Accessing Inaccessible Android APIs: An Empirical Study

Li Li, Tegawendé F. Bissyandé, Yves Le Traon, Jacques Klein
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
{li.li, tegawende.bissyande, yves.letraon, jacques.klein}uni.lu

***Abstract*—As Android becomes a de-facto choice of development platform for mobile apps, developers extensively leverage its accompanying Software Development Kit to quickly build their apps. This SDK comes with a set of APIs which developers may find limited in comparison to what system apps can do or what framework developers are preparing to harness capabilities of new generation devices. Thus, developers may attempt to explore in advance the normally "inaccessible" APIs for building unique API-based functionality in their app.**

The Android programming model is unique in its kind. Inaccessible APIs, which however are used by developers, constitute yet another specificity of Android development, and is worth investigating to understand what they are, how they evolve over time, and who uses them. To that end, in this work, we empirically investigate 17 important releases of the Android framework source code base, and we find that inaccessible APIs are commonly implemented in the Android framework, which are further neither forward nor backward compatible. Moreover, a small set of inaccessible APIs can eventually become publicly accessible, while most of them are removed during the evolution, resulting in risks for such apps that have leveraged inaccessible APIs. Finally, we show that inaccessible APIs are indeed accessed by third-party apps, and the official Google Play store has tolerated the proliferation of apps leveraging inaccessible API methods.

I. Introduction

Any piece of successful software will inevitably be maintained [1]. This law of software evolution applies well today to the Android ecosystem where the Software Development Kit (SDK) provided by Google for app developers is ever evolving to optimize key Application Programming Interfaces (APIs) and take into account new hardware capabilities. Adaptations of the SDK are even key in the strategy for attracting developers by regularly exposing functionalities for harnessing new hardware capabilities and satisfying end-user requirements for fancy functionalities. This strategy is paying on Android where developers have now contributed with over 2 millions mobile apps [2] in the Google Play official market alone, the largest software distribution platform ever built.

In a development environment, which integrates an Android SDK, provides the library *android.jar* which exposes a set of APIs for exploiting the capabilities of the Android operating system. Once developed, apps are packaged and installed on devices, the referred APIs now point to implementations in the *framework.jar* library. Although both libraries are built from the same source code, the runtime library (i.e., *framework.jar*) is much richer than the development library (i.e., *android.jar*) as it includes a set of APIs which are *inaccessible* for the development of third party apps as illustrated in Figure 1.

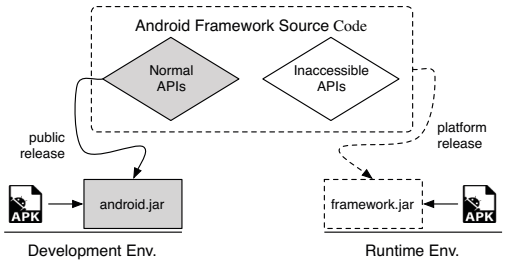


Fig. 1: The release mechanism of Android SDKs.

Inaccessible APIs are however widespread in the source code base of Android and come in two forms: 1) *Internal* APIs located in the package `com.android.internal` are reserved to system apps, whose development is more rigorous, giving them access to sensitive resources, e.g., low level access to hardware. 2) *Hidden* APIs are scattered across the source code as a collection of classes marked with the javadoc `@hide` annotation. These are often deemed not stable enough to be promoted or are still subject to invasive changes in future releases. It is no longer a secret in the Android development community that inaccessible APIs are key to getting a competitive edge for creating some unique API-based functionality in mobile apps [3]. For example, in early adoption of Android apps, game box manufacturers have attempted to build apps that allowed user devices to be used as joysticks via Bluetooth connection. Developers thus had to experiment with hidden APIs for automated pairing as we will discuss in Section II.

To access *inaccessible APIs*, developers can build a custom development SDK library where internal and hidden APIs are included. This solution however presents a risk that the framework library shipped with user devices mismatches developer SDK library, leading to runtime application crashes. A second option is to rely on reflection techniques to invoke the available APIs at runtime. This option allows to take into account the cases where the API is unavailable. Unfortunately, the use of reflection challenges static analyses of app code [4].

Because APIs are the main building blocks of applications that interface with core systems such as the Linux kernel, the Android framework, or the Eclipse IDE, their design and implementation as well as their evolution may have a substantial impact on the ecosystem. Previous studies have indeed shown

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.35

for example that building Linux APIs with implicit usage preconditions may lead to safety holes [5], [6]. Stability of Java APIs has also been found to be problematic [7]. Finally, the usage of unstable, discouraged and unsupported APIs of the Eclipse platform have been extensively investigated in recent work [8]–[10]. In this work, we focus on the Android ecosystem who has witnessed recently a rapid growth in terms of developers and applications. We study inaccessible APIs to quantify the risks that they pose in terms of the impact of their evolution, the sensitivity of the features that they offer access to, and how prevalent is the use of such APIs in market apps. This study is a first step towards understanding the necessity and impact of inaccessible APIs in the Android ecosystem. The findings of this study may also raise concerns to the development community as well as to Android framework management team.

Android inaccessible APIs are similar to the notion of *private APIs* [11] in the iOS world with the difference that apps using private APIs in iOS will be banned from the Apple store during app review, whereas Android apps using inaccessible APIs are tolerated on the Google Play store. This difference motivates the need to study Android inaccessible APIs where there is a mismatch in which APIs are available at development time and which can actually be invoked at runtime. In this paper, we propose to empirically investigate such APIs, and propose research questions around the significance of the presence of inaccessible APIs in the SDK, their impact on Android programming w.r.t. the resources they allow access to (e.g., sensitivity) and the stability of their implementation, as well as their adoption by the development community. Eventually, we summarize, based on the empirical findings, the exploratory implications we could learn from the evolution of inaccessible APIs.

Significance. *What is the proportion of inaccessible APIs in the Android framework?:*

- RQ1 – What is the evolution trend over time of inaccessible APIs?
- RQ2 – Will inaccessible Android APIs be removed in their subsequent releases? If yes, how often?
- RQ3 – For how long are inaccesible APIs kept in the framework?

Impact. *Are inaccessible APIs reliable for production development?:*

- RQ4 – What sensitive resources are accessed by inaccessible APIs?
- RQ5 – How stable are the implementation of inaccessible APIs?
- RQ6 – Do hidden APIs ever become openly available for use in the SDK?

Adoption. *How prevalent is the use of inaccessible APIs in the community?:*

- RQ7 – To what extent are inaccessible APIs being leveraged by app developers?
- RQ8 – Is the use of inaccessible APIs similar across malicious and benign apps?
- RQ9 – Is there any framework that ease the access of inaccessible APIs?
- RQ10 – Is there any correlation between the publication of a hidden API and the number of apps using it?

II. Background

Android apps are written in the Java programming language on traditional desktop computing platforms. To compile their code for execution by the Android runtime system, developers are provided with a development kit that includes tools that can be used in command line or via a GUI integrated with an Integrated Development Environment (Eclipse ADT, or Android Studio built on IntelliJ IDEA by JetBrains).

The Android SDK includes a device emulator, a debugger, libraries exposing functionalities of the Android platform as well as various code examples and documentation. Maintenance of the Android SDK goes hand in hand with the overall Android platform development. Thus, the SDK library classes evolve to support most recent devices but may also include older versions of the Android platform for testing apps on older devices.

A. Android API levels

Android uses API levels to manage app compatibility across different versions of the runtime system. Indeed, as the Android platform evolves to include new features and address new hardware requirements, new Android versions are released with each version being assigned a unique integer identifier, called *API level*. In practice, each release of Android is referred to multiple names: (1) its version number (e.g., Android 4.4); (2) its API level (e.g., API level 19); and (3) a name of sweet (e.g., KitKat). Figure 2 overviews the adoption of API levels by millions of Android-powered devices using Google Play official store as of March 2016 [12].

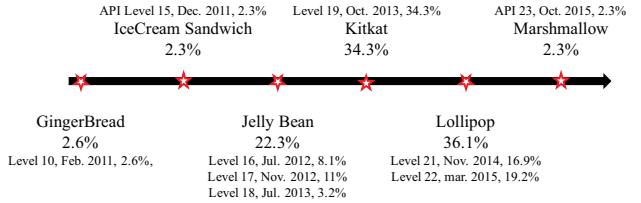


Fig. 2: Distributions of API levels supported by Android-powered devices (Versions with less than 1% are not shown).

It is noteworthy that, approximately every 6 months, the API level is upgraded, suggesting substantial updates in the API set provided to Android developers. In any case, at a given time, each Android device supports exactly one API level which represents the version of the libraries that developer apps can invoke at runtime. The API level is even used to determine whether an application is compatible with an Android framework version prior to installing the application on the device. Thus, when developers build their apps, they can specify in the Manifest file included in the Android package (apk) the *target* API level that the app is built to run on as

well as the *minimum* API that is required to run the app and the *maximum* API level on which the app can run. These values help the SDK tools warn developers on the use of APIs. The flexibility of supporting a range of API levels is further put in place to ensure a larger compatibility of apps with devices in the wild. Unfortunately, when developers use inaccessible APIs from a given API level, they bet on the availability of the necessary libraries on user device. This device however may be running with a different API level that falls within the supported range but where maintenance efforts have extensively changed/dropped the accessed APIs. Let us assume that a given app uses hidden API *getUserId(int)* of class *android.content.pm.PackageManager* from Android SDK with API level 14. The developer has fixed to the app the minimum, target and maximum API levels to 13,15,16 respectively. Once uploaded on the market, the app may be installed on devices running a platform version of Android at various API levels. Unfortunately, the hidden API is available at runtime only on devices with API level ranging from 14 to 15. As a consequence, the app cannot be executed on devices with API level 13 and 16.

B. Motivating examples

Third party apps developed based on the Android SDK are normally restricted in the set of APIs that they can use to interact with Android runtime libraries. Listing 1 showcases three methods which are available in the source code of the Android SDK, but which cannot be used in the normal settings of an Android development environment.

The first method, *isRoamingBetweenOperators()*, is a method located within a package whose code is reserved to system apps. This method is used to activate roaming on GSM networks. On purpose, we illustrate an inaccessible API with a Java `private` qualifier to further show how this restriction can be bypassed via reflection and used as an API. This internal API was introduced since API level 4.

The second and third methods, *createBond()* and *setTrust()*, which are annotated with `@hide`, are not available within the compiled development library of the SDK. These Bluetooth APIs allows to directly pair the Android device with a remote device. Both of them were introduced in API level 5 and later removed after API level 19 and 21, respectively. Actually, unlike *setTrust()*, which was removed from the source code, *createBond()* was disclosed to public (i.e., `@hide` annotation is removed), making it directly accessible for third-party apps.

We now discuss the case of two real-world Android apps distributed via the official Google Play store which use the above illustrated APIs. The first app, *Gravity Xbox - Tweek box* (*com.ceco.gm2.gravitybox*), is an advertizing app that can inject code into any app, including system services. In the excerpt of Listing 2, the app reflectively finds and calls the private internal API method discussed above. After submitting this app to VirusTotal, we see that 8 anti-virus engines flag it as malicious. This app has also been removed from the app store [13].

```
//Internal API
package package com.android.internal.telephony.gsm;
final class GsmServiceStateTracker extends
     ServiceStateTracker {
 /**Set roaming state when the first parameter is true*/
 private boolean isRoamingBetweenOperators(boolean,
     ServiceState) {}
}
//Hidden APIs
package android.bluetooth;
public final class BluetoothDevice implements
     Parcelable {
  /**Start the bonding process with the remote device.
   * @hide*/
 public boolean createBond() { }
 /**Set trust state for a remote device.
  * @hide*/
 public boolean setTrust(boolean value) { }
}
```

Listing 1: Examples of internal and hidden APIs.

```
r2 = findClass("com.android.internal.telephony.gsm.
     GsmServiceStateTracker", null);
r3 = newarray (java.lang.Object)[3];
r3[0] = Class.TYPE;
r3[1] = "android.telephony.ServiceState";
r3[2] = new ModPhone$5(...);
findAndHookMethod(r2, "isRoamingBetweenOperators", r3);
```

Listing 2: Code excerpt from *Gravity Xbox - Tweek box* illustrating a reflective access to an internal API at runtime.

The second app, *Phonejoy* (*com.phonejoy.store*), is a benign app designed to work with Phonejoy Bluetooth Game Controllers. To allow a transparent connection – without the need to request system-level pairing – with an interfaceless remote gaming box, the app directly makes use of the Bluetooth *createBond()* and *setTrust()* hidden APIs discussed. Listing 3 shows the code excerpt where the hidden API will be reflectively accessed at runtime.

```
public class com.hellostore.bluetooth.BluetoothDriver {
 public boolean createBond(BluetoothDevice r1) {
  r4 = r1.getClass();
  r5 = r4.getMethod("createBond", null);
  r6 = r5.invoke(r1, null); }
 public boolean setTrust(BluetoothDevice r1) {
  r4 = r1.getClass();
  r5 = newarray (java.lang.Class)[1];
  r5[0] = Class.TYPE;
  r6 = r4.getMethod("setTrust", r5);
  r7 = newarray (java.lang.Object)[1];
  r7[0] = Boolean.valueof(1);
  r9 = r6.invoke(r1, r7); }}
```

Listing 3: Illustrative example of accessing a hidden API. This snippet is extracted from app "com.phonejoy.store" (*1F6DA3*).

Unfortunately, the reflection mechanism as they are used in the examples above are risky. Indeed, after app distribution on the store, there is no guarantee on the presence of the inaccessible APIs in the framework runtime libraries. In the case of the benign Phonejoy app, this app version remains only available to older devices supporting lower API levels where the hidden APIs was still available.

III. Study Setup

To answer the research questions motivating this work we must collect data on the historical evolution of Android APIs, infer the feature categories (such as *telephony*, representing

telephone-related features) they belong to as well as the permissions that govern them, and check their adoption rate on a representative set of apps.

A. APIs data collection

Framework code: We refer to the Android framework project, which as of March 2016, includes 219 revision tags [14]. Because several revision releases can be made for a same API level when the changes (e.g., critical bug fixes) do not significantly change the API set, we consider selecting a unique release for every API level. Practically, we retain the latest revision in a given API level. Table I provides details on the release considered per API version. The source code base of Android include the history of releases starting from Android 1.6 (Donut, API level 4). API level 20 is specific to wearables, and is thus not considered in this study.

TABLE I: List of Android versions considered for investigation. For each API level, we select the latest revision.

API Level	Release Counts	Selected Revision	API Level	Release Counts	Selected Revision
23	31	android-6.0.1_r9	13	1	android-3.2.4_r1
22	35	android-5.1.1_r9	10	8	android-2.3.7_r1
21	11	android-5.0.2_r3	9	2	android-2.3.2_r1
19	24	android-4.4w_r1	8	10	android-2.2_r1.3
18	11	android-4.3_r3.1	7	5	android-2.1_r2.1s
17	7	android-4.2_r1	6	1	android-2.0.1_r1
16	11	android-4.1.2_r2.1	5	1	android-2.0_r1
15	7	android-4.0.4_r2.1	4	7	android-1.6_r2
14	4	android-4.0.2_r1			

Overall, our study considers 17 versions where the number of API methods ranges from 51,607 to 223,786 between API level 4 and API level 23. Fig. 3 further shows the evolution in quantity of inaccessible and accessible APIs. We note a significant jump in the number of accessible APIs after API level 19. While API level 20 is only specific to wearables, API level 21 brings substantial changes such as the change from Dalvik to ART as runtime system, the support for 64-bit CPUs, etc.

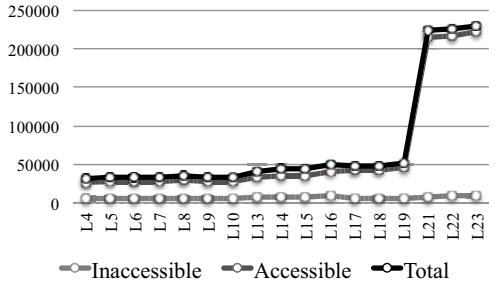


Fig. 3: Trend of API methods exposed in the Android framework.

App code: We leverage the Androzoo repository [15]. We consider apps where the target API level has been released in the last 3 years (i.e., above API Level 4). At the time of experiments, the dataset collected included 23,666 apps (90% from the official Google Play market, and 10% from alternative markets such as Anzhi, appChina, etc.). Figure 4 shows how the dataset spans over several years of app development.

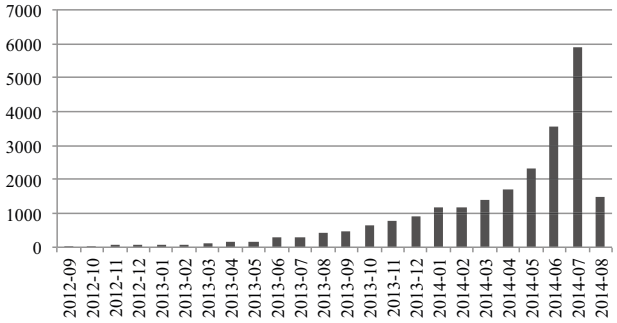


Fig. 4: Distribution of the packaging date of our selected apps.

B. Metadata collection

Aside source code data on framework and apps, we also collect metadata for further characterizing our dataset. First, we use the PSCout [16] dataset to retrieve which permissions govern the APIs of the Android framework. Unfortunately, PSCout only provides API-to-permission mappings for 10 release tags, while two of them are contributed to a same API level (*4.41* and *4.4.4* for level 19), for which we only consider the latest one for this study. Because of this, in this work, we take 9 versions of API-to-permission mappings into consideration, where each mapping represents a different API level.

Second, for each app considered in this study, we also send them to VirusTotal [17] to check whether it is malicious or not. VirusTotal is a free service that hosts over 50 anti-virus products. In this work, we take a given app as malicious as long as one of those anti-virus products hosted on VirusTotal reports it as such.

Third, to harvest the inaccessible APIs that are leveraged by Android apps, we present in this paper a simple but fast approach, which first statically extracts all the constant strings from a given Android app and then we match them with all the inaccessible APIs. If there is a perfect match (both class and method name), we select it as a candidate. For further details on how reflective calls, which are used to access inacessible APIs, are identified, we refer the reader to our recent work on taming reflection in Android apps with DroidRA [18].

Finally, we consider a straightforward approach to infer the features concerned by the APIs. Indeed, we rely on the naming mechanism of Java packages and the implicit requirement for them to be "meaningful" to derive the keywords representing features. To that end, we preprocess the package names to split them into a set of keywords from which we drop the common terms such as *com*, *org*, *android*, etc., and directly consider the remaining keywords as a description of a feature (e.g., *telephony* represent a given feature with its set of classes).

IV. Empirical Findings

We now report on the findings of our empirical study with regards to the research questions outline in Section I.

A. Significance of the phenomenon of Inaccesible APIs

Considering the source code base for the different API levels under study, we investigate the evolution in the number of Java methods that will be normally inaccessible once the SDK development library is compiled. Figure 5 shows the overall trend where hidden methods are found more and more in the Android framework. Internal methods have also been increasing until a significant decrease between API level 16 and API level 17. We investigate the reason for this decrease and found that API level 17 came after a cleanup of *telephony*-related internal methods, with a drop from 4,137 methods to 47 methods.

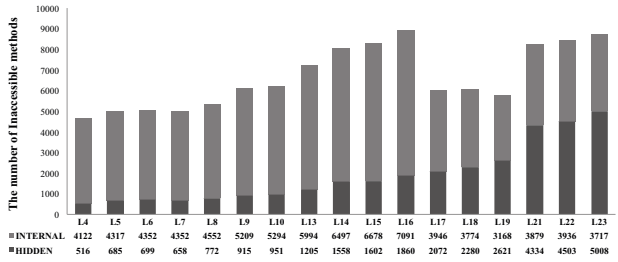


Fig. 5: Trend of the number of inaccessible (internal + hidden) methods in Android framework source code base.

In Figure 5, the data accounts for all methods, i.e., those with Java qualifiers `public` and `private`, since all can be reflectively accessed by third party apps, as illustrated in Section II-B. Nevertheless, we show in Figure 6 that the ratio of methods qualified as `public` is very high, and relatively stable, among internal and hidden methods. In the remainder of this paper, we refer to any hidden or internal method as an API method which is inaccessible.

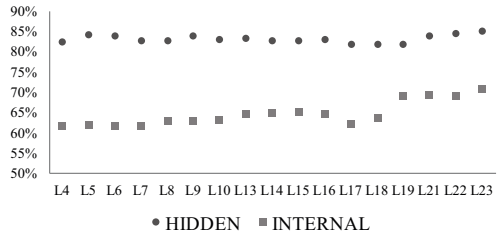


Fig. 6: Ratios of inaccessible methods with Java `public` qualifier.

RQ1: Inaccessible APIs are continuously implemented in the Android framework code base.

We further investigate whether the addition of new inaccessible APIs is actually accompanied by a renewal of existing ones. To that end we conduct two studies exploring the removal rate of inaccessible APIs and the "life expectancy" of an inaccessible API. For the first study, we perform pairwise comparisons between two consecutive API level releases of the Android framework. Table II indicates that, at every new release, some API methods are removed from the framework code base. The rate of inaccessible API removals range between 0.09% and 62.91% for internal API methods and between 0 and 15.56% for hidden API methods.

TABLE II: The number of removed inaccessible APIs for each update.

Update	Internal	Hidden	Total	Update	Internal	Hidden	Total
L4 → L5	179	36	215	L14 → L15	37	17	54
L5 → L6	4	0	4	L15 → L16	228	188	416
L6 → L7	6	54	60	L16 → L17	4461	215	4676
L7 → L8	165	80	245	L17 → L18	386	126	512
L8 → L9	354	8	362	L18 → L19	998	89	1087
L9 → L10	35	10	45	L19 → L21	353	245	598
L10 → L13	599	148	747	L21 → L22	66	144	210
L13 → L14	306	80	386	L22 → L23	960	346	1306

RQ2: There is no guarantee of forward compatibility when using inaccessible APIs

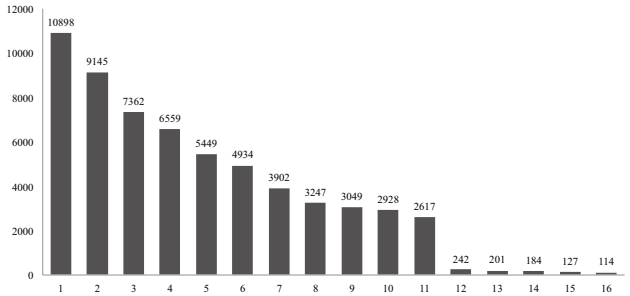


Fig. 7: Life expectancy of inaccessible APIs. Age corresponds to the number of API level generations before an API method is removed from the code base.

To compute the life expectancy of inaccessible APIs for the second study, we check their ages when they are removed, i.e., the number of releases they survive on before being dropped from the framework code base. We observe from the results in Fig. 7 that most inaccessible API methods are removed early after being introduced. Over 56% of inaccessible API methods are removed on the subsequent update. In comparison only 9% of public accessible API methods are removed after one update. If we consider the methods that are still remaining in the framework since the beginning of Android, after 16 updates, there are only 2% (i.e., 114) of inaccessible methods, whereas 44% of public accessible methods have stayed. According to these numbers, it is clear that the public accessible APIs are more stable than inaccessible ones. However, over 50% of public accessible APIs are removed has also surprised us, as we would expect that most of them need to stay in the framework, in order to keep apps being backward-compatible.

We further look into this problem and find that Google has introduced the so-called "support libraries" to solve the backward-compatible problem [19]. More specifically, when a feature F is no longer supported in the framework base code, its implementing code is simply removed from the code base. However, for some reasons, developers may still want to support F for old Android versions. In this case, they have to integrate a supporting library (e.g., a frequently used one is

android.support.v4) into their apps' code, which contains the implementing code of F that has previously been removed from the framework base. Since the supporting library code will be released along with the app code, the old-fashioned features will work fine even if the running framework does not support them anymore.

RQ3: The turn-over of inaccessible APIs is very high, most of them being removed from the framework after a few version updates.

B. Potential Impact of the use of Inaccessible APIs

We now investigate which features of Android system are concerned with the prevalence of inaccessible APIs. Table III summarizes top different features where the implementation of inaccessible APIs are manipulated (i.e., either added, removed or updated). We note that *telephony*-related inaccessible API methods are the most updated over time.

TABLE III: Added, updated, and removed features of each SDK update.

Update	added	updated	removed
L4 → L5	webkit	telephony, cdma	gsm
L5 → L6	SlidingTab, DigitalClock	widget	view, telephony
L6 → L7	SignalStrength, widget, view	-	webkit, telephony, WebSettings
L7 → L8	util, content	telephony	widget, vcard
L8 → L9	policy, telephony, sip	-	awt, location, AndroidGraphics2D
L9 → L10	GsmSmsCbTest	telephony, nfc	policy
L10 → L13	view, widget	telephony	gsm, stk
L13 → L14	widget	telephony, view	policy
L14 → L15	gsm, UsimDataDownloadCommands	telephony	policy, ArrayListCursor
L15 → L16	widget, view	telephony	webkit, net
L16 → L17	policy, keyguard, keyguard_obsolete	-	telephony, gsm, cdma
L17 → L18	util	policy, view	keyguard_obsolete
L18 → L19	os	policy	keyguard, PagedView
L19 → L21	renderscript	widget	ActionBarImpl
L21 → L22	widget	telephony, TelephonyManager	SubscriptionManager
L22 → L23	os	policy, widget	PhoneWindowManager

Another means for characterizing the implications and impact of the use of inaccessible APIs is to investigate the permissions that govern them. Figure 8 shows how inaccessible APIs are increasingly protected by permissions. In recent versions of the Android framework, we see that around 25% of inaccessible API methods are guarded by system permissions while it is the case for only less than 10% of openly accessible API methods.

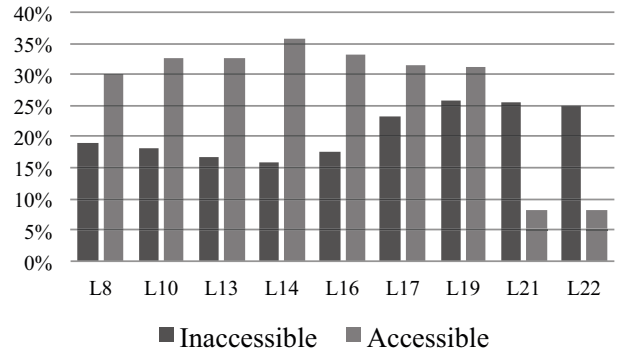


Fig. 8: Ratios of inaccessible and accessible API methods whose access is protected by Android system permissions.

We further look at the top permissions requested for inaccessible API methods in each API level. Findings in Table IV are noteworthy since some of the top permissions, namely `DUMP` and `MODIFY_PHONE_STATE`, are also marked in Android documentation as "Not for use by third-party applications" [20].

TABLE IV: Permissions required for using inaccessible APIs.

API Level	PSCout Version	Total Permissions	Distinct Permissions	Top 2 Permissions
L8	mapping_2.2.3	1,410	29	WAKE_LOCK, MODIFY_PHONE_STATE
L10	mapping_2.3.6	1,696	35	WAKE_LOCK, BROADCAST_STICKY
L13	mapping_3.2.2	2,249	35	WAKE_LOCK, BROADCAST_STICKY
L14	mapping_4.0.1	2,605	38	WAKE_LOCK, MODIFY_PHONE_STATE
L16	mapping_4.1.1	1,128	16	DUMP, READ_PHONE_STATE
L17	mapping_4.2.2	864	8	DUMP, INTERNET
L19	mapping_4.4.4	760	7	DUMP, INTERACT_ACROSS_USERS
L21	mapping_5.0.2	948	7	DUMP, INTERACT_ACROSS_USERS
L22	mapping_5.1.1	938	7	DUMP, INTERACT_ACROSS_USERS

Finally, we check whether the set of permissions requested for inaccessible API methods is requested in the same proportion for accessible API methods. In this experiment, we only concern the API methods implemented in the Android release for API level 16 (i.e., *android-4.1.1*), which is the release with the highest number of inaccessible methods. In this release, there are 71 permissions in total. All of these permissions have been required by accessible APIs, while only 16 (23%) of them are required by inaccessible APIs. This huge difference suggests that there are only a specific set of features of resources that are interested by inaccessible APIs (i.e., for system apps).

RQ4: Inaccessible APIs only access a specific set of features of resources, comparing to accessible APIs.

To study the overall stability of inaccessible API implementations we leverage the ChangeDistiller [21] tool on the source code of the framework. We record changes at the method level between releases of API levels. Figure 9 provides the change rates between consecutive releases of methods. We differentiate hidden methods and internal methods from openly accessible methods. The histograms highlight the fact that accessible API methods are significantly less subject to invasive changes than hidden and internal API methods.

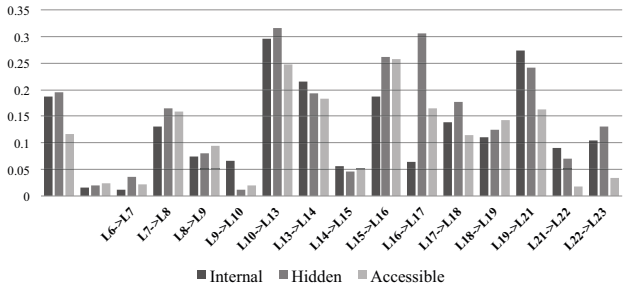


Fig. 9: Change rate comparison between inaccessible and accessible APIs. Hidden and Internal are the higher in 8 and 5 cases respectively, while Accessible is only in 3 cases.

Since ChangeDistiller is a semantic differencer, its output lists semantic change actions applied from one version to another. Most changes of inaccessible API methods are

about adding (*statement_insert*), removing (*statement_delete*), modifying (*statement_update*) and moving statements (*statement_parent_change*). Figure 10 enumerates top 20 change types and the number of times they have occurred during inaccessible API method evolutions.

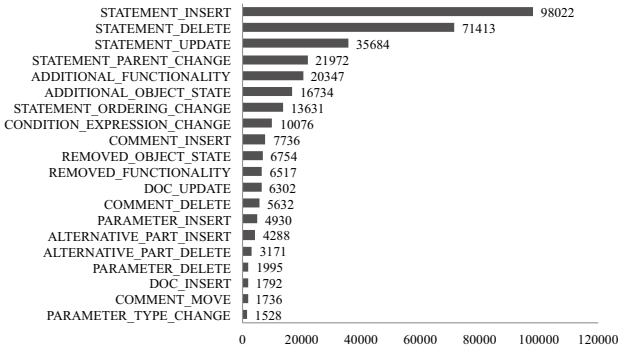


Fig. 10: Top 20 change types of inaccessible methods.

RQ5: Overall inaccessible APIs, internal as well as hidden, are more unstable than accessible APIs.

Finally we consider the evolution of inaccessible API methods to compare their probability of being removed against their probability of becoming accessible. We focus on hidden API methods as they are the most susceptible of later being disclosed when they become stable. Figure 11 illustrates the total number of inaccessible methods removed from the code base against the number of those that have been disclosed for normal access. From one API level to the next, hidden API methods are more removed than made fully accessible.

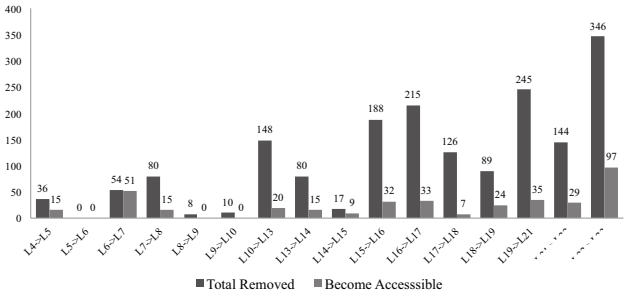


Fig. 11: Total removed hidden APIs VS. Disclosed hidden APIs.

RQ6: It's more likely for an inaccessible API method to be removed at the release of a new API level, than to be made officially accessible in the compiled SDK development library.

C. Adoption of inaccessible APIs by third-party app developers

We have found that 5.4% (i.e., 1269) third-party apps of our dataset exploit inaccessible API methods in their code. 965 (i.e., 76%) of these apps were even distributed in the official Google Play market. We note the readers that our selected dataset is initially composed with 90% of Google Play apps. The 14% (i.e., 90%-76%) decreasing indicates that apps from the alternative markets (appChina, Anzhi, etc.) are more favorable to access inaccessible APIs, comparing to such ones that are distributed through Google Play store.

Figure 12 provides a boxplot of the number of inaccessible methods used per app for the 1,269 apps. On average, each app uses 3.13 inaccessible methods with a median value of 2.

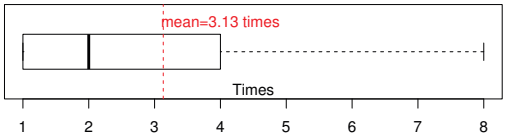


Fig. 12: Number of inaccessible methods used in apps that exploit such API methods.

RQ7: The official Google Play store maintainers during app reviews has somehow allowed the proliferation of apps leveraging inaccessible API methods in their code.

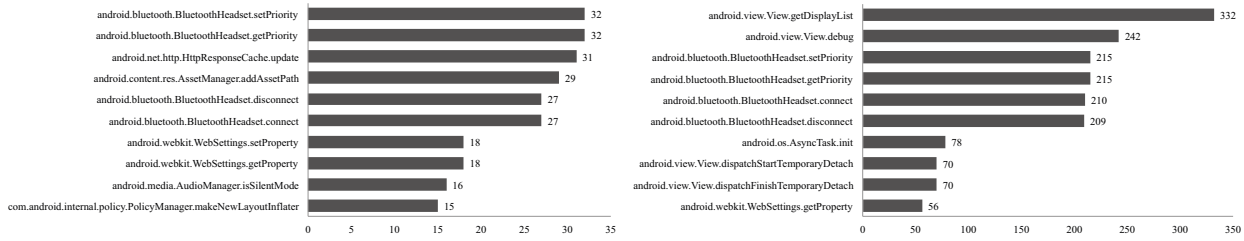
We further investigate the likelihood of a malicious app to use inaccessible API in comparison with benign apps. We found that 9% of the malicious[1] apps in our dataset actually use inaccessible methods, while this is the case for only 5% of benign apps. Figure 13 summarizes the top used API methods by malicious apps and benign apps. We note that, aside from the Bluetooth Headset priority settings API methods, top methods used in malicious apps discriminate from top methods used in benign apps. Benign apps mostly use inaccessible methods related to *view* functionalities.

RQ8: Inaccessible APIs with specific permission requirements leveraged by malicious apps are not always the same as those used by benign apps.

During our investigation on inaccessible APIs, we have found that some apps are actually leveraging a framework called *Xposed* [22] to ease the implementation of accessing inaccessible APIs. The Xposed Framework is proposed to change the behavior of the system (e.g., through system-level inaccessible APIs) without installing a new custom ROM, which is the only way to achieve that kind of functionality previously. In order to support customized features for third-party apps, Xposed framework implements a module system, where each module corresponds to a feature that developers can choose. As an example, in Listing 2, the internal method *isRoamingBetweenOperators()* is actually accessed by a module of the Xposed framework.

In our experiments, we have found 168 apps that leverage Xposed framework for inaccessible APIs. Among the 168

[1]We consider an app to be malicious when a single antivirus engine from VirusTotal has flagged it as such.



(a) Malicious. (b) Benign.

Fig. 13: The top 10 used inaccessible APIs methods by our dataset apps.

apps, 13 of them are malicious while the majority remaining 155 apps are benign, suggesting that Xposed framework is not dedicated to malicious apps, and tweaking the system (or app) behaviors are also a choice for legitimated apps.

RQ9: Both malicious and benign apps are willing to use generic framework to ease the implementation of accessing inaccessible APIs.

We now consider the last research question to investigate whether developers happen to use inaccessible APIs that are likely to be openly accessible later. Fig. 14 represents two boxplots: one related to the number of inaccessible API methods invoked by third-party apps which then becomes accessible; another one related to the number of inaccessible API methods invoked by third-party apps which remains inaccessible. In the first case, the median value is 1, whereas in the second case, the median value is 3.

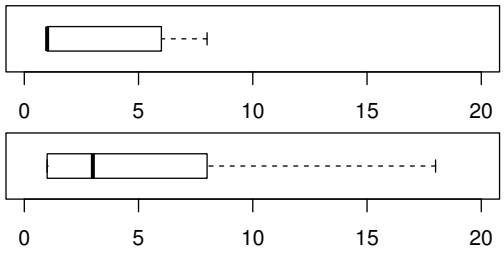


Fig. 14: Boxplot on the number of apps using inaccessible APIs that are disclosed to be accessible (the above one) and are remained to be inaccessible (the bottom one).

RQ10: Third-party app developers do not appear to take into consideration whether an inaccessible API will become accessible in future releases before using it. Instead, they are simply interested in harnessing immediately the potential of some sensitive APIs.

V. What all this means

We now discuss the exploratory implication of the evolution of inaccessible APIs that our community could observe based on this study. Section V-A focuses on the app development and distribution model aspects, while Section V-B focuses on the developer behaviour aspect.

A. App development and distribution model

a) the open source model of Android development creates the opportunity for third-party app developers to see how the code base evolves, and what features are available for system apps. This in turn incites developers to search for ways to leverage all APIs, whether allowed or not. Indeed, as our study has shown, inaccessible APIs are widely used by third party apps, although they are only published in the code base (not in the API reference list). On the other hand, since Google open-sourced the implementation of Android around 2009, there are opportunities for any third-party developer to contribute to the framework code base with new features and bug fixes that will maintain it as a leading development platform for mobile devices

b) Java and Dalvik bytecode format makes it easy for developers to explore SDK library code compiled in Java classes as well as system apps. There are even a number of well know tools such as Soot [23] and dex2jar which allow the translation of Android app bytecode into Java.

c) Java reflection, as already pointed out in several works, remains challenging for static analysis [24], [25]. All data in this study on the adoption of inacessible APIs by market apps are extracted from the detection of reflective calls using constant string analysis. Thus, in order to facilitate the analysis of inaccessible APIs, there is a need to tackle reflection for Android apps. To the best of our knowledge, however, reflection has not yet been fully investigated in static analysis approches. Recently, Barros et al. [26] proposed a strategy within their Checker framework [27] to extract reflection-related values for Android apps. We have also recently proposed an approach based on modeling reflection as a constant propagation problem [18]. This approach however still presents limitations. Therefore, researchers in the community should be encouraged to put more effort on solving reflection problems and thus to benefit the systematic analysis of inaccessible APIs.

d) App review is not strict, even on the official market. For instance, we found a large number of apps using inaccessible APIs which are governed by permissions that are clearly described as “not for use by third-party apps”. As a comparison, Apple has released a vetting system to keep

apps from using "private" APIs that access to sensitive user information. Considering the rigour by iOS app reviewers to ban apps violating the rules, and the performance in terms of malicious app removal, we advocate for more strict reviews on Android markets to limit the use of inaccessible APIs. We believe this is one of the first essential steps towards building a safer Android ecosystem.

B. Developer behaviours

a) Use of inaccessible API is for short-term benefits. In the realm of mobile development, the multitude of apps offering the same services pushes app developers to take risks and seek the immediate rewards of building fancy functionality based on unstable, unreliable, and short-lived APIs (cf. RQ.10).

b) The usage of Inaccessible APIs can be learned from others. Since there is no "official" documentation on how to use inaccessible APIs, we are interested in how developers obtain the skills of accessing inaccessible APIs. To this end, we take each inaccessible API as a feature, which result in 341 features for every app. Then, we cluster them through the Expectation-maximization (EM) algorithm [28], in an attempt to investigate if there are correlations in terms of the usage of inaccessible APIs among Android apps. As a result, our dataset are categorized into 9 clusters, where each of them shows a set of apps that share similar usage of inaccessible APIs. As an example, the biggest cluster contains 944 apps, and all of them have used the same four inaccessible APIs, which are *connect()*, *setPriority()*, *disconnect()*, and *getPriority()* of class *android.bluetooth.BluetoothHeadset*. This clustering suggests that an experience in how to connect a bluetooth headset to Android devices programmatically has been systematically reused in hundreds of apps, without any documentation being available for this task in the Android community.

VI. Threats to validity

First, our empirical findings are dependent on the validity of the meta data that we collected as introduced in Section III-B. For example, our permission-related findings are relying on PSCout's results. Although PSCout's results have been widely used, we have found that there are some strange behaviors among their evolution. For example, as what we show in Table IV, the results before and after *L16* are quite different, suggesting a potential bias due to tool malfunctioning errors.

Second, our findings are based on a selected subset of releases of the Android framework source code base. The selection introduces threats into the external validity, making our results potentially not representative for the whole evolution of Android framework base. To mitigate this threat, we have taken into account nearly all the API levels of releases.

Third, our study on apps that use inaccessible APIs is based on the constant strings that we can harvest from the apps, which may cause false positives (the found class/method name may not really be used in practice) and false negatives (i.e., our results may not represent the full list of accessed inaccessible APIs). Regarding false positives, we have manually sampled 20 apps and found that all of them have really accessed inaccessible APIs either through reflection or Xposed framework. Regarding false negatives, to the best of our knowledge, we believe our results can be improved by a thorough analysis on the usage of reflective calls in Android apps [18], [29].

Forth, our app-based investigation is based on all the code, including common libraries. As shown in our previous work [30], the fact that common libraries are pervasive in Android apps may impact our findings. Filtering out such libraries could improve our results, we thus leave this part as future work.

Finally, since our empirical study is conducted purely on software artifacts (either framework base or apps), our findings are supported by analyzing those artifacts alone and thus may not reflect the opinions of developers. To mitigate this, in our future work, we plan to contact developers for a more comprehensive understanding on their motivation of using inaccessible APIs.

VII. Related Work

To the best of our knowledge, we are the first to investigate the evolution of inaccessible Android APIs. However, there are several works that have studied the general Android APIs. In this section, we summarize them through permission, evolution, and documentation directions. In the end of this section, we also discuss some related works tackling inaccessible APIs in the iOS system.

API Permission. As an example, research works [16], [31], [32] have investigated the mappings between Android APIs and their permissions. As an example, Bartel et al. [31] statically analyzing the framework source code base to extract the mapping, in which they have introduced an advanced class-hierarchy and field-sensitive set of analyses. Instead of analyzing the framework code statically, Felt et al. [32] presents a tool called Stowaway that leverages an automated testing approach to dynamically extract the API to permission mappings. Unlike the previously approaches, Au et al. [16] present a tool called PSCout, which builds the mapping through analyzing the Android permission specifications. All of these approaches have built a mapping from APIs to permissions and shown that it was quite common for Android apps to be granted more permissions than they actually needed, which are known as "permission gap".

API Evolution. API evolution is a frequently researched topic in the software maintenance field [33], [34]. McDonnell et al. [35] have performed an empirical study on API stability and adoption in Android, in which they show that Android is evolving fast at a rate of 115 API updates per month on average, while the average time taken to adopt new versions is much longer comparing to fast evolving APIs. Linares et al. [36], [37] investigated the relationships between the success of Android apps and the SDK API changes. They empirically found that more successful Android apps generally use APIs that are less change-prone. In another direction, Linares et al. have also shown that SDK API changes will trigger more stack overflow discussions [38]. Our findings on inaccessible

APIs are overall in line with their findings, as the change of inaccessible APIs may cause serious crashes, which will consequently impact the app's success and also prompt more discussions on social medias.

API evolution has also been studied for many other platforms. For example, Jezek et al. [7] investigated the API changes and their impacts for Java programs. They have found that API instability is common and will eventually cause problems. As shown by Mastrangelo et al., although Java is a relatively safe language, it does also support unsafe features to go around its safety guarantees [39]. Hora et al. have also studied how developers react to API evolution for the Pharo system. Their findings further confirm that API evolution can have a large impact on a software ecosystem in terms of client systems, methods, and developers. Finally, Businge et al. have performed several empirical investigations [8]–[10] on the usage of unstable, discouraged and unsupported APIs of the Eclipse platform, which shows that ill-designed APIs are quite common in big systems.

API Documentation. Wang et al. [40] investigated the obstacles of using Android APIs through analyzing API-related posts regarding Android development from *Stack Overflow*, a Q&A web site. Parnin et al. [41] also based on *Stack Overflow* to investigate how these kind of Q&A web sites facilitate crowd documentation. Their experimental results show that the crowd is capable to generate a rich source of content on the usage of APIs with code examples and discussion that are actively viewed and used by many developers. Although *Stack Overflow* has been shown useful for boosting the uses of APIs, Subramanian et al. [42], however, have shown that it could be further improved by linking its results to code example-based sources such as *Github*, and thus to enhance traditional API documentation with up-to-date source code examples. Linares et al. [43] also performed an empirical study on the usage of Android APIs. However, instead of focusing on usage obstacles, they investigated the usage pattern of energy-greedy APIs, attempting to answer the question whether the anomalous energy consumption is unavoidable or is due to sub-optimal usage or choice of APIs. Li et al. [44] empirically investigated the parameter values of Android APIs. Their experimental results have shown that actually parameter values, harvested from a large scale of apps, could be leveraged to recommend practice usage of APIs.

Inaccessible APIs in iOS system. Like Android, there are also a batch of inaccessible APIs in the iOS system, which are usually referred as *private* APIs. Apple requires every iOS app to go through a vetting process so as to prevent apps from using private APIs that access to sensitive user information. Recent attacks [45], [46] have shown the feasibility of using private APIs without being detected during Apple's app review processes. To this end, Deng et al. [47] introduce an additional vetting system called iRiS to conquer such attacks, which leverages a fast static analysis approach to resolve common API calls and an iterative dynamic analysis approach to further resolve such APIs that cannot be resolved statically.

In this paper, we have found that, there is probably no check for using inaccessible APIs in the Android field. As an example, we have shown previously that some apps accessed "private" permissions, which are clearly marked in the document as "Not for use by third-party applications", can still be published at Google Play store. Therefore, based on the successful experience of Apple's app review process (although there are several exceptions), we believe that the Android markets should also learn from Apple and thus to prevent apps from using inaccessible APIs, as we have already shown, it is quite risky as well for those apps themselves that have accessed inaccessible APIs.

Reflection Resolution. As shown in this paper, inaccessible APIs are mainly accessed through Java reflection, which by itself has been investigated in several works [48], [49]. Most recently, as a complement of this work, we present DroidRA [18], [29] to resolve reflective calls of Android apps through constant string analysis. Regarding Java apps, Bodden et al. [50] have presented TamiFlex for resolving reflections, so as to boost static analyses. More recently, Li et al. introduce a system called SOLAR, which is intended to perform sound reflection analysis [51].

VIII. Conclusion

In this paper, we have performed an empirical study on the evolution of inaccessible APIs of the Android framework base. At first, we investigate the significance of the phenomenon of inaccessible APIs, where we find that inaccessible APIs are continuously implemented in the Android framework, and there is actually no guarantee of forward compatibility when using them. Besides, most inaccessible APIs will be removed in a few version updates.

Second, we investigate the potential impact of the use of inaccessible APIs. We have found that comparing to accessible APIs, inaccessible APIs only access a specific set of features of resources and are more unstable as well as more likely to be removed at the release of a new API level, rather than to be made publicly available in the compile SDK development library.

Finally, we look into the adoption of inaccessible APIs by third-party apps. Experimental results show that there are many apps that are indeed accessing inaccessible APIs and the usage are quite different between malicious and benign apps. We have also found that some apps leverage a framework called Xposed to ease their works of accessing inaccessible APIs. Besides, it seems that developers are not appear to taken into account the risks of removes of inaccessible APIs, but instead, they are interested in harnessing immediately the potential of inaccessible APIs. Last but not the least, unlike Apple store, the Google play store does not have a vetting system for the usage of inaccessible API methods.

Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects AndroMap C13/IS/5921289 and Recommend C15/IS/10449467.

REFERENCES

[1] *Fred Brooks*. Addison-Wesley, 1975.

[2] AppBrain. Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps, 2015. Accessed: 2016-08-01.

[3] Code examples using hidden android apis. http://developer.sonymobile.com/2011/10/28/code-examples-using-hidden-android-apis/. Accessed: 2016-08-01.

[4] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. Technical report, SnT, 2016.

[5] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 60–69. IEEE, 2012.

[6] Tegawendé F. Bissyandé, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. Ahead of time static analysis for automatic generation of debugging interfaces to the linux kernel. *Automated Software Engineering*, 23(1):3–41, 2016.

[7] Kamil Jezek, Jens Dietrich, and Premek Brada. How java apis break–an empirical study. *Information and Software Technology*, 65:129–146, 2015.

[8] John Businge, Alexander Serebrenik, and Mark van den Brand. Survival of eclipse third-party plug-ins. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 368–377. IEEE, 2012.

[9] John Businge, Alexander Serebrenik, and Mark van den Brand. Analyzing the eclipse api usage: Putting the developer in the loop. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 37–46. IEEE, 2013.

[10] John Businge, Alexander Serebrenik, and Mark GJ van den Brand. Eclipse api usage: the good and the bad. *Software Quality Journal*, 23(1):107–141, 2015.

[11] ios apps caught using private apis. https://sourcedna.com/blog/20151018/ios-apps-using-private-apis.html. Accessed: 2016-08-01.

[12] Dashboards android developers. https://developer.android.com/about/dashboards/index.html. Accessed: 2016-08-01.

[13] Best apps market. http://www.bestappsmarket.com. Accessed: 2016-08-01.

[14] Android framework classes and services. https://android.googlesource.com/platform/frameworks/base.git. Accessed: 2016-08-01.

[15] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *MSR*, 2016.

[16] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[17] Virustotal. https://www.virustotal.com. Accessed: 2016-08-01.

[18] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[19] Support library android developers. https://developer.android.com/topic/libraries/support-library/index.html. Accessed: 2016-08-01.

[20] Manifest permission android developers. https://developer.android.com/reference/android/Manifest.permission.html. Accessed: 2016-08-01.

[21] Beat Fluri, Michael Wursch, Martin PInzger, and Harald C Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[22] Xposed module repository. http://repo.xposed.info. Accessed: 2016-08-01.

[23] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[24] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[25] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[26] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Armorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE, Lincoln, Nebraska, 2015.

[27] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 1092–1104, Scottsdale, AZ, USA, November 4–6, 2014.

[28] Tood K Moon. The expectation-maximization algorithm. *Signal processing magazine, IEEE*, 13(6):47–60, 1996.

[29] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration Track*, 2016.

[30] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[31] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *Software Engineering, IEEE Transactions on*, 40(6):617–632, 2014.

[32] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[33] Meiyappan Nagappan and Emad Shihab. Future trends in software engineering research for mobile apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[34] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *RN*, 16:02, 2016.

[35] Tyler McDonnell, Bonnie Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[36] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[37] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, 41(4):384–407, 2015.

[38] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94. ACM, 2014.

[39] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. *ACM SIGPLAN Notices*, 50(10):695–710, 2015.

[40] Wei Wang and Michael W Godfrey. Detecting api usage obstacles: A study of ios and android developer questions. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 61–64. IEEE, 2013.

[41] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep*, 2012.

[42] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[43] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[44] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[45] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. Launching generic attacks on ios with approved third-party applications. In *Applied Cryptography and Network Security*, pages 272–289. Springer, 2013.

[46] Tielei Wang, Kangjie Lu, Long Lu, Simon P Chung, and Wenke Lee. Jekyll on ios: When benign apps become evil. In *Usenix Security*, volume 13, 2013.

[47] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56. ACM, 2015.

[48] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*, pages 378–400. Springer, 2013.

[49] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[50] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[51] Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In *International On Static Analysis*, pages 162–180. Springer, 2015.