the manual effort required in such checking is also limited — around 80% apps have fewer than ten potentially inconsistent API calls each. This indicates that the number of inconsistency warnings per app reported by our bytecode search is well manageable for developers to perform a one-time manual check. It is worth noting that this paper is not for bug detection; instead, we aim for a comprehensive study on the current `DSDK` practice and its potential impacts. By employing a lightweight yet conservative approach, we can maximize the coverage of valid code and thus minimize false negatives (the dataflow tracking is sometimes too tight and could fail to process complex implicit flows, e.g., as high as 13 different kinds of implicit flows missed in FlowDroid according to a systematic assessment recently [20]).

In a nutshell, our study sheds light on the current `DSDK` practice of app developers and quantitatively measures two side effects caused by the inconsistency between `DSDK` versions (configured by the app developers in the manifest file) and API calls (made by the app during its execution). Specifically, the compatibility effect occurs when a minimum `DSDK` version is set too low so that certain APIs do not even exist in the corresponding lower versions of Android platforms. The consequence of such compatibility effect can cause runtime crashes. Additionally, the security effect could also happen when a target `DSDK` version is outdated (i.e., a lower version of APIs will be used even when a device runs on later versions of Android), causing that a vulnerable API is still rendered by the underlying system when the app runs on higher versions of Android. Next, we present our three sets of measurement results on `DSDK` versions and their inconsistency with API calls. Note that due to the conservative nature of our approach, the measurement results reported in this paper represent an upper bound of all potential `DSDK` problems (under the condition that common analysis difficulties, such as native code, are not considered).

Firstly, our measurement reveals some interesting characteristics of declared SDK versions in the wild. Specifically, nearly all apps define the `minSdkVersion` attribute, but 4.76% apps still do not claim the `targetSdkVersion` attribute (in our dataset collected in late 2018). Fortunately, this percentage has significantly dropped from 16.54% in 2015, which indicates that `DSDK` attributes nowadays are more widely adopted in modern apps. We further find that the minimal platform version most apps support nowadays is Android 4.1, whereas the most popular targeted platform version is Android 8.0. The median version difference between `targetSdkVersion` and `minSdkVersion` also increases from 8 (in our last analysis in 2015) to 9 (currently in the 2018 dataset).

Secondly, in terms of compatibility inconsistency, we find that around 35% apps under-set the `minSdkVersion` value, causing them to crash when running on

lower versions of Android platforms. Fortunately, only 11.3% apps could crash on Android 6.0 and above. We also show that by employing bytecode search for `SDK_INT` checking, our approach can reduce 17.3% false positives of compatibility inconsistency results. A detailed analysis of the Android APIs incurring compatibility inconsistency further reveals that some API classes, such as view, webkit, and system manager related classes, are commonly misused.

Thirdly, our analysis of security inconsistency shows that around 2% apps set an outdated `targetSdkVersion` attribute and also invoke a dangerous WebView API, making themselves exploitable by remote code execution. In particular, around half of these vulnerable apps invoke the vulnerable `addJavascriptInterface()` API only because of their embedded third-party libraries. Additionally, our bytecode search of the `addJavascriptInterface()` invocation also helps reduce 12.2% false positives.

To summarize, we highlight the contributions of this paper as follows:

- (*New problem*) To the best of our knowledge, we are the first to conduct a systematic study on `DSDK`, a modern software mechanism that allows apps to declare the supported platform SDK versions. We also give the first demystification of the `DSDK` mechanism and its two side effects on compatibility and security. In particular, our preliminary conference version of this work [51] has motivated several recent follow-up works [30] [27] on bug detection.
- (*Novel approach*) We propose a robust and scalable approach that operates directly on the original bytecode level and leverages lightweight bytecode search to timely notify developers of the `DSDK` inconsistency in their apps. The evaluation using 22,687 popular apps (with an average app size as large as 25MB) shows that our approach achieves a good performance suitable for online app vetting, requiring only around 5 seconds to process an app on average.
- (*New findings*) Our measurement study obtains three major new findings, including (i) 4.76% apps still do not claim the `targetSdkVersion` attribute, although this percentage has significantly dropped from 2015 to 2018, (ii) around 35% apps under-set the minimum `DSDK` versions and could incur runtime crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above, and (iii) around 2% apps, due to under-claiming the targeted `DSDK` versions, are potentially exploitable by remote code execution, and half of them actually invoke the vulnerable API via embedded third-party libraries.

In this journal article, we extend our preliminary conference version [51] from the following perspectives: (1) We integrate a lightweight bytecode search into our approach so that it can be deployed by online app markets to timely notify developers of the DSDK inconsistency in their apps. We also add support for multidex-based apps and enhance the detection of uninvoked third-party libraries. (2) We evolve our dataset from an old set of 23,125 random apps in 2015 to a recent set of 22,687 popular apps in November 2018. We also find a lightweight way to build the latest API-SDK mapping. (3) By running experiments using the improved approach and dataset, we obtain more representative results and compare some of our new findings with the previous ones.

2 Demystifying Declared SDK Versions and Their Two Side Effects

In this section, we first demystify declared platform SDK versions in Android apps, and then explain their two side effects if inappropriate `DSDK` versions are used. Note that `DSDK` is different from the typical compilation SDK, which is only for compiling apps while `DSDK` is mainly for interpreting run-time API behaviors.

2.1 Declared SDK Versions in Android Apps

Listing 1 illustrates how to declare supported platform SDK versions in Android apps by defining the `<uses-sdk>` element [10] in apps' manifest files (i.e., `AndroidManifest.xml` [2]). These `DSDK` versions are for the runtime Android system to check apps' compatibility, which is different from the compiling-time SDK for compiling source codes. The value of each `DSDK` version is an integer, which represents the API level of the corresponding SDK. For example, if a developer wants to declare Android SDK version 5.0, she can set its value to 21. Since each API level has a precise mapping of the corresponding SDK version [14], we do not use another term, *declared API level*, to represent the same meaning of `DSDK` throughout this paper.

```
<uses-sdk android:minSdkVersion="integer"
          android:targetSdkVersion="integer"
          android:maxSdkVersion="integer" />
```

Listing 1: The syntax for declaring platform SDK versions in Android apps.

We explain the three `DSDK` attributes as follows:

- The `minSdkVersion` integer specifies the minimum platform API level required for an app to run. The Android system refuses to install an app if its `minSdkVersion` value is greater than the system's API level. Note that if an app does not declare this attribute, the system by default assigns the value of "1", which means that the app can be installed in all versions of Android.
- The `targetSdkVersion` integer designates the platform API level that an app targets at. An important *implication* of this attribute is that Android adopts backward-compatible API behaviors of the declared target SDK version, even when an app is running on a higher version of the Android platform. Android makes such compromised design because it aims to guarantee the same app behaviors as expected by developers, even when apps run on newer platforms. It is worth noting that if this attribute is not set, the `minSdkVersion` is used.
- The `maxSdkVersion` integer specifies the maximum platform API level on which an app can run. However, this attribute is *not* recommended and already deprecated since Android 2.1 (API level 7). Modern Android no longer checks or enforces this attribute during the app installation or re-validation. The only effect is that Google Play continues to use this attribute as a filter when it presents users a list of applications available for downloading. Note that if this attribute is not set, it implies no restriction on the maximum platform API level.

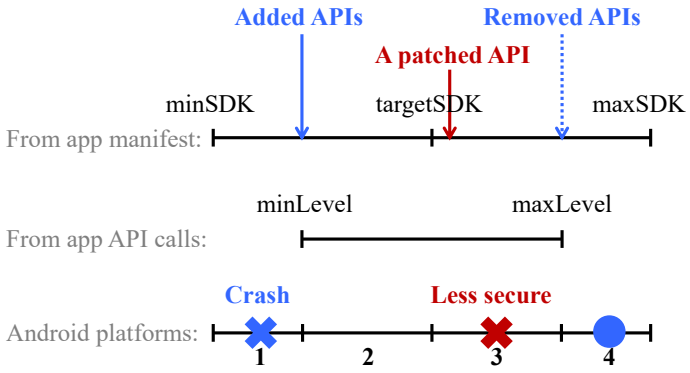


Fig. 1: Illustrating the two side effects of inappropriate `DSDK` versions.

2.2 Two Side Effects of Inappropriate DSDK Versions

Fig. 1 illustrates two side effects of inappropriate `DSDK` versions. We first explain the used symbols, and then describe the two side effects. As shown in Fig. 1, we can obtain $minSDK$, $targetSDK$, and $maxSDK$ from an app manifest file. Based on the API calls of an app, we can calculate the minimum and maximum API levels it requires, i.e., $minLevel$ and $maxLevel$. Eventually, the app will be deployed to a range of Android platforms between $minSDK$ and $maxSDK$.

2.2.1 Side Effect I: Causing Runtime Crashes

The blue part of Fig. 1 shows two scenarios in which inappropriate `DSDK` versions could cause compatibility-related inconsistency. The first scenario is $minLevel > minSDK$, which means a new API is introduced after the $minSDK$. Consequently, when an app (i) runs on Android platforms between $minSDK$ and $minLevel$ (marked as block 1 in Fig. 1) and (ii) executes that new API, it will crash. We verified this case using the `VpnService` class's `addDisallowedApplication()` API, which was introduced on Android 5.0 at API level 21. We invoked this API in the MopEye app [48] and ran it on an Android 4.4 device. When the app executed the `addDisallowedApplication()` API, it crashed with the `java.lang.NoSuchMethodError` exception.

The second scenario is $maxSDK > maxLevel$, which suggests that an old API is removed at the $maxLevel$. Although it looks like the app would crash when it runs on Android platforms between $maxLevel$ and $maxSDK$, it turns out that Google intentionally keeps the forward compatibility (by keeping those removed APIs in the framework as hidden APIs) so that developers have no concern in oversetting `maxSdkVersion`. As a result, this scenario would not cause runtime method availability errors. Therefore, in this paper, we measure only the first scenario of compatibility inconsistency that can cause runtime crashes.

2.2.2 Side Effect II: Making Apps Vulnerable

The red part of Fig. 1 shows the scenario where inappropriate `DSDK` versions cause failure for the app that should be patched. Supposing an app calls an API whose implementation is vulnerable at $targetSDK$, even when the app runs on an updated Android system (with API level $> targetSDK$), Android still exhibits the compatibility behaviors, i.e., the vulnerable implementation of the API at $targetSDK$ in this case.

Table 1: Vulnerable APIs or components on Android and their patched versions.

Vulnerable APIs/Components	Patched SDKs (Android)	Changed Behavior
`file://` scheme in WebView	$targetSDK \geq 16$ (4.1+)	Fix flawed same-origin policy [46]
Content Provider component	$targetSDK \geq 17$ (4.2+)	Disable the default exposure [54]
`addJavascriptInterface()`	$targetSDK \geq 17$ (4.2+)	Stop Java reflection for RCE [45]
`PreferenceActivity` class	$targetSDK \geq 19$ (4.4+)	Add `isValidFragment()` for apps to prevent Fragment Hijacking [37]
`javascript:` in WebView	$targetSDK \geq 19$ (4.4+)	JavaScript URLs are executed in a separate WebView context [47]
`Context.bindService()`	$targetSDK \geq 21$ (5.0+)	Do not accept Implicit Intents [29]

Table 1 summarizes the previously reported vulnerable APIs or components on Android and their patched versions. They were all wide-spread API-level vulnerabilities on Android, causing significant security impacts. Although by-default fixes were subsequently provided at the API level, as shown in Table 1, they often require developers' cooperation at the app level (e.g., updating app configuration). Otherwise, vulnerabilities could still appear even on patched versions of Android [29] [49]. In our context, an app could be exploited if they invoke the vulnerable APIs without declaring the updated `targetSdkVersion`. As a result, analyzing these "old" vulnerabilities is still worthwhile and could demonstrate the security impact of our study.

In this paper, we specifically measure the vulnerable `addJavascriptInterface()` API for two reasons. First, it has a clear API pattern for inconsistency measurement, while other cases in Table 1 involve multiple component-level factors that could cause a vulnerability. Second, the `addJavascriptInterface()` API gives rise to the most serious security issue [42]. By exploiting this API, attackers are able to inject malicious code, which can cause remote code execution (e.g., stealing sensitive information from a victim app or SD card). Google later fixed this weakness on Android 4.2 and above. However, if an app sets the `targetSdkVersion` lower than 17 and also calls this API, the system will still render the vulnerable API behavior even when running on Android 4.2+. Such vulnerable app examples are available at `https://sites.google.com/site/androidrce/`.

3 Methodology

To understand how `DSDK` versions are used in the wild and the pervasiveness of the two side effects in real apps, we propose an automatic approach for a systematic

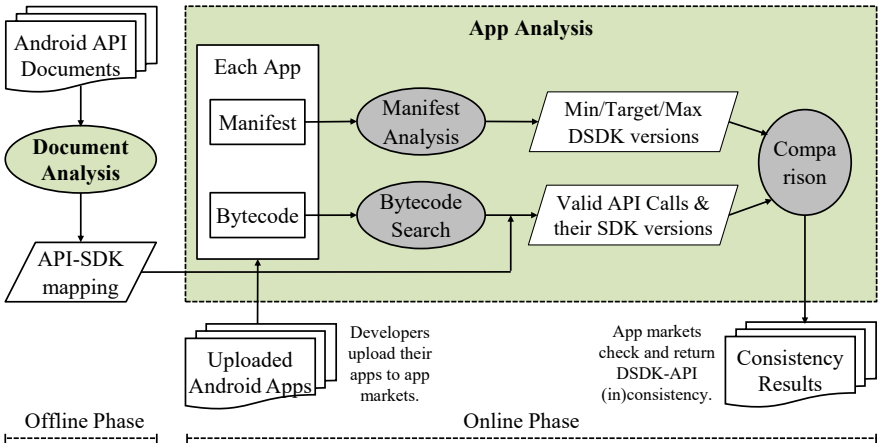


Fig. 2: The overview of our methodology.

measurement. In this section, we first present an overview of our methodology, and then its two main analysis phases.

3.1 Overview

Our main design goal is to help the app markets timely notify developers the `DSDK` inconsistency in their apps. Fig. 2 illustrates its overall design, where the app analysis part is conducted in the online phase. Since our app analysis requires the *API-SDK mapping* as an input (for calculating API levels of all valid API calls in an app), we further conduct Android API document analysis to build a mapping between each Android API and their corresponding SDK versions (or API levels). As this step is performed only once, we include it in the offline phase.

The majority of our approach is designated for the online vetting of apps. Specifically, whenever developers upload a new or updated app to app markets, we first unzip this app to obtain its bytecode DEX file(s). We then launch manifest analysis to robustly retrieve an app's declared SDK versions. For bytecode analysis, the novelty is that we propose a lightweight bytecode search, instead of heavyweight dataflow analysis, to extract valid API calls. Finally, we leverage the API-SDK mapping to calculate the range of the corresponding API levels from API calls, and compare them with the declared SDK versions. The output is the (in)consistency results between declared SDK versions and API calls. It is worth noting that *multiple-apk* analysis [51] is no longer needed in our online analysis, because app markets control all versions of APKs and multiple-apk mechanism is largely used for different hardware configuration [8].

3.2 Offline Phase: API Document Analysis

In this subsection, we present our offline phase in detail, including both the methodology and results of API document analysis.

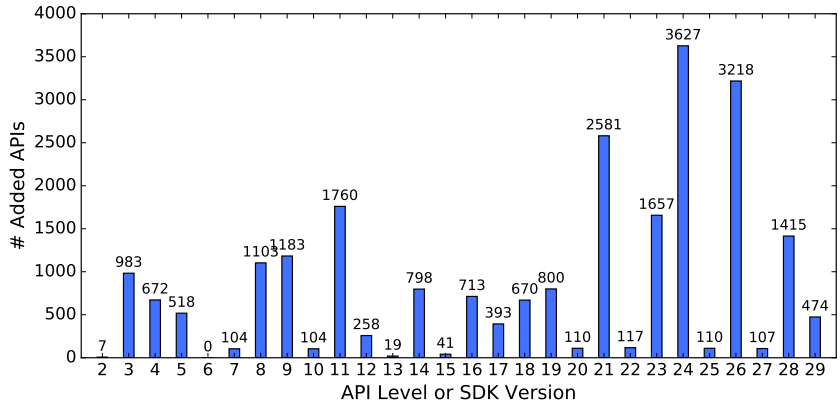


Fig. 3: The distribution of added Android APIs across different SDK versions.

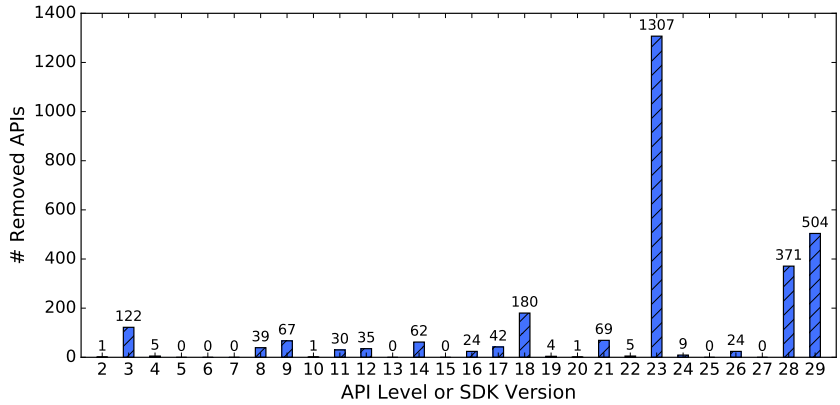


Fig. 4: The distribution of removed Android APIs across different SDK versions.

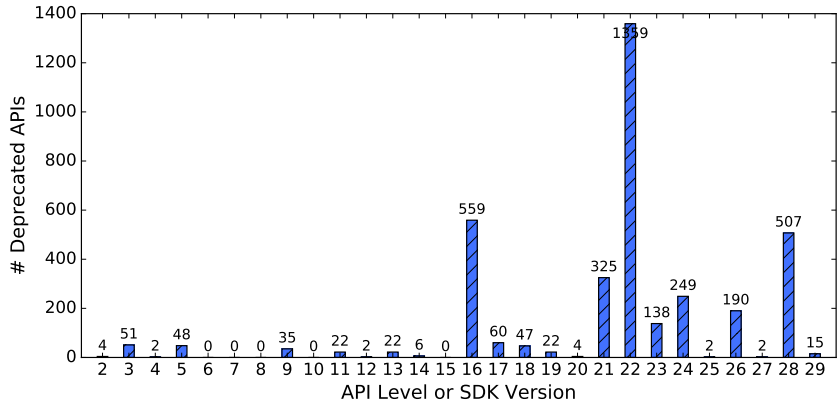


Fig. 5: The distribution of deprecated Android APIs across different SDK versions.

Building the API-SDK mapping. There are two potential approaches for building the API-SDK mapping. One is to analyze Android API documents by parsing a SDK document called `api-versions.xml`. A previous API study [35] and our preliminary study [51] leveraged this approach to obtain initial and added APIs, but they did not cover removed and deprecated APIs because of no such information in the `api-versions.xml` file. Hence, they also needed to analyze the HTML files in the `api_diff` directory, which is unfortunately error-prone [51]. The other approach is to directly retrieve the API-SDK mapping from each SDK `jar` file. However, different SDK releases under the same API level may have some API differences, and there are over 600 releases[3] for the 28 API levels at the time of our writing. As a result, conflicted API mappings could be recorded, e.g., marking the `Gravity.getAbsoluteGravity` API removed in SDK version 16 and then added back in version 17 [30].

Fortunately, we find that the first approach now covers all kinds of Android APIs. Specifically, the latest `api-versions.xml` file released in Android 9 SDK records all added, removed, and deprecated APIs. Therefore, we can simply parse this file to obtain a complete API-SDK mapping.

Document analysis results. With the accurate API-SDK mapping, we are able to present a comprehensive evolution of Android APIs across different SDK versions. Fig. 3, 4, and 5 plot the distribution of added, removed, and deprecated Android APIs from API level 2 to the very recent API level 29, respectively. Overall, we find that 26,466 (67.8%) out of the total 39,034 Android APIs are changed. This result indicates that Android APIs evolve dramatically during the whole evolution.

The biggest change in the Android API evolution is to add 23,542 APIs since level 2, as shown in Fig. 3. Specifically, Android 7.0 (API level 24) changed most, with 3,627 new APIs introduced. Android 8.0 (API level 26) and Android 5.0 (API level 21) also introduced a significant number of new APIs, with 3,218 and 2,581 APIs added, respectively. Other versions of platforms with a large number of added APIs are Android 3.0 (API level 11), Android 6.0 (API level 23), and Android 9.0 (API level 28). These new Android APIs bring a huge risk of compatibility inconsistency, causing runtime crashes on lower versions of Android. In particular, we notice that over half (13,306, 56.5%) of all the added APIs are introduced since Android 5.0, giving them a higher chance of causing compatibility inconsistency than the rest of added APIs.

In contrast, only 4,830 (18.2%) APIs involve the removal change (i.e., removed or deprecated; some of them are also introduced after API level 2), with 3,671 APIs deprecated and 2,902 APIs finally removed. According to Fig. 4 and 5, the biggest removal happens in Android 5.1 and 6.0 (API level 22 and 23), with 1,359 APIs deprecated and 1,307 APIs removed afterwards. Moreover, Android 9.0 (API level 28) deprecates 507 APIs and its next version (API level 29) removes 504 of them, which suggests that Google plans to remove a large number of APIs in the release of Android 9.0. Additionally, although Android 4.1 (API level 16) deprecated 559 APIs, only 222 APIs were removed in the subsequent Android 4.2 and 4.3.

To sum up, 23,542 (60.3%) out of all the 39,034 Android APIs are introduced at a SDK version other than the initial Android SDK version (i.e., API level 1), which brings a high risk for developers to under-set the `minSdkVersion` attribute.

[3] See tags in `https://android.googlesource.com/platform/frameworks/base.git/+refs`.

On the other hand, much fewer Android APIs, 7.4% of all APIs, are mapped to a range of SDK versions that have an upper limit (i.e., deleted in recent SDK versions).

3.3 Online Phase: Android App Analysis

In this subsection, we present three major modules in the online analysis phase, namely manifest analysis, bytecode search, and consistency comparison in Fig. 2.

3.3.1 Retrieving `DSDK` Versions via Manifest Analysis

To robustly retrieve `DSDK` versions from all apps, we propose a new manifest analysis method that leverages `aapt` (Android Asset Packaging Tool) [1] to retrieve `DSDK` *directly* from each app without extracting and decoding the manifest file. This method is more robust than the traditional `apktool`-based manifest extraction [4], which requires to extract and decode the manifest into a plaintext file. Indeed, our `aapt`-based approach can successfully analyze all 22,687 apps, whereas a previous work [52] showed that `apktool` failed six times in the analysis of just 1K apps. Specifically, we utilize the `dump baging` command in `aapt` to extract the `DSDK` versions. In this way, we can directly retrieve the correct `DSDK` versions without analyzing raw manifest files. Therefore, even when an app contains old or unreferenced manifest files, it would not affect our analysis.

In the course of implementation and evaluation, we observed and handled two kinds of special cases. First, some apps define `minSdkVersion` multiple times, for which we only extract the first value. Second, we apply the default rules (see Sec. 2.1) for apps without `minSdkVersion` and `targetSdkVersion` defined. More specifically, we set the value of `minSdkVersion` to 1 if it is not defined, and set the value of `targetSdkVersion` (if it is not defined) using the `minSdkVersion` value.

Besides retrieving `DSDK`, our manifest analysis also parses all components registered in the manifest to generate a list of valid components and their root (Java) class names. This information will be used in the app analysis module to exclude uninvoked third-party libraries. Specifically, we execute the `dump xmltree` command in `aapt` to output all component information. In the process of parsing these components, we also generate their root class names according to this rule: if the component class does not overlap with the app package or `<application>` name (i.e., this class could be from a third-party library), we record the entire class name as the root class; otherwise, only the leading two or three name portions are treated as the root class.

3.3.2 Extracting Valid API Calls via Bytecode Search

The main module in our app analysis is to extract valid API calls. A valid API call is a call *not* guarded by the `VERSION.SDK_INT` checking (a mechanism developers can use to invoke an API only in certain Android platforms). It should also not appear in uninvoked third-party libraries that are essentially dead code. To guarantee the scalability for online vetting, we propose a lightweight *bytecode search*, instead of dataflow-based approaches, for app analysis, because existing Android dataflow analyses, notably FlowDroid [15] and Amandroid [43], are expensive even

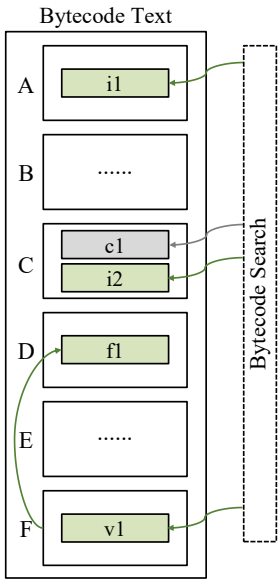


Fig. 6: A high-level overview of our bytecode search mechanism.

when analyzing medium-sized apps, e.g., requiring ∼4 minutes for just an app of size 8MB [27].

Moreover, we operate on the original Android bytecode level without decompiling app bytecodes, which helps reduce false negatives. This is because the process of transforming or decompiling Android app bytecode into an intermediate representation (usually Java bytecode) is not fully accurate [38]. As a result, many previous studies [53] [17] [34] [39] often failed to handle some apps, causing false negatives in their analysis. In contrast, by directly analyzing app bytecodes, we robustly process all 22,687 popular apps in our dataset. Specifically, we leverage the `dexdump` tool [50] to translate compressed bytecodes into plain bytecode texts (similar to using `objdump` to generate assembly code texts), upon which we can then launch bytecode search to extract valid API calls. Note that `dexdump`, as an official Android SDK tool, is very robust, and it does not generate intermediate representation. We also dump (multiple) app bytecodes into a (combined) plaintext [50] to handle multidex [5], a special bytecode format often skipped by prior works but indeed common in modern apps — 5,008 apps in our dataset split their bytecodes into multiple files. Hence, we avoid another common source of false negatives.

In the rest of this subsection, we first introduce the basic bytecode search mechanism before describing our bytecode search of `VERSION.SDK_INT` checking and vulnerable API calls in details. We then explain how we exclude uninvoked third-party libraries during the search process.

The basic bytecode search mechanism. Fig. 6 shows a high-level overview of our bytecode search mechanism. The bytecode text outputted by `dexdump` is a sequence of code statements, hierarchically organized by different class and method bodies. In Fig. 6, we show six method bodies (from method A to method F), where

```
1 VpnService.Builder builder = new VpnService.Builder();
2 if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
3     builder.addDisallowedApplication(Constant.PkgName);
4 }
```

Listing 2: An example of `VERSION.SDK_INT` checking.

their corresponding class bodies are omitted for simplicity. As illustrated in the figure, our bytecode search scans these methods to locate inconsistent API calls (e.g., call site i1 and i2 in method A and C, respectively) and vulnerable API calls (e.g., call site v1 in method F). We can perform further search to determine in which class an interested method is invoked, e.g., Fig. 6 shows that method F (containing vulnerable API call v1) is called by another method D. Besides the method search, we can also launch `if` statement search to locate conditional checking, e.g., statement c1 that surrounds call site i2 in method C.

Searching `VERSION.SDK_INT` checking. As mentioned earlier in this subsection, developers can use `if` statements with `VERSION.SDK_INT` checking to invoke an API only in certain Android platforms, thus avoiding the inconsistency problem. Listing 2 shows an example of `VERSION.SDK_INT` checking, which invokes the `addDisallowedApplication()` API (introduced since API level 21) only on Android 5.0 and above. To avoid such false positives, our approach must handle the `VERSION.SDK_INT` checking.

Our strategy is to perform both API call and `VERSION.SDK_INT` checking search and see whether the two search results overlap in the same method. For example, in Fig. 6, our bytecode search locates both checking statement c1 and API call i2 in method C. Since these two search results overlap and API call i2 is invoked below checking statement c1, we are thus confident that this API call has been guarded with a corresponding `VERSION.SDK_INT` checking. Moreover, according to a recent study [27], 88.65% of the `DSDK` checking usages directly compare the `VERSION.SDK_INT` variable with a constant Android version number, which makes our bytecode search strategy appropriate.

Searching vulnerable API calls. For a vulnerable API call, we further employ bytecode search to determine whether it is initialized by app's own code or library code. This is particularly important for the vulnerable API studied in this paper, namely `addJavascriptInterface()`, because a previous study has shown that over 47% of top 40 ad libraries create their Javascript Interfaces [45]. Specifically, after locating vulnerable API call v1 in method F, we further search the invocation(s) of method F to check its origin class.

Excluding uninvoked third-party libraries. An important issue during our bytecode search is to exclude uninvoked third-party libraries. To tackle this problem, we cannot simply employ library detection (e.g., LibScout [18] and LibD [32]) to exclude *all* libraries, because this approach also ignores those invoked and thus valid library code. To keep valid libraries as much as possible while minimizing the false positives raised by uninvoked libraries, we propose a lightweight yet practical approach that combines both heuristics-based component analysis and API-based bytecode search. Specifically, we first conservatively exclude all the code that has no relationship with app component information even though some of them might be functionality-supporting code. We achieve this by performing manifest anal-

ysis and generating root classes for all registered components, as mentioned in Sec. 3.3.1. A class whose code does not appear in any root class is thus recognized as an uninvoked library or dead code. Note that even for a valid third-party library, only its registered components will be analyzed because not all code in a library will be invoked by the main app. Furthermore, when a candidate API call is going to be reported during the detection phase, we launch one more bytecode search to double check its invocations. Eventually, the identified inconsistency cases will be confirmed by developers, and as we will discuss in Sec. 6, the effort of performing such checking is minimal. In this way, we consider valid third-party libraries but also minimize their potential false positives, without relying on the expensive dataflow-based analysis that does not meet the objective of online vetting in app markets.

3.3.3 Calculating API Levels and Comparing Their Consistency with DSDKs

With the extracted API calls, we use the API-SDK mapping to compute the range of corresponding API levels (i.e., from `minLevel` to `maxLevel`, as explained in Sec. 2.2). The `minLevel` of an app is the maximum of all its valid API calls' corresponding `minLevel` values (i.e., all correspondingly added SDK versions), while the `maxLevel` of an app is the minimum of all valid API calls' corresponding `maxLevel` values (i.e., all correspondingly removed SDK versions). If an API is never removed, we set a large flag value (e.g., 100,000) to represent its `maxLevel` value.

We then compare the extracted `DSDK` values with the calculated API levels to obtain the following two kinds of inconsistency (as previously mentioned in Sec. 2.2):

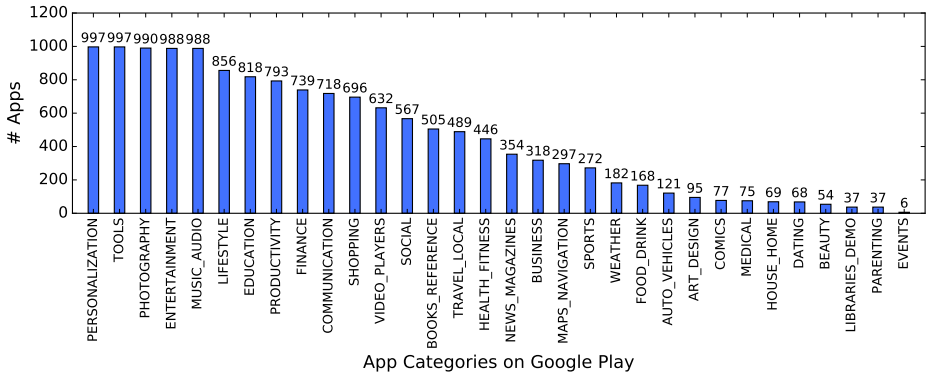
- `minSdkVersion` < `minLevel`: the `minSdkVersion` is set too low and the app would crash when it runs on platform versions between `minSdkVersion` and `minLevel`.
- `targetSdkVersion` < `maxLevel`: the `targetSdkVersion` is set too low and the app could be updated to the version of `maxLevel`. If the `maxLevel` is infinite, the `targetSdkVersion` could be adjusted to the latest Android version.

4 Evaluation

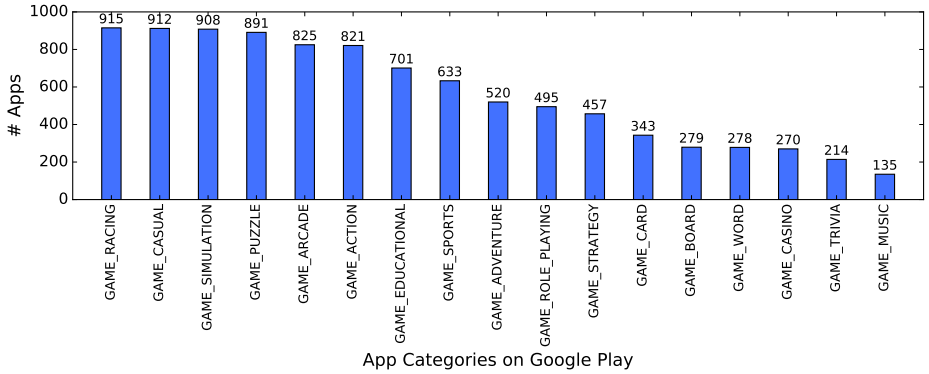
Our evaluation aims to answer the following five research questions:

RQ1 What are the *current DSDK characteristics* in popular real-world apps?
RQ2 How pervasive is the *compatibility-related inconsistency* in real-world apps?
RQ3 How pervasive is the *security-related inconsistency* is in real-world apps?
RQ4 How *scalable* is our inconsistency detection approach?
RQ5 What is the *updatablity* of the buggy apps? Are they still being maintained?

We choose popular real-world apps, instead of randomly selected apps or open-source apps, for evaluation, because they are most likely installed by regular users (according to Google Play installs). Hence, the obtained measurement results can reflect the `DSDK` practice in the wild. In this section, we first describe how we collect such a large dataset in Sec. 4.1. Based on this dataset, we then answer the five research questions from Sec. 4.2 to Sec. 4.6.



(a) 32 non-game app categories.



(b) 17 game app categories.

Fig. 7: Bar charts of the distribution of popular apps across different categories.

4.1 Dataset

We collect popular apps on Google Play via the AndroZoo repository [11], which contains a total of 3,699,731 unique[4] Google Play apps at the time of our crawling on 11 November 2018. However, AndroZoo does not provide the app install information, which is needed to determine the popularity of each app. To quickly locate popular apps, we leverage the top app lists available at `https://www.androidrank.org`. Specifically, we crawled top 1,000 app in each Google Play category (49 categories in total, including 17 different game sub-categories), and recorded the package names of apps *with over one million installs*. This allows us to obtain a list of 25,144 popular apps, 22,687 (the rest are either paid apps or not indexed by AndroZoo) of which are available on AndroZoo. We then downloaded these 22,687 apps as our dataset.

To understand the distribution of these popular apps across different app categories, we plot bar charts in Fig. 7 that cover both 32 non-game app categories and 17 game sub-categories. In particular, 17 game sub-categories contribute to a total

[4] An app is unique if its package name, instead of SHA1/256, is different from other apps.

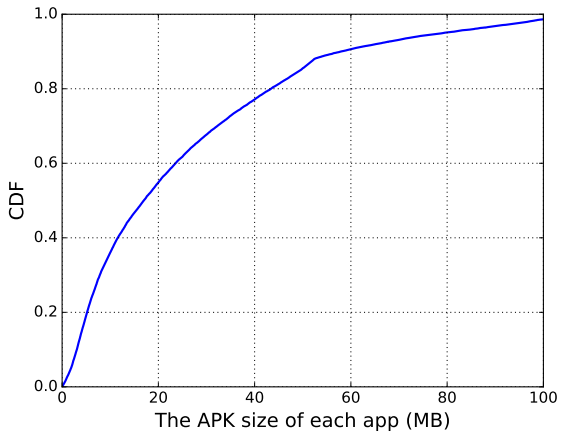


Fig. 8: CDF plot of the APK file size of each app in our dataset.

of 10,695 popular apps, which indicates that game apps are commonly installed by real-world Android users. According to Fig. 7, app categories like "Personalization", "Tools", "Photography", "Entertainment", and "Music" also produce a large number of popular apps, almost 1K popular apps per category. We notice that daily-used categories, such as "Communication" and "Social", however, do not generate an equivalent number of popular apps, with only 600 to 700 popular apps. This is because in these categories, several very popular apps, e.g., WeChat and Facebook, dominate a large portion of the market share. Lastly, it is also reasonable for some unpopular categories, such as "Medical" and "Libraries & Demo", to have a limited number of popular apps.

It is also important to measure the distribution of app size in our dataset. Fig. 8 plots the CDF (cumulative distribution function) of the APK file size of each app in our dataset. We can see that over 40% apps have a size larger than 20MB, and over 20% apps are even larger than 40MB each. This indicates that a significant portion of modern apps are no longer small. Indeed, the average app size in our dataset is 25MB, much larger than the size of apps used in several prior dataflow analysis studies (e.g., apps below 5MB were evaluated in AppContext [53], and the maximum app size in IctApiFinder [27] is 8MB). Therefore, scalability is a key design objective for our approach, and we will evaluate it extensively in Sec. 4.5.

4.2 RQ1: Characteristics of Declared SDK Versions in the Wild

In this section, we report a total of four findings regarding RQ1. We also compare these new findings with our previous results in [51], which measured a dataset of 22.7K apps crawled in 2015.

Finding 1-1: *Nearly all apps define the* `minSdkVersion` *attribute, but 4.76% apps still do not claim the* `targetSdkVersion` *attribute, although this percentage has significantly dropped compared to our prior analysis in 2015.* Table 2 shows the number and percentage of non-defined `DSDK` attributes in our dataset. We can see that nearly all apps have defined the `minSdkVersion` attribute while almost

Table 2: The number and percentage of non-defined DSDK attributes in our dataset.

	# Non-defined	% Non-defined
minSdkVersion	5	0.02%
targetSdkVersion	1,079	4.76%
maxSdkVersion	22,623	99.72%

zero app defines the maxSdkVersion attribute. This result is good because, as we described in Section 2.1, defining minSdkVersion is necessary while maxSdkVersion is *not*. However, we also notice that 1,079 (4.76%) apps still do not claim the targetSdkVersion attribute, which causes their targetSdkVersion values be set to the corresponding minSdkVersion values by default.

Fortunately, the percentage of non-defined targetSdkVersion has dropped significantly as compared to our prior analysis in 2015, from 16.54% to 4.76%. One important factor is the popularity of Android Studio in recent years, which has become the de-facto IDE (integrated development environment) for Android app development. Since Android Studio by default sets and enforces the minSdkVersion and targetSdkVersion attributes, the percentage of non-defined targetSdkVersion naturally drops and we expect that this percentage will further decrease with more apps getting updated.

Finding 1-2: *Some targetSdkVersion attributes are set to outlier values.* We find that a total of 45 apps in our dataset declare their targetSdkVersion attributes as outlier values, which is close to our prior analysis result in 2015 when we encountered 55 such cases. There are two types of outlier values. The first is that targetSdkVersion is set to an API level not in the range of released SDKs. At the time of our analysis, the valid API levels are from 1 to 28 (Android 9.0). However, 12 apps set their targetSdkVersion to larger than 28, namely 29, 30, and 31. In our prior analysis [51], one app even set its targetSdkVersion value to 10000. This is probably because their developers want to always target at the latest Android SDK.

The other type of outliers is that the targetSdkVersion value is set to a value lower than the minSdkVersion value. Normally, targetSdkVersion should be greater than or equal to minSdkVersion, but 33 apps have negative targetSdkVersion − minSdkVersion values. This number is almost the same as that in our prior analysis in 2015 (34 apps at that time). In particular, one app (com.leftover.CoinDozer) defines its targetSdkVersion as 0, although its minSdkVersion value is 8. We believe that this class of outliers is due to developers' mistakes in declaring the DSDK attributes.

Finding 1-3: *The minimal platform version most apps support is Android 4.1, whereas the most targeted platform version is Android 8.0. This has dramatically evolved since our last analysis in 2015.* We first study the distribution of minSdkVersion. According to Fig. 9, the majority (89%) of apps have minSdkVersion lower than or equal to level 16 (Android 4.1), which means that they can run on nearly all (99.5%) Android devices in the market nowadays [13]. Specifically, the minimal platform version most apps support is Android 4.1 (level 16), while that in our last analysis in 2015 was only Android 2.3 (level 9). However, Android 2.3 still ranks in the second place, with 3,614 apps' minSdkVersion

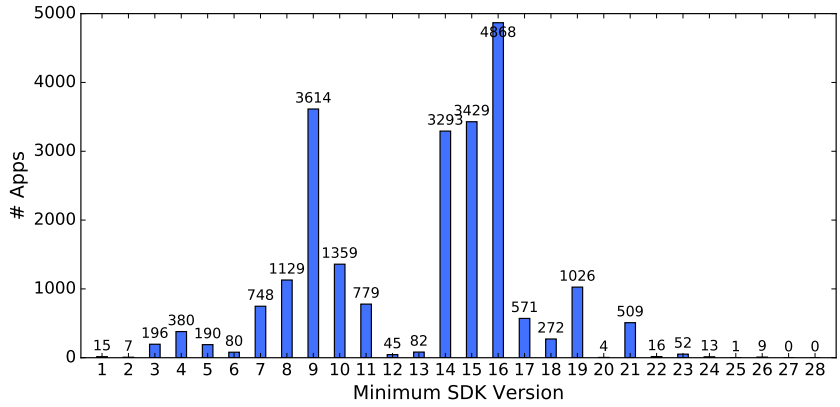


Fig. 9: Distribution of `minSdkVersion`.

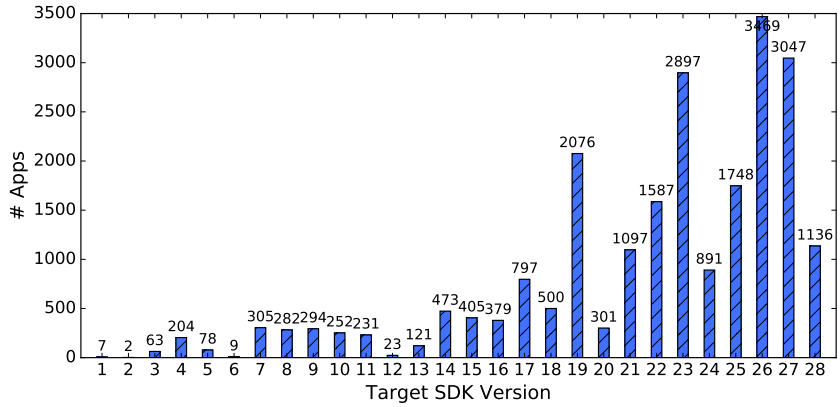


Fig. 10: Distribution of `targetSdkVersion`.

targeted at. Besides Android 4.1 and 2.3, two Android 4.0.x (level 14 and 15) platform versions are also commonly defined as apps' `minSdkVersion`.

On the other hand, Fig. 10 plots the distribution of `targetSdkVersion`. We can see that 80% apps set their `targetSdkVersion` values to larger than or equal to level 19 (Android 4.4). In particular, the two most targeted platform versions are the most recent Android 8.0 (level 26) and 8.1 (level 27), while those in our last analysis in 2015 were Android 4.4 and 5.0. This suggests that modern apps keep pace with the evolution of the Android operating system. Besides Android 8, Android 6.0 (level 23) and 4.4 (level 19) still hold a significant portion of apps with the corresponding `targetSdkVersion` setting. Moreover, Android 7.0.x (level 24 and 25) and Android 5.0.x (level 21 and 22) also attract considerable apps being targeted at.

Finding 1-4: *The median version difference between* ***`targetSdkVersion`*** *and* ***`minSdkVersion`*** *is 9, while that in our last analysis was 8. This 11% increase indicates that Android apps nowadays need to support more Android platforms.* We define a new metric called `lagSdkVersion` to measure the version difference

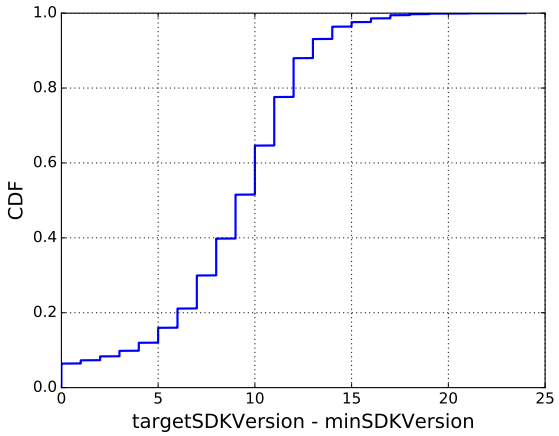


Fig. 11: CDF plot of `lagSdkVersion`.

between `targetSdkVersion` and `minSdkVersion`, as shown in Equation 1.

$$\texttt{lagSdkVersion} = \texttt{targetSdkVersion} - \texttt{minSdkVersion} \tag{1}$$

After removing the negative `lagSdkVersion` values (i.e., outliers mentioned in Finding 1-2), we draw the CDF plot of `lagSdkVersion` in Fig.11. We first find that the median value of `lagSdkVersion` in our dataset is 9, while that in our last analysis in 2015 was 8. It indicates that Android apps nowadays need to support more Android platforms. This conclusion is further supported by the percentage of apps that have a `lagSdkVersion` value greater than 12. Compared to our prior analysis, this percentage has increased from 5% to 20%, which clearly shows that more and more apps nowadays support a wide range of Android platforms. On the other hand, the percentage of apps that have the same value for `targetSdkVersion` and `minSdkVersion` has also dropped from 20% in 2015 to 6.4% in 2018.

4.3 RQ2: Inconsistency Results with Compatibility Effect

In this section, we report three important findings regarding RQ2. Besides presenting compatibility results as the major finding, we summarize the reduced false positives by our bytecode search as compared to the prior conference version, and show that the newly added API classes are common sources of compatibility inconsistency.

Finding 2-1: *Around 35% apps under-set the* `minSdkVersion` *value, causing them potentially crash when running on lower versions of Android platforms. Fortunately, only 11.3% apps could crash on Android 6.0 and above.* As explained in Sec. 3.3.3, the compatibility inconsistency happens if `minSdkVersion` is less than `minLevel`. In our experiments, we thus count the number of API calls that have higher API level than `minSdkVersion` in each app, and denote it by `minOverNum`. The higher value an app's `minOverNum` is, the more likely that this app has the compatibility inconsistency.

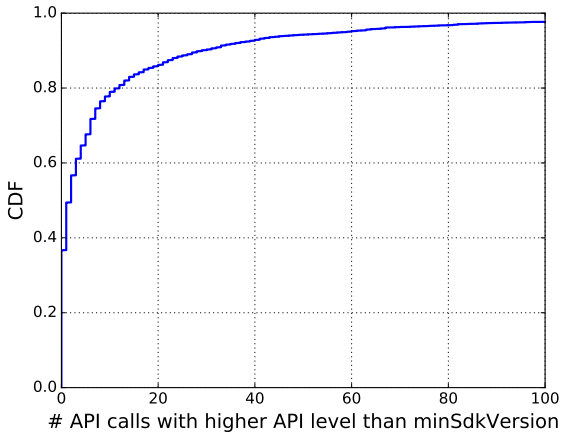


Fig. 12: CDF plot of `minOverNum` in each app.

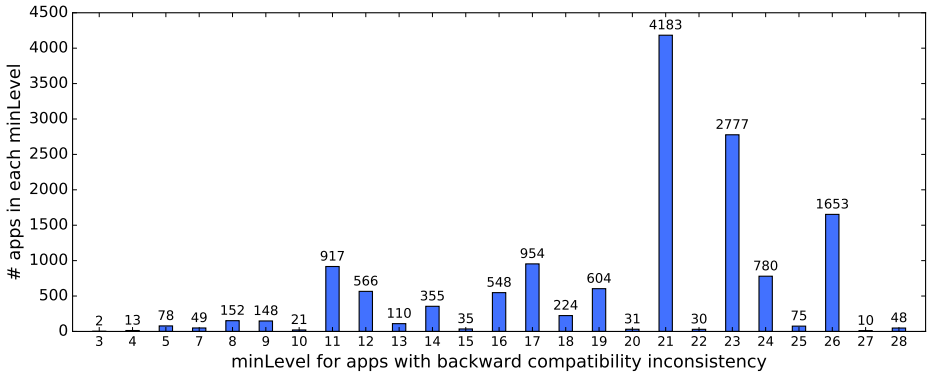


Fig. 13: Bar chart of the number of apps in each `minLevel`.

Fig. 12 shows the CDF plot of `minOverNum` in each app. We find that 14,363 (63.3%) apps have at least one API call that has higher API level than the corresponding `minSdkVersion`. To further increase the confidence of our analysis, we count the 8,019 (35.4%) apps that invoke over five different API calls with higher API levels than corresponding `minSdkVersion`. Therefore, we estimate that around 35% apps could crash when running on lower versions of Android platforms because they under-set the `minSdkVersion` value. Fortunately, we find that the number of inconsistency warnings per app reported by our bytecode search is well manageable for developers — 77.8% of the 14,363 apps have fewer than 10 different inconsistent API calls. It is thus not difficult for developers to perform a one-time manual check.

Fortunately, apps with compatibility inconsistency issues could crash *only* on certain Android platforms. More specifically, they could crash only on versions of platforms between `minSdkVersion` and `minLevel`, as illustrated earlier in Sec. 2.2. Therefore, it is necessary to study on which Android platforms those buggy apps could crash, because nowadays some lower versions of Android hold a limited

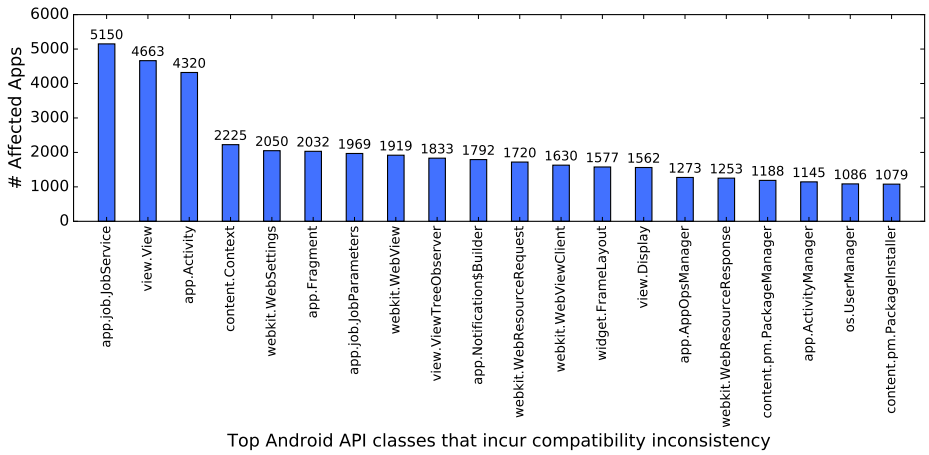


Fig. 14: Bar chart of the top 20 Android API classes (with "`android.`" prefix omitted) that incur compatibility inconsistency in our dataset.

market share, e.g., only 10.7% for Android lower than 5.0 as of July 2020 [13]. As a result, even if some apps are buggy with compatibility inconsistency, they cannot trigger the crash on user phones equipped with recent versions of Android.

Since `minLevel` is the indicator for maximum versions of Android platforms a buggy app could crash on, we plot a bar chart of `minLevel` in Fig. 13 for the 14,363 app detected with potential compatibility inconsistency. We can see that only 2,566 (11.3% of 22,687) apps could crash on Android 6.0 and above (via counting apps with `minLevel` larger than 23), and similarly 1,786 (7.9%) for Android 7.0 and above. In other words, most (11,797 out of 14,363) of potentially buggy apps cannot exhibit their incompatibility bugs on the majority of Android phones that are with 74.8% market share in July 2020 [13]. Furthermore, 8,990 out of 14,363 apps could crash only on Android lower than 5.0, which significantly limits the consequences of their incompatibility issues.

Finding 2-2: *We find that by employing bytecode search for `SDK_INT` checking, our approach can reduce 17.3% false positives of compatibility inconsistency results.* As mentioned in Sec. 3.3.2, a false positive of compatibility inconsistency could appear if an API call guarded with `SDK_INT` checking is not detected. Here we measure the number of such false positives that could be excluded by the bytecode search. We find that our search of `SDK_INT` checking avoids 3,003 apps from being mistakenly marked with compatibility inconsistency. Since there are at most 14,363 apps (i.e., true positives) that could crash when running on lower versions of Android platforms, the percentage of reduced false positives due to bytecode search is at least 17.3%.

Finding 2-3: *A detailed analysis of Android APIs that incur compatibility inconsistency reveals that some API classes, such as view, webkit, and system manager related classes, are commonly misused.* We further try to understand the common sources of compatibility inconsistency by analyzing the newly added Android APIs that incur compatibility inconsistency in our dataset. We find that 6,454 (27.4% of all 23,542) newly added APIs from 1,138 unique classes cause

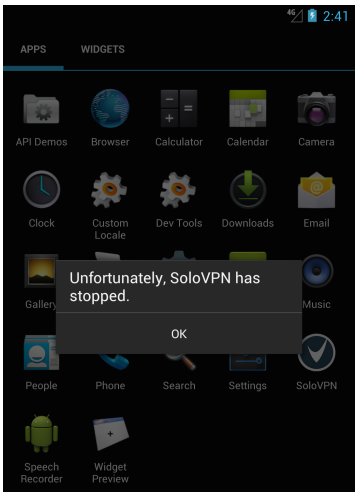


Fig. 15: A case study of the `DSDK` issue with incompatibility effect: Solo VPN.

compatibility inconsistency in at least one app in our dataset. In particular, 232 (20.4%) API classes affect more than 100 different apps each, making them the common sources of compatibility inconsistency. Fortunately, half of API classes only affect fewer than 10 apps each, which suggests that only some portions of API classes are prone to misuses.

We thus take a closer look at the top 20 Android API classes that cause compatibility inconsistency. As shown in Fig. 14, all of these classes affect over 1K apps each. In particular, the `JobService` class (introduced in Android 5.0, level 21) alone could cause compatibility inconsistency in around 5K apps. Other commonly misused API classes include those related to view (e.g., the `View`, `Activity`, `Context`, and `Fragment` classes), webkit (e.g., the `WebSettings` and `WebView` classes), and system manager (e.g., the `AppOpsManager` and `UserManager` classes). These classes nearly occupy all the top 20 misused ones.

Case study: Solo VPN. To demonstrate the impact of incompatibility `DSDK` issues, we identify a problematic app in our dataset and try to make it crash at the runtime. However, it is non-trivial to dynamically achieve this because a crash point may hide deep in certain paths or under certain conditions, which is why the previous work, CiD [30], requested developers themselves to help validate their detection results [3]. To simplify our testing, we intentionally targeted at the VPN apps based on the observation that some `VpnService` APIs require Android 5.0 at the API level 21. After testing a few VPN apps in our dataset, we quickly identified a buggy app, Solo VPN (`co.solovpn`, version: 1.32), which crashed immediately after we clicked the "Connect" VPN button on an Android 4.1 device. Fig. 15 shows the alert dialog popped up, stating that "Unfortunately, SoloVPN has stopped".

4.4 RQ3: Inconsistency Results with Security Effect

In this subsection, we present a total of three findings regarding RQ3.

Finding 3-1: *Around 2% apps set an outdated* `targetSdkVersion` *attribute and also invoke a dangerous WebView API, making themselves exploitable by remote code execution.* As explained in Sec. 2.2.2, we measure inconsistency results with the security effect by analyzing each app's `addJavascriptInterface()` API call and the declared `targetSdkVersion` attribute. In our dataset, 2,791 apps invoke the `addJavascriptInterface()` API, which suggests that calling this WebView API is necessary in many apps. However, 484 of them, i.e., 2.1% of the entire dataset of 22,687 apps, still set an outdated `targetSdkVersion` attribute below level 17, making themselves not only exploitable on Android lower than 4.2 but also vulnerable on higher versions of Android platforms. This could be avoided if their `targetSdkVersion` values are updated.

Finding 3-2: *Our bytecode search of* `addJavascriptInterface()` *invocation helps reduce 12.2% false positives.* Recall from Sec. 3.3.2 that we perform bytecode search to check whether an `addJavascriptInterface()` API call is invoked by a valid class. We find that without such checking, 551 apps can be detected with vulnerable combination of `addJavascriptInterface()` and `targetSdkVersion`. In other words, our search of `addJavascriptInterface()` invocation avoids 67 (551 - 484) apps from being mistakenly marked with security inconsistency. Hence, we conclude that our bytecode search reduces 12.2% false positives in the context of `addJavascriptInterface()`.

Table 3: The top five library classes that introduce `addJavascriptInterface()` API call in vulnerable apps and the number of apps affected.

Library Class	# Vulnerable Apps
Lcom/flurry/android/CatalogActivity;	41
Lcom/openfeint/internal/ui/NativeBrowser;	30
Lcom/doodlemobile/gamecenter/moregames/MoreGamesActivity;	19
Lcom/gau/go/launcherex/theme/classic/FullScreenAdWebPage;	17
Lcom/amazon/ags/html5/overlay/GameCircleUserInterface;	13

Finding 3-3: *Around half of the vulnerable apps invoke the* `addJavascript Interface()` *API only because of their embedded third-party libraries.* Our approach can also determine whether the `addJavascriptInterface()` API is invoked by app's own code or embedded by a third-party library. It turns out that 214 (44.2%) of 484 vulnerable apps invoke `addJavascriptInterface()` only because of their embedded third-party libraries. In particular, five libraries affect at least 10 vulnerable apps each. Table 3 lists their class names and the number of apps affected. We can see that the popular Yahoo Flurry SDK [7] and OpenFeint Game SDK [9] cause some apps with outdated `targetSdkVersion` vulnerable.

This finding gives two implications. First, developers must check whether a third-party library invokes some vulnerable APIs before embedding it into apps. Second, library producers also need to ensure that certain dangerous APIs are invoked only in safe versions of Android platforms, because a library can be used in any app with all kinds of `targetSdkVersion` values.

Case study: Exsoul Browser. To demonstrate the impact of insecure `DSDK` issues, we try to exploit a problematic app in our dataset. To exploit `addJavascript Interface` vulnerabilities, an adversary needs to inject a piece of malicious Javascript

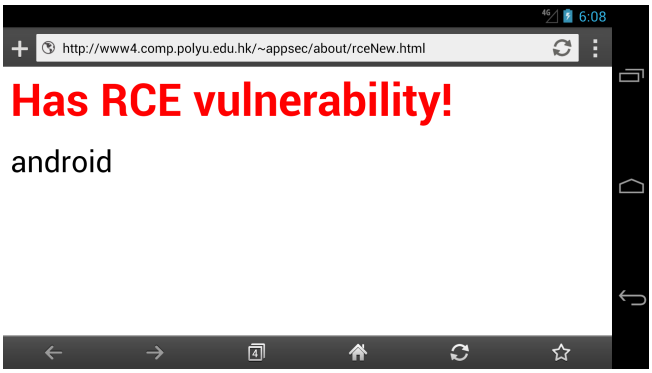


Fig. 16: A case study of the `DSDK` issue with security effect: Exsoul Browser.

code into a vulnerable WebView-based interface in the victim app. He or she could achieve this by either intercepting the HTTP traffic via a Man-In-The-Middle proxy or tricking victim users to directly browse a malicious website. We chose the second more convenient way and directly targeted at the browser apps in our dataset for tests. There was only one browser app, Exsoul Browser (`com.exsoul`), reported with `DSDK` security problems. We used it to browse a demonstration exploit website that we prepared before, `http://www4.comp.polyu.edu.hk/~appsec/about/rceNew.html`, which would output "Has RCE Vulnerability" if the tested browsing interface is vulnerable. As shown in Fig. 16, we successfully validate the `addJavascriptInterface` vulnerability in Exsoul Browser on an Android 4.1 device. We also find that Exsoul Browser exposed a Javascript interface named "android", which allows a malicious website to execute arbitrary commands by simply invoking this Javascript code: `android.getClass().forName("java.lang.Runtime").getMethod("getRuntime",null).invoke(null,null).exec(cmdArgs)`.

4.5 RQ4: Performance Metrics of Our Approach

In this section, we evaluate performance metrics of our approach to answer RQ4.

Finding 4-1: *Our approach achieves good scalability with an average time of 5.39s and the analysis time of 90% apps in less than 10 seconds. This makes our approach suitable for online vetting.* In Fig. 17, we present CDF plot of the amount of time required for our approach to analyze each app. We can see that more than 50% apps can be analyzed in less than five seconds each, with the median time of 4.75s. The average analysis time of all the 22,687 apps is only 5.39s. These results indicate that our approach achieves good scalability, therefore suitable for online vetting. App markets can deploy our approach to timely notify developers the `DSDK` inconsistency in their apps.

In contrast, dataflow-based approaches [30] [27] suffer from the scalability problem. Specifically, CiD [30] failed to analyze 387 apps (out of a dataset of 2,000 apps) due to timeouts and bugs. This 19.4% timeout or failure rate makes it infeasible for online vetting, let alone performance statistics were also not clear for those

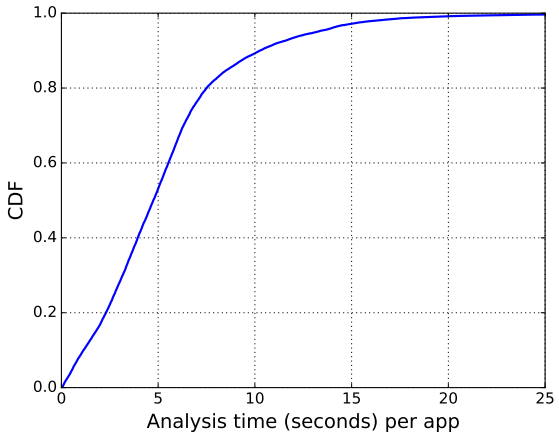


Fig. 17: CDF plot of the amount of time required for our approach to analyze each app.

successfully analyzed. On the other hand, IctApiFinder [27] takes 3 minutes and 45 seconds to analyze only an app of 8MB (the app is available via historical versions on `https://f-droid.org/en/packages/com.nextcloud.client/`), a size much smaller than the average size (25MB) of our dataset. This suggests that IctApiFinder is impractical to perform online vetting of a modern app dataset from Google Play (all apps evaluated by IctApiFinder were open-source apps from the F-Droid website).

Finding 4-2: *A further correlation analysis between analysis time and app size shows that the performance of our approach is approximately in a linear relationship with DEX file size of the app.* We find that the performance of our approach is always under control regardless of app size. This can be evaluated by performing a correlation analysis between analysis time and app size. In Fig. 18, we draw a scatter plot of the relationship between analysis time and the size of DEX file of the app (APK file contains both bytecode and resource files while DEX file is only for bytecode). According to this figure, the analysis time and DEX file size are approximately in a linear relationship, at the rate of around 30 seconds for a 40MB DEX file (note that we count the file size of multiple DEX files if any). There are some outliers of small apps with more analysis time (e.g., five apps under 20MB exceeding 30s), which is largely because these apps involve much more vulnerable API calls to search. On the other hand, the outliers of large apps with less analysis time is due to unused third-party libraries embedded. Overall, the linear relationship between analysis time and app size indicates that our approach can achieve good performance even with large apps.

4.6 RQ5: The Updatability of The Buggy Apps

In this subsection, we continue to understand the updatability of apps that were measured with `DSDK` issues in our dataset, i.e., whether they are still maintained by their developers. This is important because compared with the updatable apps

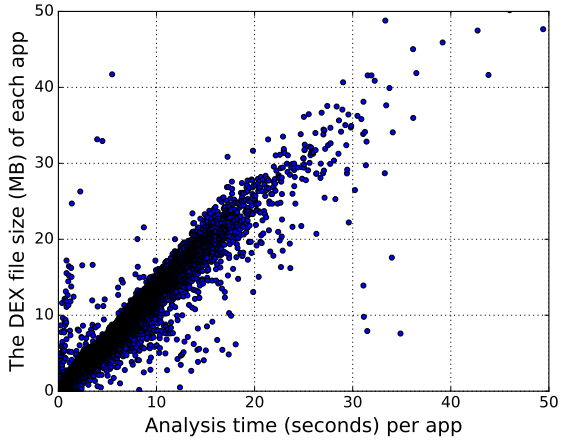


Fig. 18: Scatter plot of the relationship between analysis time and DEX size.

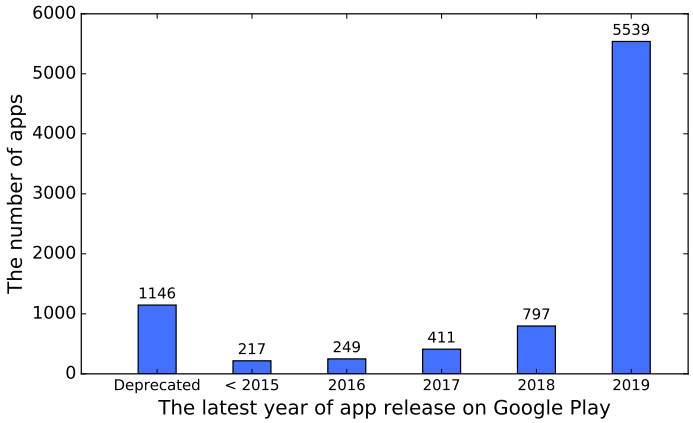


Fig. 19: Bar chart of the distribution of apps that were measured with `DSDK` issues in our dataset and their latest release years on Google Play.

that could eventually address their `DSDK` issues via the app updates, outdated apps have no maintainers to periodically update and fix their `DSDK` problems. To study to what extent this problem is, we use 8,359 unique apps (8,019 incompatible apps and 484 vulnerable apps) that were reported with potential `DSDK` problems in Sec. 4.3 and 4.4 for the analysis. Since our dataset was crawled in November 2018, we collected the latest release date of those buggy apps on Google Play in early December 2019. We believe that this one-year time frame allows us to test the app updatability by analyzing whether apps have been updated in 2019 or not. We show our finding in the next paragraph.

Finding 5: *Around 20% of the 8,359 buggy apps were never updated in 2019, and 13.7% have been deprecated from Google Play, causing a total of 33.7% apps outdated.* Fig. 19 shows a bar chart of the distribution of apps that were measured with `DSDK` issues in our dataset and their latest release years on Google Play.

According to this figure, 5,539 (66.3%) apps have been updated at least once in 2019, which allows their developers to upgrade `DSDK` versions to fix their `DSDK` problems. However, there are still one third of the measured apps not updatable. Specifically, the latest release years of 1,674 (20%) apps have been 2018, 2017, 2016, and even before 2015. Besides these "old" apps, we find that 1,146 (13.7%) apps are even deprecated from Google Play for various reasons (e.g., being taken down by developer themselves or violating the advertisement policy on Google Play). No matter for what reasons, they are no longer on Google Play due to no further maintenance, whereas their previously downloaded versions could still be in user phones. Both old and deprecated apps incur a large number of outdated apps in the wild, with a total of 33.7% in our dataset. Therefore, it is worthwhile for researchers to further develop techniques for automatically fixing `DSDK` issues in those outdated apps.

5 Implications

In this section, we further present two implications on the qualitative analysis of identified `DSDK` problems and actionable countermeasures for developers.

Implication 1: *Android's original design of the `DSDK` mechanism, despite the good intention, does not satisfy the expectation of developers' real usage.* One major problem is that it is difficult to evolve the `DSDK` versions correctly when apps are updated with new or deprecated APIs. The original `DSDK` design is a *static* mechanism, and there was no automatic mechanism to *dynamically* update the outdated `DSDK` versions. However, quite a number of apps are updated frequently, e.g., 1,448 of the top 10,713 apps studied in 2014 were updated on a bi-weekly basis or even more frequently [36]. In this way, it is challenging for developers to maintain the `DSDK` versions while they are already busy with the functionality update. Moreover, the `addJavascriptInterface()` vulnerabilities reported in Sec. 4.4 indicate that there is a semantic gap between the `targetSdkVersion` design and developers' understanding. Indeed, it is somehow confusing that lower versions of API behaviors would be used even when an app is running on a higher version of the Android platform (see Sec. 2). To our knowledge, this is not the first case where a misunderstanding between Android's design and developers' knowledge happens. Another notable example is that Android once by default exported all content provider components that have no `android:exported` attribute defined, which caused a large number of vulnerable apps [54] since developers did not expect their content provider components to be exported.

It is worth noting that this implication is only our plausible conjecture. Based on the factual analysis results reported in Sec. 4, other conjectures could also be possible. However, no matter what the causes are from the developers' perspective, the consequences remain the same and significant.

Implication 2: *To mitigate the `DSDK` problems, the Android community could take countermeasures from different levels.* We list the following three actionable countermeasures that can be adopted by different stakeholders:

– Google Android could provide better IDE (integrated development environment) to help developers check `DSDK` versions before uploading their apps to the markets. Such checking is ideally automatic and should launch whenever

there are new changes in apps. We have seen a good trend in the recent Android Studio IDE, which performs more user-friendly `DSDK` checking than its predecessor, i.e., the Android Lint plugin in Eclipse.
- The app markets can deploy our approach to perform a quick and mandatory checking of each uploaded app. The suspicious `DSDK` conflicts and recommendations need to be either approved or dismissed. In this way, we can guarantee that developers are at least aware of potential `DSDK` problems in their apps.
- As the last line of defense, end-user Android devices can dynamically upgrade `DSDK` versions in victim apps or enforce mandatory access control [49] so that they are no longer incompatible or vulnerable at the operating system level. This is especially important for the apps no longer maintained (see Sec. 4.6).

6 Threats To Validity

In this section, we summarize some major threats to the validity of our study.

Firstly, same as typical Android static analysis, our approach does not handle Java reflection, dynamic code loading, native code, and complicated code obfuscation. However, some apps may employ these mechanisms to access certain Android APIs. If one such API call has inconsistency issues, a false negative would appear. Since these code protection mechanisms are usually used in malware, our statistical results of popular apps will be less affected and we will consider these mechanisms in our future work.

Secondly, although our bytecode search in Sec. 3.3.2 has minimized false positives caused by `VERSION.SDK_INT` checking and uninvoked third-party libraries, it is theoretically less accurate than dataflow-based approaches. Fortunately, in our deployment model, we can rely on developers to manually check and correct inconsistency reported by our approach. Moreover, as evidenced in Sec. 4.3, the manual effort required in such checking is also limited — around 80% apps are reported with fewer than ten inconsistent API calls each, which is manageable for developers to perform a one-time manual check. Due to this limitation, the measurement results reported in this paper represent an upper bound of all potential `DSDK` problems (under the condition that the common analysis difficulties above are not considered). This satisfies our objective of conducting a comprehensive `DSDK` study, whereas it is not suitable for bug detection.

Thirdly, the consistency detection in this paper focuses on changed APIs, but there are also added and removed Java/Android fields during the SDK evolution. To build the mapping between fields and SDK versions, we found that we can leverage the same document analysis method in Sec. 3.2, because the `api-versions.xml` file also records added, removed, and deprecated fields in all Android classes. By inputting this mapping to our app analysis, we can extend our consistency detection to evolved Android fields as well in our future work.

Lastly, although we have updated the original 2015 dataset with a recent dataset crawled in November 2018 and further checked its updatability in December 2019, we are not able to keep updating it. As a result, the findings reported in this paper may not represent the latest scenarios. We invite other researchers to replicate our findings on more recent datasets.

7 Related Work

In this section, we summarize some related research on declared SDK versions, Android APIs, and Android app static analysis.

7.1 Research on Declared SDK Versions

To the best of our knowledge, there were no systematic studies on declared SDK versions previously, except for some specific studies on the `targetSdkVersion` or `minSdkVersion` attributes in different scenarios. Notably, Wu and Chang [46] showed that due to using outdated `targetSdkVersion` attributes, many Android browser apps were vulnerable to `file://` vulnerabilities. They further demonstrated more security consequences caused by outdated `targetSdkVersion` attributes [47]. Following this line of research, Mutchler et al. [37] conducted a large-scale measurement of multiple vulnerabilities that are affected by the fragmented `targetSdkVersion` attributes. Wei et al. [44] also studied Android fragmentation with the focus on compatibility issues. In particular, our preliminary conference version of this work [51] has motivated three recent follow-up works [30] [27] [40] on detecting compatibility issues caused by inappropriate `minSdkVersion` attributes. Compared to all these works, our study is the first systematic work on measuring all kinds of `DSDK` versions and their (in)consistency with API calls.

7.2 Android API Studies

Besides `DSDK` and fragmentation, our paper is also related to prior studies on Android APIs or SDKs. Among these studies, the work performed by McDonnell et al. [35] is the closest to our paper. They also studied the Android API evolution but focused on how client apps follow Android API changes. In contrast, our focus is the consistency between apps' `DSDK` and API calls. Other related works have studied the correlation between apps' API change and their success [33], the deprecated API usage in Java-based systems [21], the inaccessible APIs in Android framework and their usage in third-party apps [31], and the Android Alarm API usage and their impacts to network latency [12]. In particular, the work conducted by Almeida et al. [12] analyzed `targetSdkVersion` in the apps that invoke Alarm APIs. Additionally, several security papers analyzed the mappings between Android APIs and their permissions [23] [16] [19].

7.3 Android App Static Analysis

A large number of Android studies have leveraged static analysis in many applications over past years. The major methodology can be roughly classified into control-flow based reachability analysis and dataflow-based taint analysis. For the reachability analysis, RiskRanker [26] and Woodpecker [25] are two pioneer representative works in the domains of malware detection and vulnerability discovery, respectively. They tested the reachability from entry points to sink APIs. In contrast, more prior works employed dataflow analysis to taint the propagation flows

of an interested data variable. FlowDroid [15], Amandroid [43], DroidSafe [24], and HornDroid [22] are representative works in this research direction. In particular, FlowDroid and Amandroid have been used or customized in many follow-up static analysis tools (e.g., [53] [17] [41] [28] [27]). One common thing between reachability analysis and dataflow analysis is that they both require to generate an app call graph, the precision of which affects the entire analysis accuracy. However, generating a high-precision call graph requires expensive pointer analysis [43], and the scalability concern is why we proposed lightweight bytecode search for our online vetting of API-SDK inconsistency in this paper.

8 Conclusion and Future Work

In this paper, we conducted a systematic study of declared SDK versions in Android apps, a modern software mechanism that received little attention. We measured the current practice of declared SDK versions or `DSDK` versions in a large set of 22,687 modern apps and the inconsistency between `DSDK` versions and their host apps' API calls. To facilitate the analysis that can be readily deployed by app markets for online vetting, we proposed a robust and scalable approach that operates on the Android bytecode level and employs a lightweight bytecode search for app analysis. We have obtained some interesting new findings, including (i) 4.76% apps do not claim the targeted `DSDK` versions, although this percentage has significantly dropped over recent three years, (ii) around 35% apps under-set the minimum `DSDK` versions and could incur runtime crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above, and (iii) around 2% apps, due to under-claiming the targeted `DSDK` versions, are potentially exploitable by remote code execution, and half of them invoke the vulnerable API via embedded third-party libraries. In the future, we plan to conduct more `DSDK` case studies and report buggy cases to app developers and markets for fixes, and further improve our approach to mitigate some threats to validity.

Acknowledgements We thank editors and all the reviewers for their valuable comments and helpful suggestions. This research/project is supported by the Singapore National Research Foundation under the National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001) and partially supported by a direct grant (ref. no. 4055127) from The Chinese University of Hong Kong.

References

1. aapt: Android Asset Packaging Tool. `http://elinux.org/Android_aapt`
2. The AndroidManifest.xml file. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`
3. API compatibility issues in the emdete/tabulae project. `https://github.com/emdete/tabulae/issues/12`
4. apktool. `https://ibotpeaches.github.io/Apktool/`
5. Enable multidex for apps with over 64K methods. `https://developer.android.com/studio/build/multidex`
6. IDC: Smartphone Market Share. `https://www.idc.com/promo/smartphone-market-share/os`
7. Integrate Flurry SDK for Android. `https://developer.yahoo.com/flurry/docs/integrateflurry/android/`

8. Multiple APK support - Android Developers. https://developer.android.com/google/play/publishing/multiple-apks
9. Openfeint is the largest mobile social gaming network in the world. http://www.openfeint.com/
10. The uses-sdk manifest element. http://developer.android.com/guide/topics/manifest/uses-sdk-element.html
11. Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L.: AndroZoo: Collecting millions of Android apps for the research community. In: Proc. MSR (2016)
12. Almeida, M., Bilal, M., Blackburn, J., Papagiannaki, K.: An empirical study of Android alarm usage for application scheduling. In: Proc. Springer PAM (2016)
13. Android: Distribution dashboard. https://developer.android.com/about/dashboards/
14. Android: Platform codenames, versions, and API levels. https://source.android.com/source/build-numbers.html
15. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: ACM PLDI (2014)
16. Au, K., Zhou, Y., Huang, Z., Lie, D.: PScout: Analyzing the Android permission specification. In: Proc. ACM CCS (2012)
17. Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., Bodden, E.: Mining apps for abnormal usage of sensitive data. In: Proc. ACM ICSE (2015)
18. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in Android and its security applications. In: Proc. ACM CCS (2016)
19. Backes, M., Bugiel, S., Derr, E., McDaniel, P.D., Octeau, D., Weisgerber, S.: On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In: Proc. USENIX Security (2016)
20. Bonett, R., Kafle, K., Moran, K., Nadkarni, A., Poshyvanyk, D.: Discovering flaws in security-focused static analysis tools for Android using systematic mutation. In: Proc. USENIX Security (2018)
21. Brito, G., Hora, A., Valente, M.T., Robbes, R.: Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In: Proc. IEEE SANER (2016)
22. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of Android applications by SMT solving. In: Proc. IEEE EuroS&P (2016)
23. Felt, A., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proc. ACM CCS (2011)
24. Gordon, M.I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., Rinard, M.: Information-flow analysis of Android applications in DroidSafe. In: Proc. ISOC NDSS (2015)
25. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock Android smartphones. In: Proc. ISOC NDSS (2012)
26. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: Scalable and accurate zero-day Android malware detection. In: Proc. ACM MobiSys (2012)
27. He, D., Li, L., Wang, L., Zheng, H., Li, G., Xue, J.: Understanding and detecting evolution-induced compatibility issues in Android apps. In: Proc. ACM ASE (2018)
28. Jia, Y., Chen, Q., Lin, Y., Kong, C., Mao, Z.: Open doors for Bob and Mallory: Open port usage in Android apps and security implications. In: Proc. IEEE EuroS&P (2017)
29. Lei, L., He, Y., Sun, K., Jing, J., Wang, Y., Li, Q., Weng, J.: Vulnerable Implicit Service: A Revisit. In: Proc. ACM CCS (2017)
30. Li, L., Bissyandé, T.F., Wang, H., Klein, J.: CiD: Automating the detection of API-related compatibility issues in Android apps. In: Proc. ACM ISSTA (2018)
31. Li, L., Bissyandé, T.F., Traon, Y.L., Klein, J.: Accessing inaccessible Android APIs: An empirical study. In: Proc. IEEE ICSME (2016)
32. Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W.: LibD: Scalable and precise third-party library detection in Android markets. In: Proc. ACM ICSE (2017)
33. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M.D., Oliveto, R., Poshyvanyk, D.: API change and fault proneness: A threat to the success of Android apps. In: Proc. ACM FSE (2013)
34. Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: MaMaDroid: Detecting Android malware by building markov chains of behavioral models. In: Proc. ISOC NDSS (2017)
35. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the Android ecosystem. In: Proc. IEEE ICSM (2013)

36. McIlroy, S., Ali, N., Hassan, A.E.: Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. In: Empirical Software Engineering, Volume 21, Issue 3 (2016)
37. Mutchler, P., Safaei, Y., Doupe, A., Mitchell, J.: Target fragmentation in Android apps. In: Proc. IEEE Mobile Security Technologies (MoST) (2016)
38. Octeau, D., Jha, S., McDaniel, P.: Retargeting Android applications to Java bytecode. In: Proc. ACM FSE (2012)
39. Pan, X., Wang, X., Duan, Y., Wang, X., Yin, H.: Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps. In: Proc. ISOC NDSS (2017)
40. Scalabrino, S., Bavota, G., Linares-Vásquez, M., Lanza, M., Oliveto, R.: Data-Driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study. In: Proc. MSR (2019)
41. Shao, Y., Ott, J., Jia, Y.J., Qian, Z., Mao, Z.M.: The misuse of Android Unix domain sockets and security implications. In: Proc. ACM CCS (2016)
42. Tiwari, A., Prakash, J., Groß, S., Hammer, C.: A Large Scale Analysis of Android — Web Hybridization. In: Journal of Systems and Software, Volume 170 (2020)
43. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In: Proc. ACM CCS (2014)
44. Wei, L., Liu, Y., Cheung, S.C.: Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In: Proc. ACM ASE (2016)
45. Wei, T., Zhang, Y., Xue, H., Zheng, M., Ren, C., Song, D.: Sidewinder Targeted Attack against Android in The Golden Age of Ad Libraries. In: Black Hat USA (2014)
46. Wu, D., Chang, R.K.C.: Analyzing Android browser apps for file:// vulnerabilities. In: Proc. Springer Information Security Conference (ISC) (2014)
47. Wu, D., Chang, R.K.C.: Indirect file leaks in mobile applications. In: Proc. IEEE Mobile Security Technologies (MoST) (2015)
48. Wu, D., Chang, R.K.C., Li, W., Cheng, E.K.T., Gao, D.: MopEye: Opportunistic monitoring of per-app mobile network performance. In: Proc. USENIX Annual Technical Conference (2017)
49. Wu, D., Cheng, Y., Gao, D., Li, Y., Deng, R.H.: SCLib: A practical and lightweight defense against component hijacking in Android applications. In: Proc. ACM Conference on Data and Applications Security and Privacy (CODASPY) (2018)
50. Wu, D., Gao, D., Chang, R.K.C., He, E., Cheng, E.K.T., Deng, R.H.: Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In: Proc. ISOC NDSS (2019)
51. Wu, D., Liu, X., Xu, J., Lo, D., Gao, D.: Measuring the declared SDK versions and their consistency with API calls in Android apps. In: Proc. Springer International Conference on Wireless Algorithms, Systems, and Applications (WASA) (2017)
52. Wu, D., Luo, X., Chang, R.K.C.: A Sink-driven Approach to Detecting Exposed Component Vulnerabilities in Android Apps. In: CoRR arXiv, abs/1405.6282 (2014)
53. Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., Enck, W.: AppContext: Differentiating malicious and benign mobile app behaviors using context. In: Proc. ACM ICSE (2015)
54. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in Android applications. In: Proc. ISOC NDSS (2013)