6-2017

Measuring the declared SDK versions and their consistency with API calls in android apps

Daoyuan WU
Singapore Management University, dywu.2015@phdis.smu.edu.sg

Ximing LIU
Singapore Management University, xmliu.2015@phdis.smu.edu.sg

Jiayun XU
Singapore Management University, jyxu.2015@phdis.smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

Measuring the Declared SDK Versions and Their Consistency with API Calls in Android Apps

Daoyuan Wu(✉), Ximing Liu, Jiayun Xu, David Lo, and Debin Gao

School of Information Systems, Singapore Management University, Singapore, Singapore
{dywu.2015,xmliu.2015,jyxu.2015,davidlo,dbgao}@smu.edu.sg

Abstract. Android has been the most popular smartphone system, with multiple platform versions (e.g., KITKAT and Lollipop) 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 make a first effort to study this modern software mechanism. Our objective is to measure the current practice of the declared SDK versions (which we term as `DSDK` versions afterwards) in real apps, and the consistency between the `DSDK` versions and their app API calls. To this end, we perform a three-dimensional analysis. First, we parse Android documents to obtain a mapping between each API and their corresponding platform versions. We then analyze the `DSDK`-API consistency for over 24K apps, among which we pre-exclude 1.3K apps that provide different app binaries for different Android versions through Google Play analysis. Besides shedding light on the current `DSDK` practice, our study quantitatively measures the two side effects of inappropriate `DSDK` versions: (i) around 1.8K apps have API calls that do not exist in some declared SDK versions, which causes runtime crash bugs on those platform versions; (ii) over 400 apps, due to claiming the outdated targeted `DSDK` versions, are potentially exploitable by remote code execution. These results indicate the importance and difficulty of declaring correct `DSDK`, and our work can help developers fulfill this goal.

Keywords: Android bug detection · Android app security

1 Introduction

Recent years have witnessed the extraordinary success of Android, a smartphone operating system owned by Google. At the end of 2013, Android became the most sold phone and tablet OS. As of 2015, Android evolved into the largest installed base of all operating systems. Along with the fast-evolving Android, its fragmentation problem becomes more and more serious. Although new devices

X. Liu, J. Xu—These two author names are in alphabetical order.

ship with the recent Android versions, there are still huge amounts of existing devices running old Android versions [1].

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

In this paper, we make a first attempt to systematically study the `DSDK` mechanism. In particular, our objective is to measure the current practice of `DSDK` versions in real apps, and the consistency between `DSDK` versions and their apps' API calls. To this end, we perform a three-dimensional analysis that analyzes Google Play, Android documents, and each individual app. We use a large dataset that contains over 24K apps crawled from Google Play in July 2015. Our study sheds light on the current `DSDK` practice and quantitatively measures the two side effects of inappropriate `DSDK` versions.

We summarize the contributions of this paper as follows:

- (*New problem*) We study a modern software mechanism, i.e., allowing apps to declare the supported platform SDK versions. In particular, we are the first to measure the declared SDK versions and their consistency with API calls in Android apps.
- (*New understanding*) We give the first demystification of the `DSDK` mechanism and its two side effects of inappropriate `DSDK` versions.
- (*Hybrid approach*) We propose a three-dimensional analysis method that operates at both Google Play, Android document, and Android app levels.
- (*Insightful results*) We have three major findings, including (i) around 17% apps do not claim the targeted `DSDK` versions or declare them wrongly, (ii) around 1.8K apps under-set the minimum `DSDK` versions, causing them crash when running on lower Android versions, and (iii) over 400 apps under-claim the targeted `DSDK` versions, making them potentially exploitable by remote code execution.

2 Demystifying the Declared SDK Versions and Their Two Side Effects

In this section, we first demystify the declared platform SDK versions in Android apps, and then explain their two side effects if inappropriate `DSDK` versions are being used.

2.1 Declared SDK Versions in Android Apps

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

Listing 1.1. The syntax for declaring the platform SDK versions in Android apps.

Listing 1.1 illustrates how to declare the supported platform SDK versions in Android apps by defining the `<uses-sdk>` element in apps' manifest files (i.e., `AndroidManifest.xml`). 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 the SDK version 5.0, he/she sets its value as 21 (the API level of Android 5.0 is 21). Since each API level has a precise mapping of the corresponding SDK version [2], we do not use another term, *declared API level*, to represent the same meaning of `DSDK` throughout this paper.

We explain the three `DSDK` attributes as follows:

- The `minSdkVersion` integer specifies the minimum platform API level required for the 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 the app targets at. An important *implication* of this attribute is that Android adopts the back-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 developers expect, even when apps run on newer platforms. It is worth noting that if this attribute is not set, the default value equals to the value of `minSdkVersion`.
- 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). That said, 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 download. Not that if this attribute is not set, it implies no any restriction on the maximum platform API level.

2.2 Two Side Effects of Inappropriate DSDK Versions

Figure 1 illustrates the two side effects of inappropriate `DSDK` versions. We first explain the symbols used in this figure, and then describe the two side effects

in the subsequent paragraphs. 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 $maxSDK$.

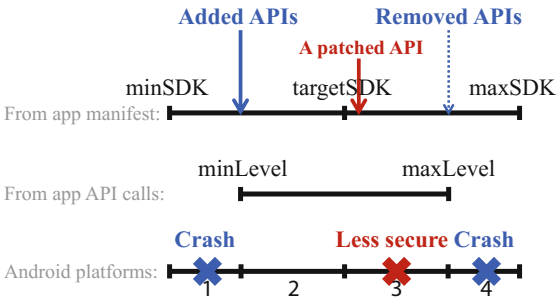


Fig. 1. Illustrating the two side effects of inappropriate `DSDK` versions. (Color figure online)

Side Effect I: Causing Runtime Crash Bugs. The blue part of Fig. 1 shows two scenarios in which inappropriate `DSDK` versions can cause app crash. The first scenario is $minLevel > minSDK$, which means a new API is introduced after the $minSDK$. Consequently, when an app runs on the Android platforms between $minSDK$ and $minLevel$ (marked as the block 1 in Fig. 1), it will crash. We verified this case by using the `VpnService.Builder.addDisallowedApplication()` API, which was introduced at Android 5.0 at the API level 21. We called this API at the MopEye app [3] and ran MopEye on an Android 4.4 device. When the app executed the `addDisallowedApplication()` API, it crashed with the `java.lang.NoSuchMethodError` exception.

The second crash scenario is $maxSDK > maxLevel$, which means an old API is removed at the $maxLevel$. Similar to the first scenario, the app will crash when it runs on the Android platforms between $maxLevel$ and $maxSDK$.

Side Effect II: Making Apps Less Secure. The red part of Fig. 1 shows the scenario in which inappropriate `DSDK` versions cause apps fail to be patched that they originally should be able to. Suppose an app calls an API (e.g., `addJavascriptInterface()` [4]) that is vulnerable before the $targetSDK$. However, if the `targetSdkVersion` of the app is lower than the patched API level, Android will still take the compatibility behaviors, i.e., the non-patched API behavior in this case, even when the app runs on the patched platforms (between $targetSDK$ and $maxLevel$). Some such vulnerable app examples are available in https://sites.google.com/site/androidrce/.

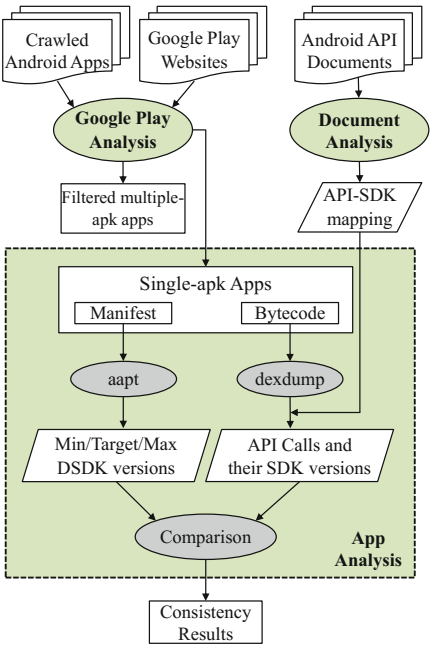


Fig. 2. The overview of our methodology.

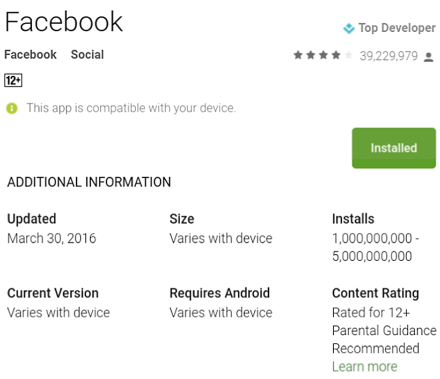


Fig. 3. The Facebook app's Google Play page (with irrelevant contents removed).

Table 1. The dataset of our study.

	#	Note
All crawled apps	24,426	The initial dataset
Multiple-apk apps	1,301	Filtered apps
Single-apk apps	23,125	The final dataset

3 Methodology

In this section, we present an overview of our methodology and its three major components.

3.1 Overview

Figure 2 illustrates the overall design of our method. It performs the analysis at three levels. First, we crawl and analyze each app's Google Play page to filter *multiple-apk* apps that provide different app binaries (i.e., *apks*) for different Android platforms. Since each apk of these apps is tailored for a particular Android version, its declared platform SDK version is no longer important. We therefore exclude these multiple-apk apps for further analysis. Second, we parse Android API documents to build a complete mapping between each API and their corresponding platform versions. We call this mapping the *API-SDK mapping*.

In the final app analysis phase, we first extract apps' declared SDK versions and API calls, then leverage the existing API-SDK mapping to infer the range of SDK versions from API calls, and finally compare these two SDK versions (i.e., the declared SDK versions and the SDK versions inferred from API calls).

The output is the (in)consistency results between declared SDK versions and API calls, which can be further leveraged to detect bugs and vulnerabilities.

3.2 Google Play Analysis

Design and Implementation. The main objective of running Google Play analysis is to filter multiple-apk apps. We explain this step using a representative Google Play page, the Facebook app's page as shown in Fig. 3. We can notice that three attributes ("Size", "Current Version", and "Requires Android") all have the same value of "Varies with device". This indicates that Facebook employs the multiple-apk approach to handle the app compatibility over different versions of Android platforms. The apps that do not have the value of "Varies with device" are thus the single-apk apps.

To implement the Google Play analysis, we write Python scripts based on our previous codes [5,6] and Selenium, a web browser automation tool. We use Selenium's Firefox driver to load each app's Google Play page, and extract the attribute values we are interested by parsing the page's HTML source.

Dataset. Table 1 lists the dataset used in this paper. We have crawled 24,426 apps from Google Play in July 2015. We run Google Play analysis for all these apps, among which we identify and filter 1,301 multiple-apk apps. Therefore, the remaining 23,125 single-apk apps assemble our final dataset, which will be further analyzed in Sect. 3.4. Unless stated otherwise, we refer to our dataset as these 23,125 apps in this paper.

3.3 Android Document Analysis

Method. To build the API-SDK mapping, we analyze Android SDK documents based on a previous work [7]. Specifically, we first build a list of all Android APIs and the corresponding platform versions they were introduced to by parsing a SDK document called `api-versions.xml`. This file covers both initial APIs (those introduced in the first Android version) and other newly added APIs in subsequent Android versions. We further count the API change (e.g., deprecated and removed APIs) by analyzing the HTML files in the `api_diff` directory.

After running the document analysis for 23 Android versions (from 1.0 to 6.0), we recorded a total of 30,083 APIs, out of which 794 APIs were afterwards deprecated and 190 APIs were finally removed. However, we found that the lists of deprecated and removed APIs are not fully accurate, probably due to the mistakes made by Google developers when they wrote SDK documents. For example, the `removeAccount(Account, Callback, Handler)` API in the `AccountManager` class was recorded as "removed in SDK version 22" in the documents, but actually it is still available in the SDK version 23. This result implies that such a document-based analysis employed by the previous work [7] requires further improvement. As a future work, we will explore to retrieve the API-SDK mapping directly from each SDK `jar` file. In this paper, since the list

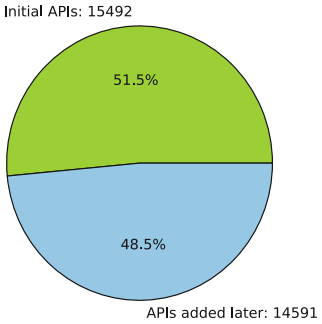


Fig. 4. The comparison between initial and added APIs.

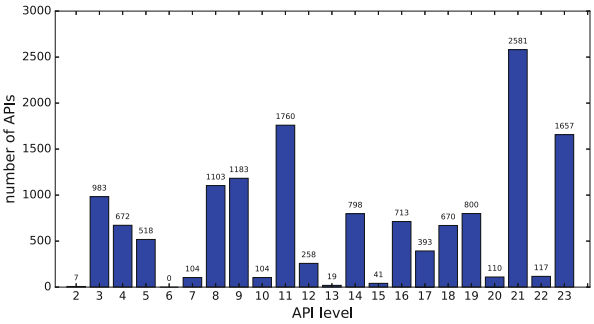


Fig. 5. The distribution of added Android APIs.

of added APIs is accurate, we use only this part of results for the subsequent `DSDK` analysis in Sect. 4.

Results. We now present the results of document analysis. Figure 4 shows the comparison between the initial Android APIs and those subsequently added APIs. We can see that almost half of all APIs were added afterwards. This indicates that Android evolves dramatically along the whole process. In Fig. 5, we further plot the distribution of those subsequently added APIs since API level 2. Android 5.0 (API level 21) changed most, with 2,581 new API introduced. The following two most changed versions are Android 3.0 (API level 11) and Android 6.0 (API level 23), with 1,760 and 1,657 new APIs, respectively.

3.4 Android App Analysis

Retrieving Declared SDK Versions. We leverage `aapt` (Android Asset Packaging Tool) to retrieve `DSDK` versions *directly* from each app without extracting the manifest file. This method is more robust than the traditional `apktool`-based manifest extraction employed in many other works. Indeed, our `aapt`-based approach can successfully analyze all 23,125 apps, whereas a recent work [8] shows that `apktool` fails six times in the analysis of top 1K apps.

In the course of implementation, 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 by-default rules (see Sect. 2.1) for the non-defined `minSdkVersion` and `targetSdkVersion`. 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.

Extracting API Calls and Their SDK Versions. To extract API calls from apps' bytecodes, we first translate the compressed bytecodes into readable texts by using the `dexdump` tool. We then use a set of Linux bash commands to extract each app's method calls from their `dexdump` outputs.

With the extracted API calls, we use the API-SDK mapping to compute their corresponding SDK versions (i.e., `minLevel` and `maxLevel`, as explained in Fig. 1). To compute the `minLevel`, we calculate a maximum value of all API calls' added SDK versions. Similarly, to compute the `maxLevel`, we calculate a minimum value of all API calls' removed SDK versions. If an API is never removed, we set its removed SDK version to a large flag value (e.g., 100,000).

During the experiments, we find that it is necessary to exclude library codes' API calls from host apps' own API calls. Libraries such as Android Support Library provide the stub implementation of higher-version APIs on lower-version platforms to ensure the backward-compatibility of higher-version APIs. If an app is running on a higher-version platform, the library directly calls the corresponding API. Otherwise, the library calls the stub implementation, which actually does nothing but would not crash the app. Since we currently do not differentiate such control-flow information, we exclude library codes for the consistency analysis.

Comparing Consistency. With the `DSDK` and API level information, it is easy to compare their consistency. We compute the following three kinds of inconsistency (as previously mentioned in Sect. 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.
- `maxSdkVersion` > `maxLevel`: the `maxSdkVersion` is set too large and the app would crash when it runs on platform versions between `maxLevel` and `maxSdkVersion`.

4 Evaluation

Our evaluation aims to answer the following three research questions:

RQ1: What are the *characteristics* of the `DSDK` versions in real-world apps?
RQ2: What are the *characteristics* of the API calls in real-world apps?
RQ3: Could we identify the *inconsistency* between `DSDK` versions and API calls in real apps? In particular, could we discover crash bugs and potential security vulnerabilities?

4.1 RQ1: Characteristics of the Declared SDK Versions

In this section, we report a total of four findings regarding the RQ1.

Finding 1: Not all apps define the `minSdkVersion` and `targetSdkVersion` attributes, and 16.5% apps do not claim the `targetSdkVersion` attributes. From Table 2, we can see that rare apps (about 0.22%) do not

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

	# Non-defined	% Non-defined
minSdkVersion	51	0.22%
targetSdkVersion	3,826	16.54%
maxSdkVersion	23,109	99.93%

define the minSdkVersion, while a noticeable portion of apps (over 15%) do not define the targetSdkVersion. Out of these apps, 48 apps declare neither the minSdkVersion, nor the targetSdkVersion. Consequently, the values of both minSdkVersion and targetSdkVersion will be assigned to "1" by the system. We also notice that almost all apps (over 99%) do not define the maxSdkVersion. This result is reasonable because, as we described in Sect. 2.1, the maxSdkVersion attribute is strongly suggested *not* to define.

Finding 2: There are 53 outlier targetSdkVersion values. We also find out some declared targetSdkVersion are outlier values. One app defines its targetSdkVersion as 0, which is lower than the minSdkVersion. Others' targetSdkVersion are larger than the newest SDK version (API level 23 at that time). Some apps declare targetSdkVersion as 24, 25, 26 or larger, however, these SDK versions have not been released yet in year 2015. Even more surprisingly, one app sets the targetSdkVersion value to "10000". In general, targetSdkVersion should be always greater than or equal to the minSdkVersion, but 34 apps have negative targetSdkVersion- minSdkVersion value.

Finding 3: The minimal platform versions most apps support are Android 2.3 and 2.2, whereas the most targeted platform versions are Android 4.4 and 5.0. In Figs. 6 and 7, we plot the distribution of minSdkVersion and targetSdkVersion, respectively. We can see that most apps (around 85%) have minSdkVersion lower than or equal to level 11 (i.e., Android 3.0), which means that they can run on the majority of Android devices in the market [1]. Moreover, the minimal platform versions most apps support are Android 2.3 and 2.2. Figure 7 shows that more than 89% apps test their apps on platform versions larger than Android 4.0, and the most targeted platform versions are Android 4.4 and 5.0.

Finding 4: The mean version difference between targetSdkVersion and minSdkVersion is 8. We define a new metric called lagSdkVersion to measure the version difference between targetSdkVersion and minSdkVersion, as shown in Eq. 1.

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

After removing negative targetSdkVersion values and outliers, we draw the CDF (Cumulative Distribution Function) plot of lagSdkVersion in Fig. 8.

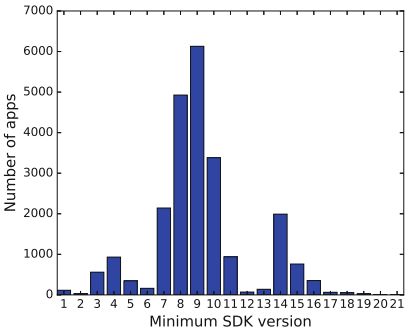


Fig. 6. Distribution of `minSdkVersion`.

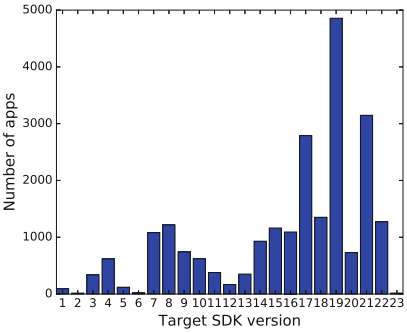


Fig. 7. Distribution of `targetSdkVersion`.

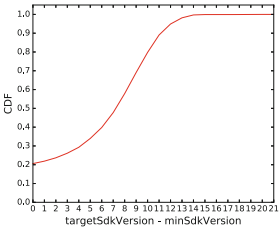


Fig. 8. CDF plot of `lagSdkVersion`.

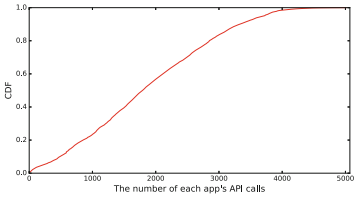


Fig. 9. CDF plot of the number of each app's API calls.

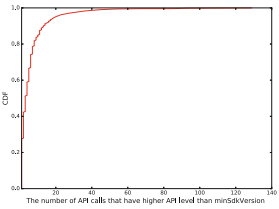


Fig. 10. CDF plot of each app's number of API calls that have higher API level than `minSdkVersion`.

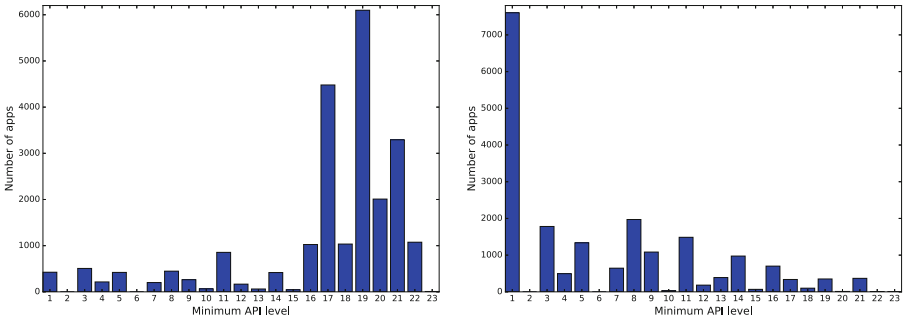
It shows that more than 20% apps have equal `targetSdkVersion` and `minSdkVersion`. Furthermore, the majority of apps (more than 95% apps) have a `lagSdkVersion` less than 12.

4.2 RQ2: Characteristics of the API Calls

In this section, we briefly present two more findings related to the RQ2. It is worth noting that here we consider all API calls that include the API calls in libraries.

Finding 5: Around 500 apps call less than 50 APIs, making them lightweight apps. On the other hand, half of apps call over 1.8K APIs. We find that 446 apps call less than 50 APIs. The majority of them are about user interface improvement, such as system theme and wallpaper apps. These apps are regarded as lightweight ones that have less dependency on the SDK versions. Additionally, many other apps contain several thousand API calls. We plot the distribution of apps by API call numbers in Fig. 9.

Finding 6: Library codes contribute more higher-version API calls than apps' own codes. Libraries such as Android support library provide



(a) All API calls with library code. (b) App's own API calls without library code.

Fig. 11. The distribution of `minLevel` that is calculated from API calls w/o library.

backward-compatible versions of Android framework APIs, as well as the features that are only available through the library APIs. Each support library is backward-compatible to a specific API level, which allows an app that contains higher-version APIs run correctly on a lower version of Android system. Figure 11(a) shows that distribution of the `minLevel` of API calls with the library code, whereas Fig. 11(b) presents the distribution of the `minLevel` of API calls without the library code. By analyzing and de-compiling the support library, we found that they can redirect the APIs calls in a higher-version SDK to some similar APIs which are already in a lower SDK or to an empty function.

4.3 RQ3: Inconsistency Results

In this section, we report two important findings regarding the RQ3.

Finding 7: Around 1.8K apps under-set the `minSdkVersion` value, causing them would crash when they run on lower Android versions. We find that 1,750 apps have over five API calls, the levels of which are larger than the declared `minSdkVersion`. In 692 apps, more than ten API calls have higher API level than `minSdkVersion`. In Fig. 10, we draw the CDF plot of the number of API calls that have higher API level than `minSdkVersion`. Based on this figure, we find that several apps have more than 50 API calls whose API level is higher than `minSdkVersion`.

Finding 8: Around 400 apps fail to update their `targetSdkVersion` values, making them potentially exploitable by remote code execution. The `addJavascriptInterface()` API [4] has a serious security issue. By exploiting this API, attackers are able to inject malicious codes, which may obtain any information from SD card. Google later fixed this bug on Android 4.2 and afterward. However, as mentioned in the side effect II, if an app has the `targetSdkVersion` lower than 17 and calls this API, the system will still call the vulnerable API even when running in Android 4.2 and afterward. In our dataset, we find that 909 apps call the `addJavascriptInterface()` API. Among these

apps, 413 apps are vulnerable, which may cause privacy information leakage. In particular, out of these 413 apps, 238 apps do not define the `targetSdkVersion` attribute (i.e., `targetSdkVersion` is null).

5 Threats to Validity

In this section, we discuss a couple of threats to the validity of our study.

First, we have not performed the control-flow analysis to determine whether an API call will be invoked only when running on certain Android versions. During the experiments, we noticed that many library codes take `if-else` blocks to call higher-version APIs on when the app is running on the corresponding versions. To mitigate its impact to our analysis, we currently exclude the library codes for consistency analysis (Sect. 3.4), and use a threshold value to minimize the potential version-related `if-else` blocks in app codes (Sect. 4.3).

Apps may employ Java reflection to call private Android APIs [9] that are not included in the SDK but contained in Android framework. Similarly, developers may use native codes to access Android APIs. Currently we have not handled these two cases and leave them as our future work.

Our assumption in Sect. 3.1 that multiple-apk apps do not have compatibility issues may not be always true. In particular, developers may provide only one apk for several Android platforms to share. In this case, those shared apks are similar to single-apk apps.

6 Related Work

Our paper is mainly related to prior works that also study Android APIs or SDKs. The work performed by McDonnell et al. [7] is the closest to our paper. They studied the Android API evolution and how client apps follow Android API changes, which is different from our focus on the consistency between apps' `DSDK` and API calls. In the methodology part, we followed their document analysis method for extracting the API-SDK mapping. But in the future we plan to directly analyze Android SDKs instead of documents for more accurate mapping extraction. Other related works have studied the coefficient between apps' API change and their success [10], the deprecated API usage in Java-based systems [11], and the inaccessible APIs in Android framework and their usage in third-party apps [12]. Two recent works [13,14] also focused on the fragmentation issues in Android. Compared to all these works, our study is the first systematic work on `DSDK` versions and their consistency with API calls.

7 Conclusion and Future Work

In this paper, we made a first effort to systematically study the declared SDK versions in Android apps, a modern software mechanism that has received little attention. We measured the current practice of the declared SDK versions

or DSDK versions in a large dataset of apps, and the consistency between the DSDK versions and their app API calls. To facilitate the analysis, we proposed a three-dimensional analysis method that operates at both Google Play, Android document, and Android app levels. We have obtained some interesting and novel findings, including (i) around 17% apps do not claim the targeted DSDK versions or declare them wrongly, (ii) around 1.8K apps under-set the minimum DSDK versions, causing them would crash when running on lower Android versions, and (iii) over 400 apps under-claim the targeted DSDK versions, making them potentially exploitable by remote code execution. In the future, we plan to contact the authors of the apps to inform them about the detected issues and collect their feedback, release a publicly available tool to let app developers detect and fix issues, and improve our approach to further mitigate the threats to validity (e.g., by designing and incorporating a suitable control-flow analysis technique).

References

1. Android: Dashboards. https://developer.android.com/about/dashboards/
2. Android: Platform codenames, versions, and API levels. https://source.android.com/source/build-numbers.html
3. Wu, D., Li, W., Chang, R., Gao, D.: MopEye: monitoring per-app network performance with zero measurement traffic. In: CoNEXT Student Workshop (2015)
4. Drake, J.: On the WebView addJavascriptInterface saga (2014). http://www.droidsec.org/news/2014/02/26/on-the-webview-addjsif-saga.html
5. Wu, D., Chang, R.K.C.: Analyzing android browser apps for:// vulnerabilities. In: Chow, S.S.M., Camenisch, J., Hui, L.C.K., Yiu, S.M. (eds.) ISC 2014. LNCS, vol. 8783, pp. 345–363. Springer, Cham (2014). doi:10.1007/978-3-319-13257-0_20
6. Wu, D., Chang, R.K.C.: Indirect file leaks in mobile applications. In: Proceedings of IEEE Mobile Security Technologies (MoST) (2015)
7. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the android ecosystem. In: Proceedings of IEEE ICSM (2013)
8. Wu, D., Luo, X., Chang, R.K.C.: A sink-driven approach to detecting exposed component vulnerabilities in android apps. CoRR abs/1405.6282 (2014)
9. Andrew: Hacking the "private" Android API. http://andrewoid.blogspot.com/2008/12/hacking-android-api.html
10. 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: Proceedings of ACM FSE (2013)
11. Brito, G., Hora, A., Valente, M.T., Robbes, R.: Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In: Proceedings of IEEE SANER (2016)
12. Li, L., Bissyandé, T.F., Traon, Y.L., Klein, J.: Accessing inaccessible android APIs: an empirical study. In: Proceedings of IEEE ICSME (2016)
13. Mutchler, P., Safaei, Y., Doupe, A., Mitchell, J.: Target fragmentation in android apps. In: Proceedings of IEEE Mobile Security Technologies (MoST) (2016)
14. Wei, L., Liu, Y., Cheung, S.C.: Taming android fragmentation: characterizing and detecting compatibility issues for android apps. In: Proceedings of ACM ASE (2016)