MAD-API: Detection, Correction and Explanation of API Misuses in Distributed Android Applications

Tianyue Luo[1], Jingzheng Wu[1,2], Mutian Yang[1], Sizhe Zhao[3], Yanjun Wu[1,2(✉)], and Yongji Wang[2,3]

[1] Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing, China
{tianyue,yanjun}@iscas.ac.cn
[2] State Key Laboratory of Computer Sciences, Beijing, China
[3] Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. Android API is evolving continuously, including API updates, deletion, addition and changes. Unfortunately, we find that the distributed Android applications (apps) often fail to keep pace with the API evolution. Specifically, the apps usually involve the APIs that are out of date, which potentially cause the apps or Android system to behave abnormally, leak sensitive information or crash down. We call this issue that making the Android phones unreliable as API misuse. To investigate the universality of this issue and detect the defective apps in the wild, we propose an automated framework MAD-API that consists of a detection method that identifies API misuses in apps and a recommendation method to trace the latest API status and correct the misuses. We implement MAD-API based on 13 Android versions, and evaluate it with the top 10,000 Android apps. According to the evaluation, 93.13% of the evaluated apps suffer from API misuse problems, and the total number of API misuses is 1,241,831. In addition, apps with larger size have more API misuses. Worst of all, some APIs are misused all the time. The results indicate that (1) the API misuse issue widely exists in distributed apps, (2) MAD-API is able to detect API misuses in Android apps effectively, and (3) MAD-API also help developers trace the defective APIs in their distributed apps conveniently and correct them immediately.

1 Introduction

Android is the most popular mobile operating system (OS) in the world, partly because of the numerous applications (apps) it provides and supports. New apps come out everyday, and are developed with the Android Application Program Interfaces (APIs) [1]. Android API is a feature rich application interface set that allows developers to build innovative, genius and useful apps for mobile devices. With thousands of developers' efforts, Android API is continuously evolving, including new features, improvements or deletions of imperfect codes.

M. Aiello et al. (Eds.): AIMS 2018, LNCS 10970, pp. 123–140, 2018.
https://doi.org/10.1007/978-3-319-94361-9_10

Taking Android 5.0.0 as an example, compared with Android 4.4.4, the APIs extend 15%, which implies many packages, classes and methods are newly introduced.

```
1 ActivityManager am = (ActivityManager) context.getSystemService(Context.
  ↪ ACTIVITY_SERVICE);
2 ArrayList<RecentTaskInfo> apps = (ArrayList<RecentTaskInfo>)
  ↪ am.getRecentTasks(Integer.MAX_VALUE,
  ↪ ActivityManager.RECENT_WITH_EXCLUDED);
```

Listing 1: Typical API Change in the Android System.

Unfortunately, the distributed Android apps cannot keep pace with this evolution. We find an issue that many developers do not keep on maintaining and updating the API calls existed in the apps that have been already distributed, thus causes the violation of API's contract. In this paper, we define such issue as API misuse issue. Further more, the API misuse issue can introduce the unexpected problems such as compatibility problem, security problem and reliable problem. For instance, *getRecentTasks* which is a method located in the class *ActivityManager* as shown in Line 1 of Listing 1, returns a list of the tasks that the user has recently launched in Line 2, with the most recent being the first and older ones after in order. Before Android API level 21, Google official developer's guide announce that *getRecentTasks* is only intended for debugging and presenting task management user interfaces, which *should never be used* for the core logic in an application, such as making decisions on different behaviors based on the information found here. However, *getRecentTasks* was deprecated in Android API level 21 because of security issues, meaning it is no longer available to third party apps with the explanation that *it can leak personal information to the caller*. Therefore, the apps calling this API will probably get into a dangerous situation if the developers fail to deal with it immediately.

Detecting API misuses in software is a classic problem of software engineering. Many researches have been presented for analyzing APIs and detecting misuses in various systems. For example, SAFE_WAPI [4] is a static analyzer that detects API misuses in JavaScript web applications, and aims to help developers write safe JavaScript web applications using vendor-specific Web APIs. MUSE (Method USage Examples) [13] is an approach for mining and ranking actual code examples, which shows how to use a specific method and aims to reduce API misuses by providing concrete method usages. [18] proposes a semi-automated framework that consists of a policy terminology-API method map linking policy phrases to API methods. It aims to help app developers check their privacy policies against their apps' code for consistency. Meanwhile, API related analysis techniques such as machine learning based approach SISE [19], Chucky [24], and Drebin [2,5,10,14–16,25,27] mainly focus on API document augmentation, missing checks exposed in source code and malware API respectively. While these approaches are efficient and scalable, they are neither suitable to help developers keep pace with the rapid Android API evolution, nor to correct the misused APIs.

To investigate the universality of this issue and detect the defective apps in the wild, in this paper, we propose an automated framework MAD-API (Misuse Android API Detector and Corrector) including a detection method to identify API misuses in apps according to the latest API status. Firstly, MAD-API extracts APIs for each Android version, and classifies Android APIs into two categories: abnormal, which includes all API methods marked *@deprecated*, *@hide* and *@removed*, and normal, which includes all other methods. Abnormal APIs may introduce risks and security vulnerabilities when used [21,23]. Secondly, MAD-API reverses apps into *"smali"* byte-code and extracts the used APIs. Thirdly, the misused APIs are detected from these two API sets. Finally, the correct APIs with understandable explanations are recommended to the developers.

Our Findings. We present an implementation of MAD-API based on 13 Android versions, and found 316,093 Android APIs in total including 3 suspicious categories of APIs, e.g., 5,153 *@deprecated*, 19,896 *@hide* and 52 *@removed*. Further more, MAD-API are evaluated with the top 10,000 Android apps distributed in 7 popular app stores and found that: (1) 9313 out of the 10,000 apps suffer from API misuse problems, and the total number of API misuses is 1,241,831, (2) the more complicated the apps are, the more misuses they have, and (3) some APIs are misused all the time.

Improvements. In practice, a detection system must not only detect the misused APIs, but also provide explanations and improvements for the detected misuses. In the case of MAD-API, to help developers use API correctly, it recommends the correct APIs to replace the misused ones based on the detected results. The explanations and improvements for the misused APIs can be derived from the Android documentations and the source code comments. As additional benefits, the generated improvements can also help developers improve the relevant API uses in app programming.

Contributions. Specifically, we made the following contributions in this paper.

- **New Findings.** Based on the static analysis of the source code of the Android system and the reverse analysis of applications, we systematically analyzed the API misuse problem, a serious risk in application programming. We found that the problem of the misused API is extremely serious. In particular 93.13% apps suffer from the API misuse problem.
- **Explainable Results.** The proposed MAD-API provides an explainable improvements for each misused API. With the explanations and recommendations, developers can eliminate Android API misuses and improve the code quality of apps.
- **Implementation, Evaluation, and Application.** We implemented MAD-API based on 13 Android versions and evaluated it using the top 10,000 apps in Android market. The results show that MAD-API is effective to detect API misuses and generate explainable improvements for the Android application developers.

2 Background

2.1 Android API Evolution

Android has more than 100 open source projects. The popularity is mainly because the numerous apps it supports, which interact with underlying Android system by API provided by the Android platform. The Android API contains a feature rich set of packages, classes, and methods, and allows developers to build innovative, useful and efficient apps for mobile devices. Google who developed Android system provides a regular updates and improvements to the APIs.

As the Android platform evolves and new versions are released frequently, Android APIs is evolving continuously, including API updates, deletion, addition and changes [17]. The first commercial version, Android 1.0, was released in September 2008. Until now, the latest version is Android 9.0 with API level 28. Taking Android 6.0.0 for example, compared with Android 5.0.0 and Android 1.6, the API growths by 5% and 68% respectively, meaning thousands of packages, classes and methods are newly introduced.

2.2 API Usage in Programming

Developers build innovative apps using various APIs provided by the Android rich application framework. To be more successful on Android market, the published apps need to adapt to various device configurations, including different languages, screen sizes, and versions. While the latest versions of Android often provide great APIs for apps, developers should continue to support older versions of Android in programming until more devices get updated.

However, by analyzing the apps published in the real world, we found that the API misuse is extremely serious, and 93.13% of the Top 10,000 apps suffer from the API misuse problem. In this paper, we design MAD-API and show that it can detect API misuses in Android apps effectively and can help developers correct API misuses in practice.

3 Android API Misuse Problems

3.1 API Misuse

We classify Android APIs into two top categories, including the normal and abnormal ones. In particular, the abnormal category is classified into *@deprecated*, *@hide* and *@removed* APIs, which may cause apps or the Android system to behave abnormally, crash down, leak sensitive information and sometimes introduce security vulnerabilities when misused in programming.

"@deprecated" API Misuses. *@deprecated* APIs might be changed or removed in a future release. The deprecated APIs usually have some irreparable flaws and should be avoided. Although developers can keep using the deprecated methods, they are advised to switch to some new APIs or find some other ways to achieve their goals.

For example, *Sticky Broadcast* is a *Broadcast* that stays around following the moment it is announced to the system, which is declared with *@deprecated* in Android API level 21–23 and advised not to use in programming. Most *Broadcasts* are usually sent, processed within the system and quickly become inaccessible. However, *Sticky Broadcasts* announce information and remain accessible after being processed. They provide no security (anyone can access them) or protection (anyone can modify them), and induce many other security problems.

"@hide" API Misuses. *@hide* APIs are not part of the published APIs in Android SDK, usually intended for debugging and presenting some inner informations and should never be used for core logic in apps. As of the later versions, *@hide* APIs may be no longer available to third party apps, or they may leak privacy information of the caller. Although developers can still access the hidden methods via java reflection, they are advised not to use them and to find some other ways to achieve their goals.

For example, *getRecentTasks()* is a *@hide* API before Android 5.1.1, and it returns the tasks that the user has recently launched with the most recent one being the first and older ones after in order. The restriction of *getRecentTasks()* could be bypassed by the *getRunningAppProcesses* function and it would leak the name of the foreground application protected by permissions with a "dangerous" protection level. Therefore, this security vulnerability was declared as CVE-2015-3833 and rated as a moderate severity vulnerability by National Vulnerability Database (NVD) [20].

"@removed" API Misuses. *@removed* APIs have been deleted from the core Android framework API set. Whenever an API is marked as removed, it is no longer available for the current Android SDK version. If an app tries to call a *@removed* method of a class (either static or instance), an error of *NoSuchMethodError* will be thrown to the caller and the app may be crashed.

For example, *getTextColor(Context, TypedArray, int)* is a function in package *android.widget* and class *TextView*, and has been removed in Level 21. It was removed because *android.R.styleable* is neither public nor stable. As a result, passing a *TypedArray* into any framework *View* is unsafe because the array indices will not match up with the internally used *android.R.styleable* arrays, which will break an app in ways that are difficult to debug.

3.2 Challenges

For Android app developers, the rapid evolution of Android APIs lets them face to the fast changing technologies of application developing. Unfortunately,

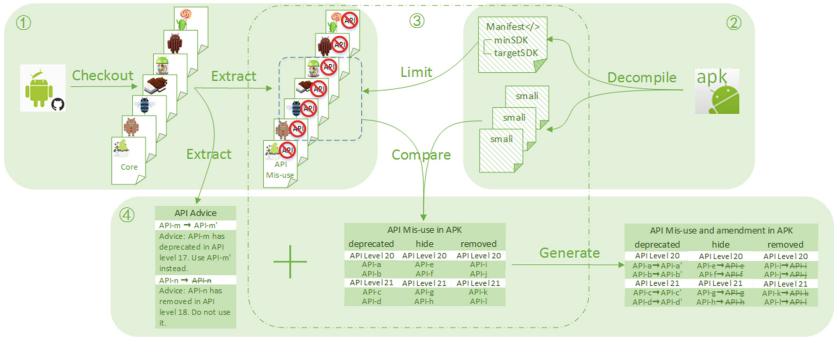

Fig. 1. Overview of MAD-API.

not all the developers can keep pace with the API evolution, while using APIs without knowing their current states. In the above subsection, we introduce three abnormal APIs includes *@deprecated*, *@hide* and *@removed* declarations. In such cases, the developers that lack a comprehensive tracking and understanding of the APIs may introduce API misuses in Android application programming. API misuses will cause apps or the Android system to behave abnormally, crash down, leak sensitive information and sometimes introduce security vulnerabilities.

To develop high quality Android apps, developers should check their API usage against their apps' code for correctness and clear the misused APIs. Therefore, it is necessary to develop a tool to detect and correct API misuses in apps. To achieve this goal, we have the following challenges.

How to detect API misuses in Android applications? The number and details of Android APIs are different for each version. More specifically, because of the compatibility problem, each API should be detected for each app compatible SDK version. Therefore, a detection tool is needed to hold all the API details and detect the API misuses.

How to correct the API misuses for the vulnerable Android applications? When it is clear that an Android app misuses APIs, a new correct method is required to eliminate the misuses and improve the code quality. Furthermore, this method also requires to be easily understood and explainable for developers.

4 MAD-API

We built MAD-API (Misuse Android API Detector and Corrector), an automated framework that consists of a detection method that identifies API misuses in apps and a recommendation method with understandable explanations to correct the misuses.

MAD-API consists of four components as shown in Fig. 1: ① an extractor that extracts and classifies the APIs for each version of Android SDK; ② a reverse extractor that decompiles Android apps and extracts the used APIs;

③ a detection engine that identifies the API misuses by comparing the SDKs' APIs and the apps' APIs; and ④ a correction engine that recommends the proper APIs with explanations to the developers. Once an Android app is put into MAD-API, the misused API reports are generated and the correct APIs are recommended.

```
/**
* @deprecated Sticky broadcasts should not
* be used. They provide no security (anyone
* can access them), no protection (anyone
* can modify them), and many other problems.
* The recommended pattern is to use a
* non-sticky broadcast to report that
* something has changed, with another
* mechanism for apps to retrieve the
* current value whenever desired.
*/
@Deprecated
public abstract void sendStickyBroadcast(Intent intent);
@Deprecated
 public abstract void removeStickyBroadcast(Intent intent);
```

Listing 2: @deprecated Code Snippet of Android SDK.

4.1 Analysis of Android APIs

The details of each API exist in the source code of Android SDK. For example, a code snippet including two *@deprecated* APIs is shown in Listing 2. The APIs that have been analyzed in Subsect. 3.1 belong to the class of *android.content.Context*, which is the interface to obtain the global information about an application environment. From the comments, it can be seen that the APIs related to *Sticky broadcasts* should not be used in programming because they provide no security and may cause problems. The recommendation method is also provided, i.e., using a non-sticky broadcast to report the changes.

Algorithm 1. Android API Extraction Algorithm

Data: $sdk \in \{Android\ versions\}$ $type \in \{@deprecated, @hide, @removed\}$ S is the set of the results
Result: Extracting APIs with *type* for SDK versions
for *each sdk* **do**
  for *each Java file* **do**
    parse *function* from the *file*
    for *each function* **do**
      if *function is declared with type* **then**
        S.add(*function*, *class*, *type*, *sdk*, *comment*)

Therefore, in the first step, MAD-API extracts the APIs for all the SDK versions and labeled the abnormal APIs with *@deprecated*, *@hide* and *@removed* marks. We design an algorithm to extract APIs iteratively as shown in Algorithm 1. For each Android SDK version, it first finds out all the *java* files. Then, it parses the *java* files and splices the files into *function* pieces including the *function* comments. Finally, it determines the type of the *function* and labels the *function* with *@deprecated*, *@hide* and *@removed* marks.

After the above steps, an abnormal API set with items of

$$api(sdk) = (function, class, type, sdk, comment) \tag{1}$$

is obtained. For Listing 2, after extraction, the set is

$$\begin{aligned} S(api\_sdk) =& \{item \mid item\ is\ shown\ as\ Eq.\ 5\} \\ =& \{(sendStickyBroadcast, android.content. \\ & Context, @deprecated, level\_21, Sticky \\ & broadcasts\ should\ not\ be\ used...), \\ & (removeStickyBroadcast, android.content. \\ & Context, @deprecated, level\_21, Sticky \\ & broadcasts\ should\ not\ be\ used...), ...\}. \end{aligned} \tag{2}$$

When all the *functions* are traversed, the three types of *@deprecated*, *@hide* and *@removed* functions are labeled and added to the set.

4.2 Analysis of Application APIs

An Android app is generally written in Java and compiled to ".class" files. Then, it is converted into ".dex" file that contains the byte-code and the associated data and can be run in Android runtime environment (Dalvik virtual machine or ART). Finally, it is packaged into a ".apk" file with an "AndroidManifest.xml" file that describes the content of the package and other resource files.

To extract the used APIs in an app, a reverse toolchain including "apktool", "dex2jar" and "jd-core" is used, which reverses the ".apk" file into ".dex" file and the readable ".smali" or "Java" files. For example, a code snippet of calling Android API is shown in Listing 3. Line 3 shows the called API is *startUsingNetworkFeature*, which is a function of the class *android.net.ConnectivityManager*. Line 7 shows an inner function of Java SDK, which will not be extracted in this step. At the same time, the API level information, e.g., target, minimum and maximum versions of SDK, are also obtained. Finally, a used API set with items of

$$api(app) = (function, class, sdk, app) \tag{3}$$

is obtained.

```
.line 693
invoke-virtual {v5, v7, p0},
Landroid/net/ConnectivityManager;->
  ↪  startUsingNetworkFeature(ILjava/lang/String;)I
move-result v2

.line 694
invoke-static {v7}, Ljava/lang/Integer;-> valueOf(I)Ljava/lang/Integer;
```

Listing 3: @deprecated Code Snippet of Android SDK.

For Listing 3, after extraction, the used API set is

$$\begin{aligned} S(api\_app) = \{&(startUsingNetworkFeature, \\ &android.net.ConnectivityManager, \\ &level\_14\_23, com.sin.web), ...\}, \end{aligned} \tag{4}$$

where *level_14_23* means the "minSdkVersion" and "targetSdkVersion" are 14 and 23 respectively. Whenever all the ".smali" files are traversed, an API usage set is obtained for the app.

Table 1. Correction and explanation of API misuses.

Misused API	Type	SDK version	Correction
sendStickyBroadcast()	@deprecated	level_21	sendBroadcast()

The explanation is that Sticky broadcasts should not be used. They provide no security (anyone can access them), no protection (anyone can modify them), and many other problems. The recommended pattern is to use a non-sticky broadcast to report that something has changed, with another mechanism for apps to retrieve the current value whenever desired.

4.3 Detection Engine of API Misuses

In the third step, MAD-API detects the misused APIs for apps from the Android API set, and the algorithm is shown in Algorithm 2. For each API used by an Android app, it is compared with each item of Android API. If an app API is in the set of Android API, and if the API is declared with one of *@deprecated*, *@hide* and *@removed* marks, the API is added to the misused API set. When all the APIs are processed, the misused API set with items of

$$api(mis) = (function, class, sdk, app, comment) \tag{5}$$

for this app is obtained.

Algorithm 2. Android API Detection Algorithm

Data: $S(api_{app})$ *is application API set* $S(api_{sdk})$ *is SDK API set* $S(api_{mis})$ *is misused API set type* $\in \{@deprecated, @hide, @removed\}$
Result: Detecting Misused APIs for Applications
for *each* $S(api_{app})$ **do**
  for *each* $S(api_{sdk})$ **do**
    if $api_{app} \in S(api_{sdk})$ **then**
      if *function is declared with type* **then**
        $S(api_{mis}).add(api_{app})$

Taking Listing 3 for example, after detection, the misused API set is

$$\begin{aligned} S(api_{mis}) = \{&(startUsingNetworkFeature,\\ &android.net.ConnectivityManager,\\ &@deprecated, level\_14\_23,\\ &com.sin.web, startUsingNetworkFeature\\ &should\ not\ be\ used...), ...\}, \end{aligned} \tag{6}$$

where API *startUsingNetworkFeature* is used by app *com.sin.web* and labeled with *@deprecated* in Android API. Therefore, *startUsingNetworkFeature* and the other detected APIs are added to the misused API set.

For quantitatively analyzing the detection results of API misuse, we define two metrics: misuse rate and misuse frequency as follows.

Definition 1 (Misuse Rate). *Misuse Rate denotes the rate of the misused APIs from the used APIs, which means how many APIs may not be understood by developers.*

$$\mathbb{R}(api) = \frac{|set(api(mis))|}{|set(api(app))|}. \tag{7}$$

Definition 2 (Misuse Frequency). *Misuse Frequency denotes the rate of the total number of misused APIs from the total number of used Android APIs, which means how often an API is misused in programming.*

$$\mathbb{F}(api) = \frac{\sum api(mis)}{\sum api(app)}. \tag{8}$$

4.4 Correction Engine of API Misuses

In practice, a detection system should not only indicate the abnormal misused APIs, but also provide explanations and corrections for the detection results. It is a common shortcoming of existing detection approaches to provide corrections. For MAD-API, we extend our detection by adding a correction engine, such that it can provide the reasons for API misuses and further recommend correct APIs.

Moreover, an explanation may also help developer understand the details of the APIs and the recommendation can also help them correct the API misuses.

MAD-API parses explanations from the API source code comments, which are programmer readable annotations and added with the purpose of making the source code easier to understand. As shown in Table 1, API *startUsingNetworkFeature()* detected from app *com.sin.web* (whose comparable SDK version is level 14 to 23) is *@deprecated* at Level 21, meaning it is misused and may cause problems. The recommended API is *requestNetwork()*, and the explanation obtained from the comments is *"Deprecated in favor of the cleaner requestNetwork(NetworkRequest, NetworkCallback) API. In M, and above, this method is unsupported and will throw UnsupportedOperationException if called."*. The explanation shows that *startUsingNetworkFeature()* should not be used in programming and provides the correct API with justification. By referring to the results, developers can know the correct API and how to use it.

5 Evaluation

5.1 Datasets

The first step in our evaluation is to construct the API sets of Android SDKs and the apps to be evaluated. In particular, we clone the official Android source tree that is located in a Git repository hosted by Google, and checkout source code from SDK Level 8 to 23. We also collected an initial dataset of 10,000 unique apps from markets. In total, the collected apps take 90.82 GB disk size.

5.2 Android API Extraction

With the datasets, it is obvious that the API of Android system is evolving continuously, including API updates, deletion, addition and changes as shown in Table 2. For the Android APIs, from SDK Level 8 to 23, the numbers are from 18,273 to 26,790, which is therefore a 46.9% increase. The significant increase denotes that Android has provided more new features during the last 8 years. From Table 2, it can also be seen that the types of API have also been labeled. Taking *@deprecated* APIs for example, from SDK Level 8 to 23, the numbers are from 253 to 622, which is a 145.8% increase. The increase implies that if these APIs are used without understanding the details, the apps may be vulnerable. The results of *@removed* and *@hide* APIs hold the same conclusion.

Therefore, from the evaluation of Android API, the following finding is obtained.

Finding 1 (Android API Evolution). *Android API is continuously evolving and the quantities of @deprecated, @hide and @removed APIs increase at the same time.*

Table 2. Statistics of APIs in each Android version.

SDK	Level	APIs	deprecated	removed	hide
Android-2.2_r1	8	18273	253	0	551
Android-2.3_r1	9	19373	258	0	651
Android-2.3.3_r1	10	19466	259	0	672
Android-3.0.x	11	—	—	—	—
Android-3.1.x	12	—	—	—	—
Android-3.2.4_r1	13	21295	319	0	879
Android-4.0.1_r1	14	21991	336	0	1152
Android-4.0.3_r1	15	22022	335	0	1161
Android-4.1.1_r1	16	22675	468	0	1335
Android-4.2.1_r1	17	23062	463	0	1498
Android-4.3_r1	18	23534	393	0	1662
Android-4.4_r1	19	24254	409	0	1903
Android-4.4w	20	—	—	—	—
Android-5.0.0_r1	21	26670	511	10	2598
Android-5.1.0_r1	22	26790	527	10	2670
Android-6.0.0_r1	23	26790	622	32	3164

5.3 Application API Extraction

Now, we apply MAD-API to the 10,000 Android apps in our dataset, and each app is reversed into *"smali"* files and then the used APIs are extracted. For example, the No.1 app, whose package name is *com.squarebit.onemanga* uses 33,888 Android APIs. On the other hand, there are also some apps that do not use any Android API. From the Android app API extraction evaluation, we have the following finding.

Finding 2 (Application API Extraction). *Developers use or repeatedly use many Android APIs in Android app programming, e.g., more than 10,000 APIs have been used in each of the top 50 apps.*

5.4 API Misuse Detection

MAD-API detects API misuses by using the SDK and app API sets obtained in the above subsections. Figure 2 presents the number of the *@deprecated*, *@hide* and *@removed* APIs detected from apps, where the apps sorted in terms of the sizes on the x-axis (the largest app on the top left). It is obvious that apps usually have the most *@deprecated* misused APIs, then the *@hide* APIs and at last the *@removed* APIs. Therefore, from Fig. 2, we have the following findings.

Table 3. Part of the correction and explanation of API misuses.

Misused API	Type	SDK version	Correction
restartPackage()	@deprecated	level_8	killBackgroundProcesses()
createSocket()	@removed	level_21	HttpURLConnection()
complexToDimensionNoisy()	@hide	level_23	Do not use

The explanations are as follows.

restartPackage() is now just a wrapper for killBackgroundProcesses(String); the previous behavior here is no longer available to applications because it allows them to break other applications by removing their alarms, stopping their services, etc.

createSocket(). API level 23 removes support for the Apache HTTP client. If your app is using this client and targets Android 2.3 (API level 9) or higher, use the HttpURLConnection ?class instead. This API is more efficient because it reduces network use through transparent compression and response caching, and minimizes power consumption.

complexToDimensionNoisy() was accidentally exposed in API level 1 for debugging purposes. Kept for compatibility just in case although the debugging code has been removed.

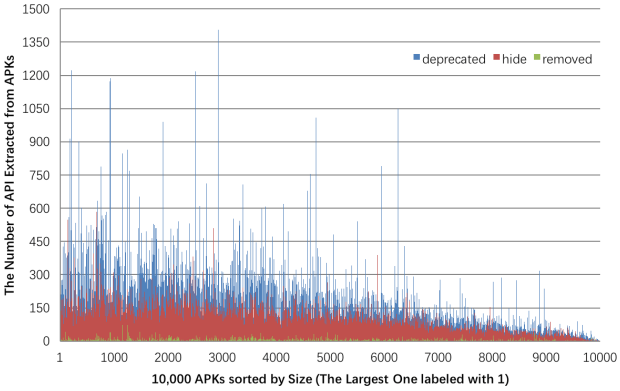

Fig. 2. API misuses detected from 10,000 applications.

Finding 3 (Detection Results). *API misuse problem is extremely serious: 9,313 out of the 10,000, i.e., 93.13%, apps suffer from it.*

Finding 4 (Misuse Factors). *Larger size apps have more API misuses, which may be caused by the participation of more developers and the complicated features.*

To quantitatively analyze the detection results of API misuse, Fig. 3 presents the two metrics that are misuse rate $\mathbb{R}(api)$ and misuse frequency $\mathbb{F}(api)$. From

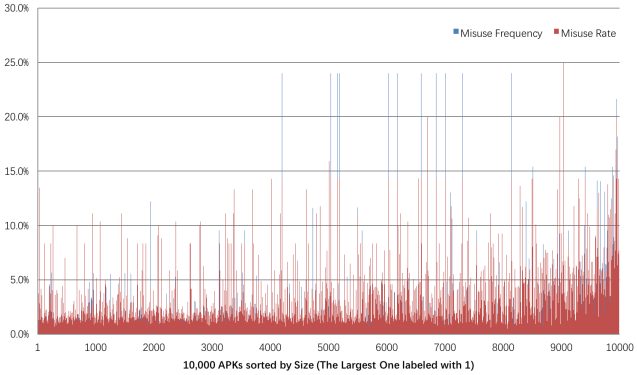

Fig. 3. The misuse rate and misuse frequency in 10,000 applications.

Fig. 3, we can see that $\mathbb{R}(api)$ is between 0%–25%, and $\mathbb{F}(api)$ is between 0%–24%. Subsequently, the following two findings show more understandable metrics.

Finding 5 (Average Misuse Rate). *Average Misuse Rate, denoted by* $\mathbb{AVG}(\mathbb{R})$ *denotes the average rate of the misused APIs of the 10,000 apps. Then, we have*

$$\mathbb{R}(api) = \frac{\sum_{i=1}^{10,000} |set(api(mis))|}{\sum_{i=1}^{10,000} |set(api(app))|} = 59\%, \tag{9}$$

which means 59% Android APIs are misused in the current app developing ecosystem.

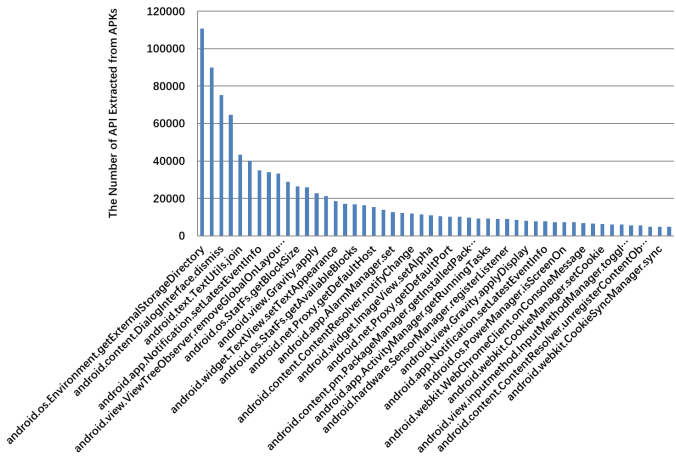

Fig. 4. Top 50 misused APIs in 10,000 applications.

Finding 6 (Average Misuse Frequency). *Average Misuse Frequency, denoted by* $\mathbb{AVG}(\mathbb{F})$*, denotes the average frequency of the misused APIs from the used Android APIs. Here, we have*

$$\mathbb{F}(api) = \frac{\sum_{i=1}^{10,000} \sum api(mis)}{\sum_{i=1}^{10,000} \sum api(app)} = 81\%, \tag{10}$$

which means 81% used APIs are misused in the current app ecosystem.

To understand which are the most easily misused APIs, we sort the detected misused APIs by their occurrences and Fig. 4 presents the top 50 misused APIs in the top 10,000 apps. For example, *android.os.Environment.getExternalStorage Directory* has been misused for 110,789 times. Therefore, we can find that not all the Android app developers can keep pace with the API evolution and they even misuse some APIs all the time.

5.5 API Misuse Correction

Another main feature of MAD-API is that it not only detects the misused APIs, but also provides explanations and corrections for its detection results. Typical correction results of the 10,000 apps are shown in Table 3, where the *Correction* and *Explanaiton* contents are derived from the Android documentation and the source code comments. If the correct API can be derived directly, it will be shown in the table. For example, *killBackgroundProcesses()* is used to replace the *@deprecated* API *restartPackage()*, and the reason is also provided. For another example, *complexToDimensionNoisy* should not be used for it was exposed in API level_1 accidentally. If the correct API cannot be derived, MAD-API will provide explanations to the developers. Therefore, we got the following finding.

Finding 7 (Correction and Explanation). *With the correction and explanation, MAD-API actually helps developers correct the API misuses, understand the API usages and improve the code quality of apps.*

6 Related Work

A large body of research has studied the API misuse detection problem [2,5,10,14–16,19,24,25,27]. In this section, we review some related researches and compare our work with them.

Android API Analysis. APIs and their documentations include the functionality and structure information that is needed by software developers. To automatically augment API documentations, [19] presents an approach that mining the "Stack Overflow" sentences that are related to a particular API type and complements the API documentations in terms of concepts, purpose, usage scenarios, and code examples. For Android APIs, [5] investigates how the fault- and change-proneness of APIs used by Android apps relates to their success, and

finds that apps having high user ratings use less fault- and change-prone APIs than the low rated ones. [12] quantifies the co-evolution behavior of Android and mobile apps and confirmed that client adoption is not keeping pace with API evolution.

We present MAD-API, extract APIs from 13 Android versions and 10,000 apps, and find that the developers falling behind the API evolution may misuse the APIs and lead problems. The top 10,000 Android apps have been analyzed and we show the problem of API misuse is extremely serious, where 93.13% apps suffer from the problem, and the total number of API misuses is 1,241,831.

Application Analysis. Android apps analysis is a hot topic nowadays [6,7,9,22,23,26,28,29]. More and more researchers use static analysis and dynamic behavior analysis, and even integrate them with machine learning techniques to identify malwares. [8] presents MassVet, an innovative malware detection technique that compares a submitted app with all other apps on a market. They analyzed nearly 1.2 million apps and discovered 127,429 malicious apps. [11] also evaluates more than 1 million apps with ANDRUBIS, combining static analysis with dynamic analysis. [2] presents Drebin that detects Android malware directly on smartphones, and provides explanations of the detection to users. [3] detects apps for abnormal usage by analyzing the sensitive data and related APIs.

Similar to the prior studies, we analyze apps while mainly focus on API misuse detection and correction. We found 93.13% on the top 10,000 apps suffer from the API misuse problem in the current application developing ecosystem, and we also recommend corrections with understandable explanations.

7 Conclusion

Android APIs is evolving continuously. However, not all the Android app developers can keep pace with the evolution after their apps have been distributed, leading API misuses issue in the distributed apps. Unfortunately, API misuses may cause many problems, e.g., abnormal behavior, crash down or sensitive information leakage, to apps or the Android system. To help app developers check their API usage against their apps' code for correctness, we propose an automated framework MAD-API and evaluate it use the top 10,000 Android apps. The results show that: (1) the problem of API misuse is serious; and (2) the proposed MAD-API is effective in detecting and correcting such misused APIs.

Acknowledgments. This work was partly supported by NSFC No. 61772507, No. 2017YFB0801902 and 2017YFB1002301.

References

1. Android: Welcome to the android open source project! http://source.android.com
2. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of android malware in your pocket. In: NDSS 2014 (2014)

3. Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., Bodden, E.: Mining apps for abnormal usage of sensitive data. In: ICSE 2015, pp. 426–436 (2015)
4. Bae, S., Cho, H., Lim, I., Ryu, S.: SAFEWAPI: web API misuse detector for web applications. In: FSE 2014, pp. 507–517 (2014)
5. Bavota, G., Linares-Vásquez, M., Bernal-Cárdenas, C.E., Penta, M.D., Oliveto, R., Poshyvanyk, D.: The impact of API change- and fault-proneness on the user ratings of android apps. TSE **41**(4), 384–407 (2015)
6. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the app is that? Deception and countermeasures in the android user interface. In: SP 2015, pp. 931–948 (2015)
7. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A., Shastry, B.: Towards taming privilege-escalation attacks on android. In: NDSS 2012 (2012)
8. Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., Zou, W., Liu, P.: Finding unknown malice in 10 seconds: mass vetting for new threats at the google-play scale. In: SEC 2015, pp. 659–674 (2015)
9. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: SEC 2011, p. 21 (2011)
10. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., Poshyvanyk, D.: API change and fault proneness: a threat to the success of android apps. In: ESEC/FSE 2013, pp. 477–487 (2013)
11. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Veen, V.v.d., Platzer, C.: ANDRUBIS - 1,000,000 apps later: a view on current android malware behaviors. In: BADGERS 2014, pp. 3–17 (2014)
12. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the android ecosystem. In: ICSM 2013, pp. 70–79 (2013)
13. Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A.: How can I use this method? In: ICSE 2015, pp. 880–890 (2015)
14. Nguyen, T.T., Pham, H.V., Vu, P.M., Nguyen, T.T.: Learning API usages from bytecode: a statistical approach. In: ICSE 2016, pp. 416–427 (2016)
15. Petrosyan, G., Robillard, M.P., De Mori, R.: Discovering information explaining API types using text classification. In: ICSE 2015, pp. 869–879 (2015)
16. Ponzanelli, L., Bavota, G., Mocci, A., Di Penta, M., Oliveto, R., Hasan, M., Russo, B., Haiduc, S., Lanza, M.: Too long; didn't watch!: Extracting relevant fragments from software development video tutorials. In: ICSE 2016, pp. 261–272 (2016)
17. Robbes, R., Lungu, M., Röthlisberger, D.: How do developers react to API deprecation?: The case of a smalltalk ecosystem. In: FSE 2012, pp. 1–11 (2012)
18. Slavin, R., Wang, X., Hosseini, M.B., Hester, J., Krishnan, R., Bhatia, J., Breaux, T.D., Niu, J.: Toward a framework for detecting privacy policy violations in android application code. In: ICSE 2016, pp. 25–36 (2016)
19. Treude, C., Robillard, M.P.: Augmenting API documentation with insights from stack overflow. In: ICSE 2016, pp. 392–403 (2016)
20. Common Vulnerabilities and Exposures: CVE-2015-3833 (2015). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3833
21. Wu, J., Liu, S., Ji, S., Yang, M., Luo, T., Wu, Y., Wang, Y.: Exception beyond exception: crashing android system by trapping in "uncaught exception". In: ICSE 2017, pp. 283–292 (2017)
22. Wu, J., Wu, Y., Yang, M., Wu, Z., Luo, T., Wang, Y.: POSTER: biTheft: stealing your secrets by bidirectional covert channel communication with zero-permission android application. In: CCS 2015, pp. 1690–1692 (2015)

23. Wu, J., Yang, M.: LaChouTi: kernel vulnerability responding framework for the fragmented android devices. In: ESEC/FSE 2017, pp. 920–925 (2017)
24. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: exposing missing checks in source code for vulnerability discovery. In: CCS 2013, pp. 499–510 (2013)
25. Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C.: From word embeddings to document similarities for improved information retrieval in software engineering. In: ICSE 2016, pp. 404–415 (2016)
26. Zhang, H., She, D., Qian, Z.: Android root and its providers: a double-edged sword. In: CCS 2015, pp. 1093–1104 (2015)
27. Zhang, M., Duan, Y., Feng, Q., Yin, H.: Towards automatic generation of security-centric descriptions for android apps. In: CCS 2015, pp. 518–529 (2015)
28. Zhang, N., Yuan, K., Naveed, M., Zhou, X., Wang, X.: Leave me alone: app-level protection against runtime information gathering on android. In: SP 2015, pp. 915–930 (2015)
29. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS 2012 (2012)