Noname manuscript No.
(will be inserted by the editor)

CDA: Characterising Deprecated Android APIs

Li Li · Jun Gao · Tegawendé F. Bissyandé · Lei Ma · Xin Xia · Jacques Klein

the date of receipt and acceptance should be inserted later

Abstract Because of functionality evolution, or security and performance-related changes, some APIs eventually become unnecessary in a software system and thus need to be cleaned to ensure proper maintainability. Those APIs are typically marked first as *deprecated APIs* and, as recommended, follow through a *deprecated-replace-remove* cycle, giving an opportunity to client application developers to smoothly adapt their code in next updates. Such a mechanism is adopted in the Android framework development where thousands of reusable APIs are made available to Android app developers.

In this work, we present a research-based prototype tool called CDA and apply it to different revisions (i.e., releases or tags) of the Android framework code for characterising deprecated APIs. Based on the data mined by CDA, we then perform an empirical study on API deprecation in the Android ecosystem and the associated challenges for maintaining quality apps. In particular, we investigate the prevalence of deprecated APIs, their annotations and documentation, their removal and consequences, their replacement messages, developer reactions to API deprecation, as well as the evolution of the usage of deprecated APIs. Experimental results reveal several findings that further provide promising insights related to deprecated Android APIs. Notably, by mining the source code of the Android framework base, we have identified three bugs related to deprecated APIs. These bugs have been quickly assigned and positively appreciated by the framework maintainers, who claim that these issues will be updated in future releases.

Keywords Android · Deprecated APIs · CDA

Li Li and Xin Xia
Faculty of Information Technology, Monash University, Australia
E-mail: li.li, xin.xia@monash.edu

Jun Gao, Tegawendé F. Bissyandé and Jacques Klein
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
E-mail: jun.gao, tegawende.bissyande, jacques.klein@uni.lu

Lei Ma
School of Computer Science and Technology, Harbin Institute of Technology, China
E-mail: malei@hit.edu.cn

1 Introduction

Android is currently dominating the smartphone market, attracting 85% of global sales to end users worldwide. Among the many potential incentives which drive Android's competitiveness in comparison to other mobile operating systems, we note the rapid and constant evolution of the Android framework: (McDonnell et al., 2013) have reported that developers should expect a new release every three months. This is an indication of the pace at which Android maintainers deal with vulnerability fixes and performance improvements on the one hand, and the introduction of new features on the other hand. While these framework code changes empower app developers to continuously provide high-quality apps, they also bring about compatibility issues. For example, during framework evolution, a class can be renamed or a method's signature may be modified (e.g., addition of an extra parameter), eventually impacting the Application Programming Interfaces (APIs), and eventually breaking the execution of developer apps (Bagherzadeh et al., 2017).

To enable a graceful adaptation of developers to framework changes, API deprecations are implemented following the so-called *deprecate-replace-remove* cycle. In this scheme, APIs that will no longer be maintained in the framework are first flagged as deprecated, through a proper *@deprecated* Java annotation, or by inserting *@deprecated* in the relevant Javadoc message. Subsequently, the code of deprecated APIs are updated with replacement messages which are meant to help developers refactor their apps in order to migrate from deprecated APIs to their replacements (Brito et al., 2018b) or support automated refactorings (Dig et al., 2007) (Perkins, 2005). Finally, after some reasonable time (e.g., several releases of the framework), deprecated APIs are eventually removed from the framework so as to clean the framework and thereby reducing the maintenance burden on the framework code base.

Unfortunately, as unveiled by several studies in the research literature (Robbes et al., 2012) (Hora et al., 2015), the *deprecated-replace-remove* cycle is not always respected, leading to challenges for both framework maintainers and app developers. A number of research works have then investigated to tackle the challenges associated to API deprecation. For example, some researchers have explored the quality of documentation for deprecated APIs (Brito et al., 2016) (Ko et al., 2014). Others have studied developer reactions to deprecated APIs(Espinha et al., 2014)(Sawant et al., 2016). There have been also various works on automatically migrating client code in response to broken APIs(Chow and Notkin, 1996) (Nita and Notkin, 2010)(Henkel and Diwan, 2005)(Xing and Stroulia, 2007). Nevertheless, despite the significant attention given to API deprecation in general, it is noteworthy that the problem has not yet been extensively explored in the Android ecosystem specifically.

Our work aims at understanding and characterising how Android APIs are deprecated in practice and how developers react to the phenomenon. The overall goal of this research is to draw insights that (1) framework maintainers can build on to improve strategies for deprecating APIs, and that (2) can be used to assist app developers in dealing with compatibility issues that can arise after API deprecation.

Towards achieving the goal of this work, we present an empirical study on the deprecation of Android APIs. This study builds on a systematic source code

mining of the Android framework, which is constituted of over 3 million lines of Java code in over 7,000 Java files. The study also involved analysing 10,000 real-world Android apps to explore questions related to the management, in practice, of deprecated APIs by developers.

In this work, we first design and implement a prototype tool called CDA, standing for Characterising Deprecated APIs. Then, we apply CDA to different revisions (i.e., releases or tags) of the Android framework code and compare the obtained results to understand the evolution of deprecated Android APIs. Finally, we explore a set of real-world Android apps attempting to understand the reaction of app developers to deprecated Android APIs. Our experimental investigation eventually finds that (1) Deprecated Android APIs are not always consistently annotated and documented; (2) Deprecated Android APIs are regularly cleaned-up from the framework code base and half of the cleaned APIs are performed in a short period of time, requiring developers to quickly react on deprecated APIs; (3) Around 78% of deprecated Android APIs have been commented with replacement messages, which however are rarely updated during the evolution of Android framework code base; (4) Most deprecated APIs are accessed by app code via popular libraries. (5) During the evolution of Android apps, deprecated APIs are likely retained rather than removed from the app code. (6) For the cases app developers do remove deprecated APIs from the app, they are unlikely replacing the deprecated APIs with their alternatives recommended by the official documentation, at least not directly at the same place (e.g., under the same caller method).

To summarise, we make the following contributions:

- We design and implement a prototype tool called CDA that automatically characterises deprecated APIs by mining the source code of Android framework releases.
- We have identified three bugs related to deprecated APIs by parsing the latest revision of the Android framework code. These bugs have been further submitted to the issue tracker system[1] of the Android Open Source Project (AOSP) and have been quickly assigned and positively appreciated by the framework maintainers, who claim that these issues will be updated in future releases.[2]
- We present a quantitative study on deprecated Android APIs along the evolution of the Android framework base.
- We harvest a comprehensive list of deprecated Android APIs and provide also their latest replacement messages that can be leveraged to guide the practical replacements of deprecated APIs.

We make available online our implementation, along with the scripts to replicate our experiments at

`https://github.com/lilicoding/CDA`

It is worth to mention that although CDA targets the Android framework code base, it is implemented generically and could be easily migrated for the analysis of common Java repositories. Concretely, the Java file parser and the API to replacement mapping should work directly to Java projects.

[1] https://issuetracker.google.com

[2] The issue IDs of the submitted bugs are 69105065, 69104762 and 69098890.

This paper is an extended and improved version of a conference paper (Li et al., 2018b) presented at the 2018 International Conference on Mining Software Repositories (MSR). In this extension, we have improved our prototype tool to take into account all the available Java classes in the Android framework base. Compared to the conference version, where only a selected set of core Java classes are considered, we consider much more classes including third-party classes such as the Apache ones (e.g., *org.apache.http.**), sensor-related code such as the ones used to support *opengl* or *nfc*, internal classes such as *com.android.internal.**, assistant code such as legacy-oriented test cases (e.g., *com.android.multidexlegacytestapp*), etc. Additionally, since the conference version of this paper only focuses on deprecated APIs at the method level, a number of deprecated APIs that are deprecated at the class level are actually missed. Specifically, if a class is deprecated, all its methods should be considered as deprecated even if they are not explicitly flagged as such (e.g., via the *@Deprecated* annotation or the *@deprecated Javadoc* tag). In this extension, we have improved our research tool to also take into account such deprecated APIs that are only flagged at the class level. Because of the improvement of our prototype tool, the experimental results (i.e., some statistics) presented in the conference paper are slightly changed (the empirical observations are almost kept the same). Therefore, we re-conduct all the experiments presented in the conference paper and subsequently update this paper with the newly obtained results, accordingly. When re-conducting the experiments, we have additionally considered two major releases (API levels 27-28) of the Android framework, keep our empirical results update to date.

In addition to the improvement of our prototype tool and the massive refactoring of our experimental results, we also introduce two new research questions aiming to (1) understand the evolution of deprecated APIs in terms of their usage in Android apps as well as (2) harvest practical fixes (conducted by developers of real-world apps) that attempt to replace deprecated APIs with their alternatives. Finally, based on the harvested fixes, we provide to the community an online web service[3] that takes as input a deprecated API and outputs a list of *diff*s showing how the searched API is removed in practice.

The remainder of this paper is organised as follows. Section 2 presents the necessary background information to allow readers to better understand this work. Section 3 presents the experimental setup of this work, including the dataset and the research questions as well as the implementation of our prototype tool CDA. Section 4 details our quantitative studies towards answering the aforementioned research questions. After that, Section 5 discusses the potential implications and the possible threats to the validity of this work. The closely related works are detailed in Section 6, followed by our conclusion to this work in Section 7.

2 Background

In this section, we provide the necessary background information on the concept of Android APIs and deprecated APIs to help readers better understand our process.

[3] The online web service can be accessed via `http://35.224.210.36/DAU/`

2.1 Android APIs

Android APIs, like any other APIs that are defined as publicly accessible methods in the code base, are provided to support developers for building shipping quality apps. Those APIs are usually shipped with Software Development Kits (SDKs) that are frequently updated as the Android system evolves: since the launch of Android in 2008, Android SDKs have been released in over 10 versions providing progressively 28 API levels. The latest Android system version is *9.0* and its API level is 28. This SDK comes with an online portal[4] that tracks all documentation written by Android maintainers to help developers correctly use the provided APIs. Fig. 1 presents the screenshot of an example documentation for API *saveLayer(RectF,Paint,int)*, from which app developers can learn the main functionality of this API as well as the necessary knowledge to correctly invoke it.

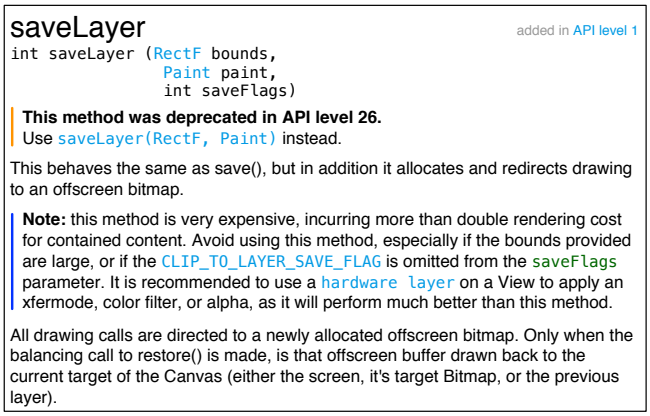

Fig. 1: The documentation and deprecation message of *saveLayer(RectF,Paint,int)*.

2.2 Deprecated APIs

With the evolution of APIs, some of them may no longer fit with the new requirements of the SDK, e.g., because of security or performance reasons (Li et al., 2016b). SDK maintainers thus need to remove such APIs so as to prevent their usage in client apps. Nevertheless, because of potential compatibility requirements, deprecated APIs cannot be directly removed as it may otherwise lead to application runtime crashes. In this context, SDK maintainers adopt a simple convention: any to-be-removed API must first be marked as deprecated API via a Java annotation *@Deprecated*. On the one hand, this annotation indicates that the marked API can be removed in any future release of the SDK and is thus not recommended to be used in a newly developed app. On the other hand, the annotation does not prevent its use in legacy apps, allowing such apps to continue to perform to some extent (e.g., depending on the device and the framework version they are running against).

[4] https://developer.android.com/index.html

Listing 1 illustrates two real examples of deprecated Android APIs, namely *isNetworkTypeValid()* and *removeStickyBroadcast()*, which were implemented in classes *ConnectivityManager* and *Context* of the Android framework base, respectively. The description (cf. lines 3 and 14) explains that these two APIs are deprecated because of function changes (i.e., there is no need to validate the network type) and security concerns (i.e., sticky broadcast provides no security protection).

```
1  //class java.android.net.ConnectivityManager
2  /**
3   * @deprecated All APIs accepting a network type are deprecated. There
        should be no need to validate a network type.
4   */
5  @Deprecated
6  public static boolean isNetworkTypeValid(int networkType)
7  {
8   return MIN_NETWORK_TYPE <= networkType &&
9        networkType <= MAX_NETWORK_TYPE;
10 }
11
12 //class android.content.Context
13 /**
14  * @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.
15  */
16 @Deprecated
17 @RequiresPermission(android.Manifest.permission.BROADCAST_STICKY)
18 public abstract void removeStickyBroadcast(@RequiresPermission Intent
        intent);
```

Listing 1: Examples of deprecated Android APIs.

3 Experimental Setup

Our objective in this work is to mine the Android framework code base for characterising the deprecated Android APIs. We expect this study to provide actionable guidelines for both app developers and market maintainers to better deal with apps accessing deprecated Android APIs. To this end, we present a research tool called CDA to support our analyses on Characterising Deprecated APIs. Before detailing the design and implementation of CDA in Section 3.2, we first present the dataset used in this study (cf. Section 3.1). We conclude the section by presenting some statistical highlights on the Android framework code base (cf. Section 3.3).

3.1 Dataset

Our dataset targets two artefacts, the Android system code base, and client code. Thus, it includes:

- GitHub repository data of the Android framework base.[5]
- A set of 10,000 apps that are randomly selected from AndroZoo (Allix et al., 2016; Li et al., 2017a). We sample 5,000 apps from the official Google Play market (GPlay) apps and 5,000 apps from third-party markets (NGPlay).

[5] https://github.com/android/platform_frameworks_base

The Android platform code, hosted in Github since October 2008,[6] is actually a mirror of the Google source code repository[7] maintained by Google. As of Oct. 2018, it has been forked over 5 000 times, and has seen the contributions of over 700 developers, while being watched for changes by almost 900 developers. The 167 git development branches have integrated changes from 377,474 commits. Each commit representing a revision state of the code base, the successive changes provide a good historical view on how do the APIs evolve. Previous studies have already investigated this evolution in other contexts (Li et al., 2016c) (Coelho et al., 2015) (Palomba et al., 2018).

Over 600 revisions in the framework development are tagged as releases. Consecutive releases can be made available without the API level being changed. We therefore assume that such releases (i.e., within the same API level) will be similar in terms of API structure. In this study, for the sake of simplicity, we pick one release (generally the latest) that is associated to each API level, to build the evolution dataset to be investigated. Note that API levels 11, 12 and 20 are irrelevant to our study as they do not actually correspond to new releases of the code base.[8] Eventually, as illustrated in Table 1, we are able to consider 22 releases (associated to 22 API levels) for our study.

Table 1: Selected Android SDK (or API) Revisions. Because there is no release for API levels 1-3, 11 and 12 and level 20 is reserved for other purposes, in this work, we do not take into account these three API levels.

API Level	Code Name	Selected Release	Date
28	Pie	android-9.0.0_r9	2018-08-30
27	Oreo	android-8.1.0_r48	2018-08-30
26	Oreo	android-8.0.0_r36	2017-08-17
25	Nougat	android-7.1.0_r7	2017-03-30
24	Nougat	android-7.0.0_r7	2016-08-23
23	Marshmallow	android-6.0.1_r9	2015-12-15
22	Lollipop	android-5.1.1_r9	2015-06-10
21	Lollipop	android-5.0.2_r3	2014-12-17
19	KitKat	android-4.4w_r1	2014-05-07
18	Jelly Bean	android-4.3_r3.1	2013-09-05
17	Jelly Bean	android-4.2_r1	2012-11-09
16	Jelly Bean	android-4.1.2_r2.1	2012-09-26
15	Ice Cream Sandwich	android-4.0.4_r2.1	2012-03-20
14	Ice Cream Sandwich	android-4.0.2_r1	2011-12-07
13	Honeycomb	android-3.2.4_r1	2011-09-09
10	Gingerbread	android-2.3.7_r1	2011-09-12
9	Gingerbread	android-2.3.2_r1	2010-12-07
8	Froyo	android-2.2.3_r2.1	2010-11-04
7	Eclair	android-2.1_r2.1s	2010-02-10
6	Eclair	android-2.0.1_r1	2009-11-18
5	Eclair	android-2.0_r1	2009-10-15
4	Donut	android-1.6_r2	2009-11-03

[6] commit: 54b6cfa9a9e5b861a9930af873580d6dc20f773c

[7] `https://android.googlesource.com/platform/frameworks/base.git`

[8] There are no releases (or tags) for API levels 1-3, 11 and 12 while the API level 20 is reserved for wearable devices.

In addition to the Android platform framework base, we also collect Android apps to investigate how deprecated APIs are addressed by app developers. To this end, we inspect 10,000 apps: 5,000 from the official Google Play store (hereinafter referred as GPlay) and 5,000 from third-party markets[9] (hereinafter referred as NGPlay) such as AppChina.[10] These apps are randomly[11] selected from the AndroZoo app repository, which contains over 7 million Android apps and is known to be so far the largest app set publicly available to our community. Apps from this dataset have been previously leveraged for a variety of research studies (Hecht et al., 2015) (Li et al., 2015) (Li et al., 2017b) (Yang et al., 2017). Since GPlay and NGPlay apps may come with different quality and maintenance requirements, they may have different usages of deprecated APIs and may receive different reactions from app developers (Wang et al., 2018). By considering apps from these two sets, we might be able to observe such difference w.r.t. deprecated Android APIs.

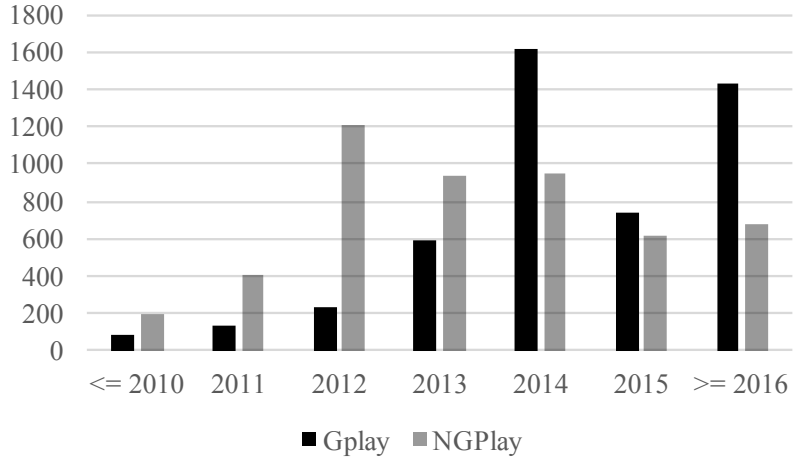

Fig. 2: Distribution of randomly selected apps based on their assembled date (i.e., dex date).

Figure 2 further summarises the distribution of randomly selected apps based on their assembly date, i.e., the time when the core code *classes.dex* was created (i.e., the last modified time). For both GPlay and NGPlay apps, the assembly time ranges from 2010 to 2016, indicated diversity in the apps. Figure 3 further confirms this diversity via the size of selected apps, where both small (less than 1 MB) and big apps (more than 20 MB) are considered. The median and mean size of considered apps are 4.7 MB and 9.1 MB, respectively.

[9] We hypothesise that these apps may be handled differently w.r.t. deprecated APIs compared to GPlay ones.

[10] The full list of involved third-party markets includes AppChina, Anzhi, MI.com, 1Mobile, Angeeks, Slideme, F-Droid, Praguard, Torrents, Freewarelovers, Proandroid, Hiapk, Genome, APK_Bang.

[11] By using *gshuf — head -5000* command.

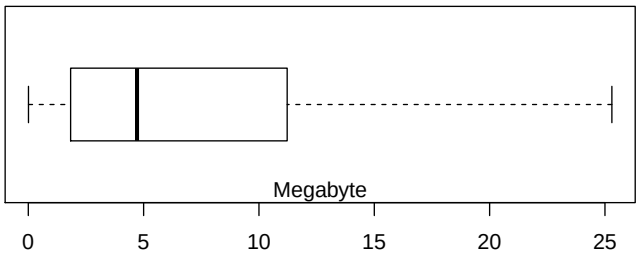

Fig. 3: Distribution of randomly selected apps based on their size (in MB).

3.2 CDA

The design of CDA is straightforward: the main process is summarised in Algorithm 1.

Algorithm 1 Characterising deprecated Android APIs.

```
1: procedure CHARACTERISE(tags)
2:     results ← {}
3:     for each t ∈ tags do
4:         inconsistentAPIs ← ()
5:         method2replacements ← {}
6:         class2comments, method2comments ← construct(t)
7:         for each cls ∈ class2comments.keySet() do
8:             clsAnno ← isAnnotatedAsDeprecated(cls)
9:             clsDocu ← isDocumentedAsDeprecated(class2comments.get(cls))
10:            for each method ∈ cls.getMethods() do
11:                flag ← isAnnotatedAsDeprecated(method)||clsAnno
12:                comment ← method2comments.get(method)
13:                if isDocumentedAsDeprecated(comment)||cDocu then
14:                    ▷ msg here can be null or empty
15:                    msg ← getReplacementMessage(comment)
16:                    method2replacements.put(method, msg)
17:                    if ¬flag then
18:                        inconsistentAPIs.add(method)
19:                    end if
20:                else
21:                    if flag then
22:                        inconsistentAPIs.add(method)
23:                    end if
24:                end if
25:            end for
26:        end for
27:        results.put(t, {inconsistentAPIs, method2replacements})
28:    end for
29:    return results
30: end procedure
```

CDA first parses all Java files in a given release of the Android framework code repository and builds a mapping between Java classes, methods and their documentation (cf. line 6). Then, for each class, CDA checks if it is annotated as deprecated via the *Deprecated* Java annotation. Since documentation and source

code annotation must be consistent, CDA further parses the comments to match the keyword **@deprecated**. If a given class is annotated by *Deprecated* or documented via **@deprecated**, we consider all its methods are flagged as such. After that, CDA goes one step further to perform similar checks to all its methods. If a given method is annotated by *Deprecated* or documented via **@deprecated**, either at the class level or at the method level, we consider it as deprecated.

Based on the aforementioned observations, in a first phase, CDA can pinpoint inconsistency cases where a deprecated API is documented but not annotated (line 18) or is annotated but not documented (line 22). In a second phase, when the API is consistently deprecated, CDA goes one step further to infer the potential replacements of deprecated APIs, attempting to build another mapping between deprecated APIs and their potential replacements which we can later leverage to recommend changes to client app code. Such a mapping can even be leveraged for automated refactoring of Android apps to mitigate the usage of deprecated APIs.

Unlike the original approach presented in the conference version, for which a conservative way is adopted (i.e., matching simply the "Use @link Method" pattern), we have improved the strategy with more strict rules to locate replacement messages. More specifically, CDA obeys the following rules to locate replacement messages: (1) the replacement method must be presented after @deprecated. (2) the replacement method must come before @hide if exists. Our manual observation reveals that @hide is usually presented after @deprecated and it can contain method links (i.e., "@link Method"), which could have been considered as replacement methods in our previous approach. (3) the replacement method must be given via the following patterns: *use/see/call "@link Method" instead.* (4) Finally, if no replacement message can be obtained based on the aforementioned rules, and *@see* is presented after *@deprecated*, we consider the message given by *@see* as the possible replacements as well, which usually provide useful hints for developers to refer to in order to replace the deprecated APIs. Because of these improvements, it is expected that fewer replacement messages will be disclosed compared with the original approach.

Once this process is completed for the first release, CDA loops on all subsequent releases and records the results for our empirical investigation on the evolution.

3.3 Statistics

Table 2 presents statistics on the quantity of code elements that are parsed and analysed by CDA for the different releases of the Android framework. We note that successive releases are constantly increasing in all the different metrics (i.e., the number of files, classes, lines of code, and API methods). Eventually, between level 4 and level 28 (the two extreme API levels in our study), the framework code has substantially grown: the number of classes has tripled, while the number of code lines has almost quintupled; the phenomenon is even more acute in methods which have grown 7-fold. These figures suggest that as time goes by, the framework code base is growing and is potentially becoming more and more complex to analyse and maintain.

Metrics in Table 2 reveal the number of deprecated APIs sharply increases in the framework code base, although the ratio of deprecated APIs vs. the total number of methods remains low (cf. Fig 4). Between level 19 and 21, the ratio has

Table 2: Statistic overview of selected releases. Deprecated APIs are considered as long as they are annotated or documented.

API Level	# Java Classes	LoC	# Total Methods	# Public Methods	# Static Methods	# Deprecated Methods
28	9078	3644369	311259	261593	15299	4309
27	8032	3303839	293223	249299	13579	3341
26	7816	3244981	290872	247442	14015	3383
25	6805	2927464	275264	237666	12456	2884
24	6680	2864293	272554	235991	12092	2865
23	5685	2538626	255411	224930	10207	1916
22	5311	2376430	247793	219729	9493	1645
21	5206	2333200	245446	218233	9324	1477
19	4120	1381169	68365	46625	7292	928
18	3814	1271452	63217	43111	6765	945
17	3835	1248085	62191	42182	6383	910
16	3837	1265976	63232	42779	6396	879
15	3418	1151084	56678	38094	5972	588
14	3387	1137869	55978	37711	5938	596
13	3109	1028975	50806	34324	5498	574
10	2745	872561	43581	29588	4908	431
9	2647	849373	42616	29234	4480	432
8	2913	896503	44947	31460	4678	498
7	2805	841184	42475	29882	4310	462
6	2803	831461	42245	29700	4280	463
5	2807	837932	42368	29776	4288	463
4	2659	774426	39621	27861	3929	354

drastically dropped. Indeed, as shown in Table 2, the total number of methods in level 21 has almost quadrupled comparing to that of level 19 while the number of deprecated methods are only slightly increased.

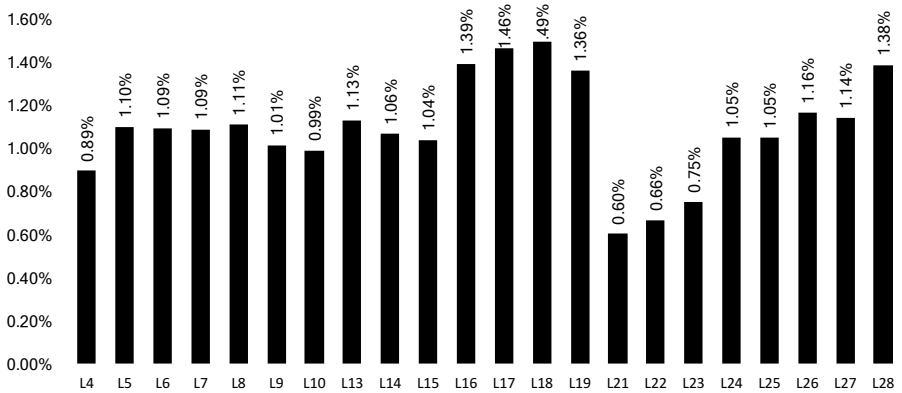

Fig. 4: Distribution of deprecated API rate. For each API level, all its deprecated APIs, including the ones that are deprecated in previous levels, are considered.

4 Empirical Investigation

Our investigations explore the data mined by CDA to answer the following research questions:

- RQ1: Are deprecated APIs properly annotated and documented in the Android framework code base?
- RQ2: To what extent are deprecated APIs stable in the Android framework code base?
- RQ3: How often do maintainers swap deprecated API code with replacement messages? Can such messages evolve over time?
- RQ4: Do app developers quickly react to the deprecation of APIs in the Android framework code base?
- RQ5: For the cases where developers do react to deprecated APIs, how long does it take for them to make the update?
- RQ6: When dealing with deprecated APIs, how often do app developers replace them with their alternatives recommended by the Android maintainers?

All the experiments discussed in this section are performed on a Core i7 CPU running a Java VM with 16GB of heap size.

4.1 Code Annotation and Documentation

Table 3: Inconsistency between annotation and documentation for deprecated Android APIs. We have submitted two issues (one for each inconsistent type) to the Android open source project and have received positive acknowledgements on confirming these two issues.

Inconsistent Type	L4	L5	L6	L7	L8	L9	L10	L13	L14	L15	L16
Annotated-Not-Documented	17	13	13	12	12	4	2	16	16	16	49
Documented-Not-Annotated	109	102	102	102	132	105	105	72	75	64	230
Inconsistent Type	L17	L18	L19	L21	L22	L23	L24	L25	L26	L27	L28
Annotated-Not-Documented	48	42	19	41	42	85	118	118	122	114	125
Documented-Not-Annotated	232	236	233	311	308	343	348	349	280	280	341

Code annotation and documentation are both necessary to properly indicate that an API is deprecated. If an API is deprecated without an explicit mention in the documentation (i.e., Annotated-Not-Documented), users will not be clearly informed by this deprecation, nor will they know the alternative, and thus may still use deprecated APIs. Similarly, if an API is deprecated without an explicit annotation in the source code (i.e., Documented-Not-Annotated), although its deprecation can still be highlighted on the documentation site (cf. Figure 1), such API will be compiled and integrated into the released SDKs and thus popular IDEs such as Android Studio and Eclipse cannot perform checks and warnings to developers about this deprecation. As indicated in Figure 1, API *saveLayer* is actually deprecated. However, since this method is not properly annotated, when accessing this method via Android Studio, as presented in Figure 5, the method will not be marked as deprecated (e.g., with a cross-line). In contrast, API *clipRegion()*, which is annotated by an explicit deprecation annotation, is correctly flagged by Android Studio as deprecated.

We would like to remind the readers that modern IDEs might be able to also cross out such APIs that are only marked as deprecated in the Javadoc comment. However, we argue that the consistency between the @deprecated tag in Javadoc and the @Deprecated annotation in Java code is very important. First of all, the @deprecated Javadoc tag is not part of the Java standard. Hence, there is no guarantee that all compilers will always issue warnings based on the @deprecated tag. Second, the @deprecated Javadoc tag cannot be read by Java code at runtime (e.g., via reflection), making it inconsistent with the actual behaviour it was intended to be.

```
Canvas c = new Canvas(null);
//deprecated without annotation
c.saveLayer(null, null, Canvas.ALL_SAVE_FLAG);
//deprecated with annotation
c.clipRegion(null);
```

Fig. 5: Android Studio does not provide indication to such deprecated methods (e.g., *saveLayer* as indicated in Figure 1) that are not properly annotated.

In this study, we are interested in checking whether deprecated APIs provide consistent documentation and annotation. Surprisingly, CDA unveils a significant number of cases where the documentation is not consistent with deprecation annotation presence/absence. This inconsistency has been confirmed by the Android team as an actual problem of the Android framework code base. Table 3 summarises statistics of cases found in the various framework releases. We note that deprecated APIs are generally well documented. Nevertheless, there do exist a number of cases where inconsistency appears. Generally, the number of *Annotated-Not-Documented* cases of inconsistencies are smaller than that of *Documented-Not-Annotated*. This finding suggests that Android framework developers are not yet aware of the inconsistency problem of deprecated APIs. This observation is further confirmed by the fact that inconsistent deprecations appear to be rarely fixed during the evolution of the Android framework code base. For the rare cases where inconsistent deprecations disappear during the evolution, our further analysis reveals that all of them are due to the removal of deprecated APIs themselves.

Previously, we have written issue reports describing the inconsistency cases (2 Annotated-Not-Documented and 34 Documented-Not-Annotated deprecated APIs) that CDA has identified for the selected set of Java classes from the Android framework base (i.e., version 26, tag *android-8.0.0_r9*). These issue reports were submitted to the Android issue tracker system under *developer.android.com* and *source.android.com* components, respectively. The submitted issues were assigned and confirmed by Android maintainers in a day: the engineering team has acknowledged the issues and promised to fix them for next releases.[12] In addition to the aforementioned issue reports, we have also reported the newly harvested results to Google and are now waiting for the response.

[12] As footnoted before, the issue IDs of the submitted bugs are 69105065, 69104762 and 69098890, where the status of these issues so far are *Fixed*, *Assigned* and *Assigned*, respectively.

RQ-1 Finding

Deprecated Android APIs can be inconsistently annotated and documented. With CDA, we have systematised the identification of such inconsistency issues. Eventually, Android project maintainers recognise that these inconsistency cases are indeed issues that must be addressed.

4.2 Clean-up and Survival of Deprecated APIs

We now investigate whether the code base is eventually cleaned-up from deprecated APIs, and what is otherwise the survival time of an API once it is marked as deprecated. To this end, we perform pairwise comparisons between every consecutive API level releases of the framework. As illustrated in Fig. 6, compared with the total number of deprecated APIs available in a given API level, the majority of deprecated APIs are actually retained in the framework until the latest API level of this study (i.e., level 28). During the evolution of the framework, only 808 out of 5,118 deprecated APIs (around 16%) are removed.

Table 4 summarises the added and removed APIs for each update (i.e., the code changes between a consecutive pair of releases considered in our study). Almost all of the updates (except for L5 → L6) have performed some clean-up for deprecated APIs. This finding suggests that it is important that app developers take steps to address deprecated APIs used in their client code, or they may otherwise face runtime crashes (hence bad user experience, and poor ratings) on latest devices (Li et al., 2018a).

Table 4: The number of added and removed deprecated APIs for each update.

Update	Addition	Removal	Update	Addition	Removal
L4 → L5	112	3	L16 → L17	115	84
L5 → L6	0	0	L17 → L18	53	18
L6 → L7	2	3	L18 → L19	34	51
L7 → L8	67	31	L19 → L21	568	19
L8 → L9	19	85	L21 → L22	173	5
L9 → L10	1	2	L22 → L23	402	131
L10 → L13	207	64	L23 → L24	1016	67
L13 → L14	32	10	L24 → L25	19	0
L14 → L15	6	14	L25 → L26	529	30
L15 → L16	320	29	L26 → L27	56	98
L16 → L17	115	84			

We further go one step deeper to check how deprecated Android APIs are removed from the framework code base. Our investigation reveals that, apart from the physically removed deprecated APIs, around 15% of the remaining APIs are tagged as hidden (i.e., marked via *@hide* in the comment of the method). Those APIs are not "actually" physically removed from the framework but will be excluded from the public Android SDK (i.e., app developers cannot access them) and they are known to be subject to removal during the evolution of framework code (Li et al., 2016c).

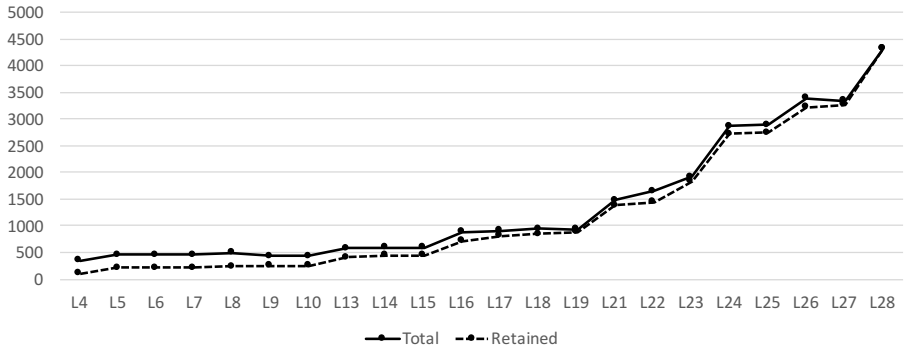

Fig. 6: The number of deprecated APIs (in accumulation) retained in the framework.

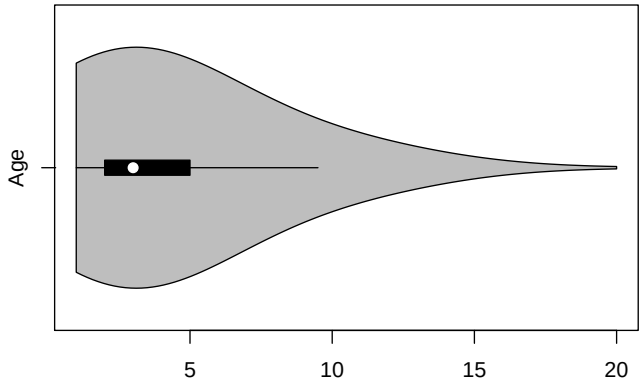

Fig. 7: Violin distribution of the life expectancy of deprecated Android APIs. Age corresponds to the number of generations (i.e., X-axis) before a deprecated API is removed from the Android framework since it is deprecated.

As shown in Table 4 (i.e., the second column), in addition to removal, there are new Android APIs recurrently flagged as deprecated as well. We therefore investigate the life expectancy of such Android APIs once they are marked as deprecated by maintainers. We model life expectancy as the number of releases where a deprecated API survives in the code base before being removed. We also consider a release as a code "generation[13]". Figure 7 presents the violin plot on the life expectancy distribution of deprecated Android APIs. The median number of generations a deprecated API is removed in the code base is 3 ($mean = 4.171$). Given the fact that the Android framework code base evolves at a fast pace (a generation occurs every 3 months(McDonnell et al., 2013)), app developers need to react quickly on replacing deprecated APIs in their client code before they become inaccessible in updated devices.

It can be observed from the results shown in Figure 8, 188 deprecated APIs (around 4% of total deprecated APIs) are removed after one update. Although this rate is low, we are still surprised that this situation does happen during the

[13] The actual time can be computed based on the released time of selected tags (e.g., android-7.0.0_r7 is released on 2016-08-23 while android-6.0.1_r9 is released on 2015-12-15).

evolution of the Android framework code base. Because of the limited time window, app developers may not yet be informed (i.e., the deprecation cycle is ignored) and hence may still leverage those deprecated APIs, resulting in immediate crashes on devices running next framework versions.

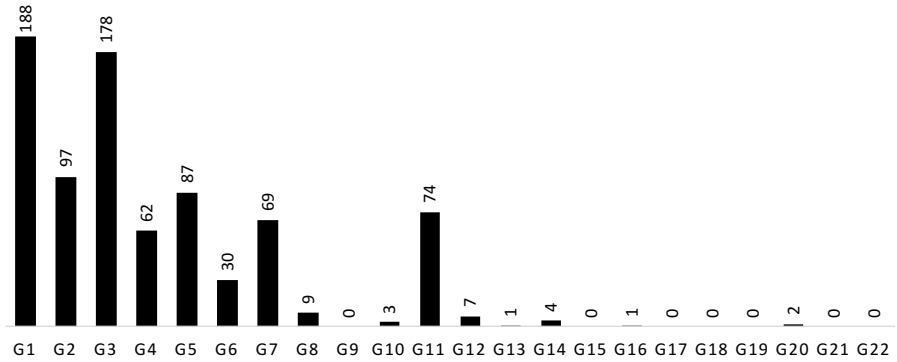

Fig. 8: Life expectancy of deprecated Android APIs. Age corresponds to the number of generations (e.g., G1 means one generation, or one release) before a deprecated API is removed from the Android framework.

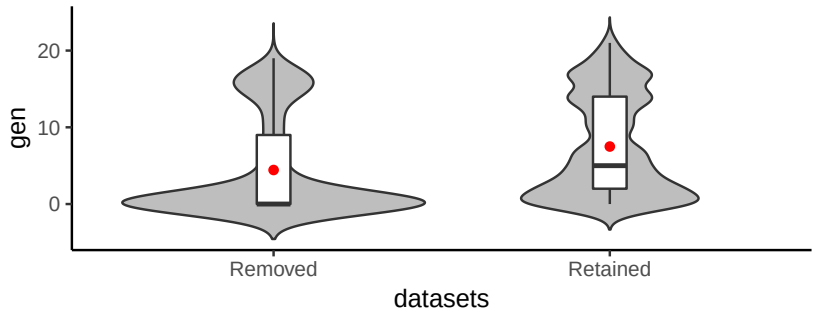

Fig. 9: Violin distribution of the number of generations Android APIs get deprecated (between removed and retained deprecated APIs).

Finally, we look into the number of generations Android APIs get deprecated after their introduction to the framework, i.e., from their birth to their deprecation. Fig. 9 illustrates the violin distribution of the generations between removed and retained deprecated APIs. The fact that these two distributions are significantly different suggests that the newer APIs get deprecated, the more likely they will be removed. This result is actually expected as the longer an Android API stay in the framework, the more dangerous to remove it from the framework even if it becomes deprecated. Indeed, the APIs existed longer in the SDK will have a higher chance to be used by client apps. Removing those APIs might break the execution of the client apps, resulting in app crashes and thereby poor user experience, for which framework maintainers would not want to confront.

RQ-2 Finding

Deprecated Android APIs are regularly cleaned-up from the framework code base, often by completely dropping the code, or by making it *hidden*. Half of these removals are performed in a short period of time (e.g., within 3 API level generations), requiring developers to quickly react on deprecated APIs.

4.3 Replacements for Deprecated APIs

In order to facilitate the usage updates of deprecated APIs in Android apps, and consequently to preserve backward compatibility, APIs should always be deprecated with clear replacement messages (i.e., how can this method be replaced by other ones?) (Monperrus et al., 2012). However, in practice, there is evidence that API elements are usually deprecated without such messages (Robbes et al., 2012) (Brito et al., 2016) (Hora et al., 2015): developers thus may not be provided with suggestions of how to avoid the use of deprecated APIs. We explore in this study the availability of replacement messages for Android deprecated APIs.

Since version 1.2, Java documentation recommends that developers should include *"Use {@link Method}"* to indicate the replacement API when deprecating a given API. CDA searches this pattern[14] in the Javadoc and builds a mapping between deprecated APIs and their replacements. Table 5 presents some examples from the built mapping. Replacement messages often refer to other API methods, but may also refer to some object fields (e.g., #onReceive).

Table 5: Examples in the constructed mapping.

Deprecated API	Replacement Message
android.database.sqlite.SQLiteClosable: void onAllReferencesReleasedFromContainer()	#releaseReferenceFromContainer()
android.webkit.WebSettings: void setDefaultZoom(ZoomDensity)	ZoomDensity#MEDIUM
android.app.admin.DeviceAdminReceiver: void onReadyForUserInitialization(Context,Intent)	#onReceive
android.content.Context: void removeStickyBroadcast(Intent)	#sendStickyBroadcast
android.database.Cursor: void deactivate()	#requery

Figure 10 illustrates the distribution of deprecated APIs with/without replacement messages for the considered API level releases. A median percentage of 64.29% deprecated APIs have been explicitly documented with replacement messages. The latest release (i.e., level 28) has replacement messages for 78.6% (i.e., 3386) of total deprecated methods. This replacement rate has slightly increased compared to the one we have computed based on non-class-level deprecated methods. This growth may be contributed by the fact that, when considering the class-level deprecated methods, one replacement at the class level will directly apply to all its methods, which consequently will increase the likelihood of having replacement message for a given deprecated APIs.

Despite that the majority of deprecated APIs have been provided with replacement messages, it is still surprising to see that around 20% to 35% of deprecated APIs are deprecated without giving replacement messages. Towards understanding the rationale behind this, we resort to a manual process to go through the comments of deprecated APIs and have observed the following reasons:

[14] Following the rules illustrated in Section 3.2.

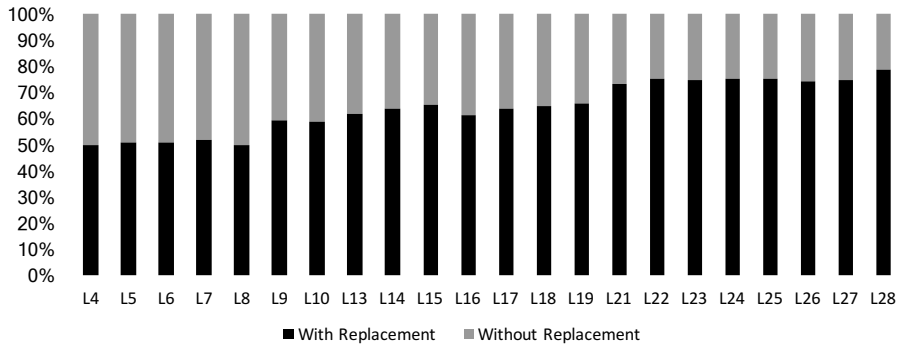

Fig. 10: Distribution of deprecated APIs per release with/without replacement messages.

– Hidden or internal APIs may be deprecated without giving alternatives. As shown in Fig. 11, around 35% of deprecated APIs without replacement messages are hidden or internal ones. Since those APIs are not meant to be used by app developers, framework maintainers may treat it differently compared with other APIs.
– Interestingly, there are some APIs that are not hidden/internal but have been only used by hidden/internal APIs. For example, method *requestWifiBugReport()*[15] of class *com.android.server.am.ActivityManagerService* has only been accessed by internal components. When deprecating these APIs, it is also possible that no replacement messages will be given.
– The methods in the testing code (e.g., unit test cases) are less likely deprecated with replacement messages. Indeed, testing code (because it will not be shipped to the final product) may not be maintained in the same way as that of the core Java classes. In this extension, we have considered much more Java modules, which may have received different attention to their deprecation-then-updating qualities. Nonetheless, we argue that even the least important module should be maintained in the same way, so as to help to keep the coding style consistent and reduce the likelihood of making mistakes when maintaining the code.
– Some deprecated APIs are simply flagged as "Do not use" without mentioning any alternatives. These APIs may reflect the functions that are no longer needed in the framework.

We now investigate whether the replacement messages provided for deprecated APIs are reliable. Concretely, we check that the provided replacement messages are stable (i.e., whether they evolve as well). To this end, we conduct a study on two aspects:[16] (1) Will deprecated APIs that have no replacement messages be complemented later with replacement messages? (2) Will the replacement messages of deprecated APIs be updated by new replacements?

[15] Comment Message: This method is only used by a few internal components and it will soon be replaced by a proper bug report API (which will be restricted to a few, pre-defined apps).

[16] In this experiment, only the APIs that are explicitly deprecated at the method level are considered. When deprecating APIs at the class level, i.e., deprecating classes, it will unlikely to provide replacement messages to their methods.

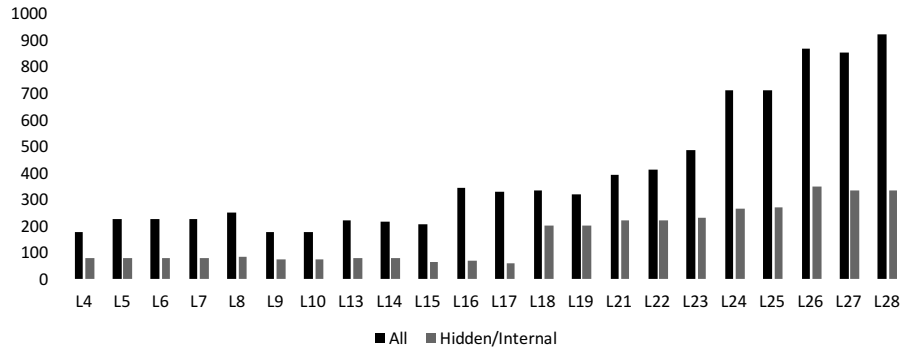

Fig. 11: Distribution of hidden/internal deprecated APIs that are deprecated without providing replacement messages.

We find that: (1) It is unlikely that replacement messages will be added to such deprecated APIs that initially have no replacement message. We only find 22 APIs (cf. Table 6 with five examples); and (2) seldom, an existing replacement message will be updated: we identified only 72 API cases (cf. Table 7 with five examples) where the original replacement messages are updated with new ones. This finding suggests that framework maintainers need to be extremely careful about the documentation, especially w.r.t the replacement messages since this documentation will remain available for a long time and will likely have an effect on app developers code.

Table 6: Five samples of Newly Added Replacement Messages.

API	Added Replacement Message
<android.os.FileUtils: boolean copyFile(File,File)>	#copy(File, File)
<android.os.FileUtils: boolean copyToFile(InputStream,File)>	#copy(InputStream,OutputStream)
<android.media.AudioManager: boolean isWiredHeadsetOn()>	AudioManager#getDevices(int)
<android.graphics.Canvas: Matrix getMatrix()>	#isHardwareAccelerated()
<android.telephony.NeighboringCellInfo: void setRssi(int)>	#NeighboringCellInfo(int,String,int)

Table 7: Five samples of updated replacement messages.

Replacement Message (original)	Replacement Message (new)
#SslCertificate(String,String,Date,Date)	#SslCertificate(X509Certificate)
#setTextZoom(int)	#setTextZoom
#getTextZoom()	#getTextZoom
#BitmapDrawable(Resources)	#BitmapDrawable(android.content.res.Resources,android.graphics.Bitmap)
#onInflate(Activity,AttributeSet,Bundle)	#onInflate(Context,AttributeSet,Bundle)

RQ-3 Finding

About 78% of deprecated Android APIs have been commented with replacement messages, which however, either exist or not, will unlikely to be updated during the evolution of the Android framework code base.

4.4 Developer Reactions

We study the reactions of app developers to the deprecation of Android APIs. More specifically, we would like to know if deprecated APIs are still used by app developers. Since app assembly time (the compilation of the DEX file in the APK) is not reliable (e.g., it is easily manipulable) (Wang et al., 2015), we resort to API level generations as the measure of time. For each app, we extract its API level based on the *targetSDK* attribute declared in app manifest files. The target SDK version informs the system that the app has been tested against the target version, which hence should not cause any compatibility issues. After the extraction of targeted SDK version, CDA goes through all the method calls of the analysed app to check if some used APIs have been deprecated in releases prior to the declared targeted SDK version. Specifically, given a compiled Android app, CDA leverages Soot, a well-known bytecode manipulation and optimization framework, to transform its bytecode to Jimple code, a 3-address intermediate representation designed to simplify analysis and transformation of Java/Android bytecode. All the method calls are then compared at the Jimple level. If a method call is matched with the signature[17] of an Android API, we consider an Android API usage is identified.

Among our randomly sampled set of 10,000 apps, CDA highlights that 61.97% apps are making use of deprecated APIs. Among the flagged 6,197 apps, the GPlay subset contributes 3,941 apps while NGPlay contributes 2,256 apps. This finding is very interesting as we would have expected that there should be less apps in Google Play accessing deprecated APIs than that of other markets as normally Google Play provides high-quality apps comparing to other alternative markets. Moreover, as shown in Fig. 12, Google Play apps also utilise more deprecated APIs than that of alternative markets. We ensure that this difference is significant by conducting a Mann-Whitney-Wilcoxon (MWW) test,[18] where the resulting p-value confirms that there is a significant difference between Google Play and alternative markets apps at a significance level[19] of 0.001. Cohen's d, which is of practical interest to estimate the magnitude of the difference, further suggests that the effect size between these two sets of deprecated API usages is median (equals to 0.67).

Towards understanding the reason why Google Play apps access deprecated APIs, we further record all the callers of deprecated APIs. Our investigation reveals that actually most of the deprecated APIs are accessed by third-party libraries.[20] Indeed, the number of deprecated APIs accessed by libraries (i.e., 256,325) is almost doubled by that of app code (i.e., 141,359). Table 8 further highlights the top 10 caller packages that have invoked deprecated APIs in Google Play and Third-party market apps, respectively. Library *android.support* remains to be the top leveraged one in both Google Play and third-party apps. This is expected as the main reason why the *android.support* library is introduced is to safely access

[17] Declared class name, method name, and arguments.

[18] We have appended zero to third-party markets (i.e., NGPlay) to balance the number of elements.

[19] Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, there is one chance in a thousand that the difference between the datasets is due to a coincidence.

[20] In this work, we consider the common libraries revealed by (Li et al., 2016a) as the white-list to flag whether a caller belongs to libraries. This white-list contains over 1,000 common libraries mined from over 1.5 million Android apps.

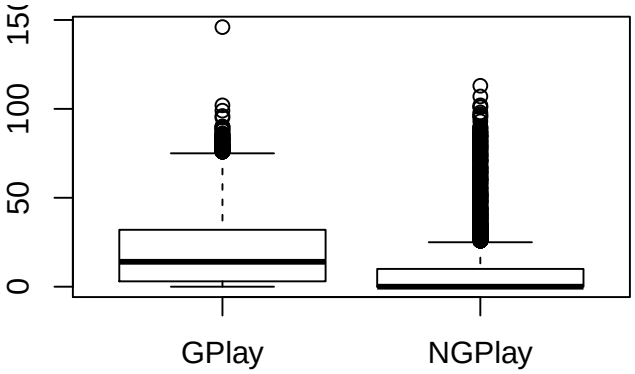

Fig. 12: Distribution of the number of deprecated APIs utilised per app.

Table 8: The top 10 packages calling into deprecated Android APIs (the total times appearing in the considered apps), which account for over 90% and 80% of total deprecation usages in Google Play and Third-party Markets, respectively.

GPlay Apps		NGPlay Apps	
android.support	105,182	android.support	22,739
com.google	43,040	com.tencent	12,582
com.facebook	5,301	com.umeng	5,432
org.apache	4,381	com.baidu	5,127
com.unity3d	3,452	com.alipay	3,194
com.biznessapps	2,621	com.google	3,080
com.adobe	2,495	com.unity3d	2,851
com.good	1,143	com.sina	1,265
com.flurry	1,084	com.adobe	1,002
com.paypal	1,072	cn.jpush	993

historical APIs that are deprecated from the latest framework version. Indeed, the Android framework regularly deprecates APIs, which could be eventually removed from the system, app developers are recommended to include this library for solving possible backwards-compatibility issues. Apart from that, the usage of other libraries is quite different between Google Play and third-party apps. For example, library *com.google* is the second top leveraged library in the Google Play set while is only the sixth in the third-party set. Nevertheless, the fact that the number of deprecated APIs are significant in both app sets suggests that common libraries, especially such ones that are provided by well-known parties such as Google, are not frequently updated in developer app code.

In addition to the frequency enumerated in Table 8, we have also investigated the number of deprecated APIs accessed by each library. Fig. 13 further illustrates the distribution of the number of deprecated APIs leveraged by each library, where only such libraries that have accessed into at least one deprecated API are considered. The median and mean numbers of accessed APIs are 6 and 13.01, respectively. This result further backups our previous finding: many deprecated APIs are actually accessed by Android app code via popular libraries. The fact that app developers are not recurrently updating the libraries used in their apps could be explained by the empirical findings disclosed by (Derr et al., 2017): app developers

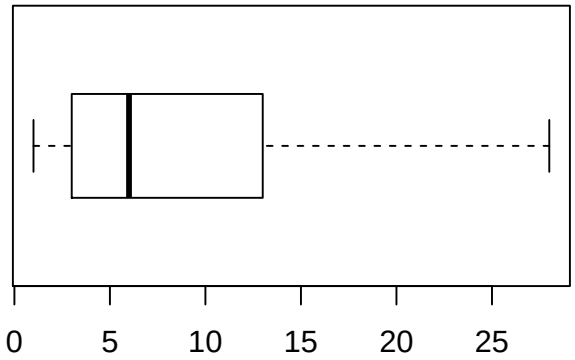

Fig. 13: Distribution of the number of deprecated APIs leveraged by common libraries.

are hesitated to update libraries in order to avoid ostensible re-integration efforts and version incompatibility problems, based on a survey of 203 app developers from Google Play on their usage of libraries and requirements for more effective library updates.

We explore the gap between the targeted SDK level and the API deprecation level, indicative of time delay, i.e., $delay = targetSDK - deprecationLevel$. This delay represents the number of generations where app developers are still able to call deprecated APIs. In this work, the targeted SDK version is chosen to compute the delay. Ideally, the supported SDK ranges (from minimal SDK version to the latest version) should be considered. However, it is hard to represent the results over a range of SDK versions, while the minimal SDK versions provided by app developers are usually small and hence may not be representative to recently deprecated APIs. Indeed, as shown in Fig. 14, the distribution of minimal SDK versions and targeted SDK versions are significantly different (as confirmed by a WMM test). The median minimal version is only at 9, which was released in 2010. Cohen's d (equals to 1.88) suggests also a large effective size between these two sets. Instead of choosing the minimal SDK version, we leverage the targeted SDK version to compute the delay. We believe this version is more suitable for our experiments. Since our idea in this work is to check how app developers react to deprecated APIs at the development time, the targeted SDK version actually reflects the desired version that the app is developed for.

The delay computed based on thousands of deprecated APIs ranges from 1 to 21. Fig. 15 further presents the distribution of API level delays between Google Play and third-party market apps. The callers of deprecated APIs are also separated into two folds: app code and common library code. Interestingly, although most deprecated APIs are leveraged by library code, their accessing delay is however shorter than that of app code for Google Play apps. This difference is also further confirmed by a MWW test.

Besides a small number of deprecated APIs, most APIs accessed by Android apps should be normal APIs, i.e., they are not deprecated at the API level that the apps target. Since the Android framework evolves fast, those normal APIs might become deprecated in future as well. Towards verifying this assumption, we explore again the gap between the targeted SDK level and the API level when the API is deprecated despite it is not deprecated at the targeted SDK level, i.e.,

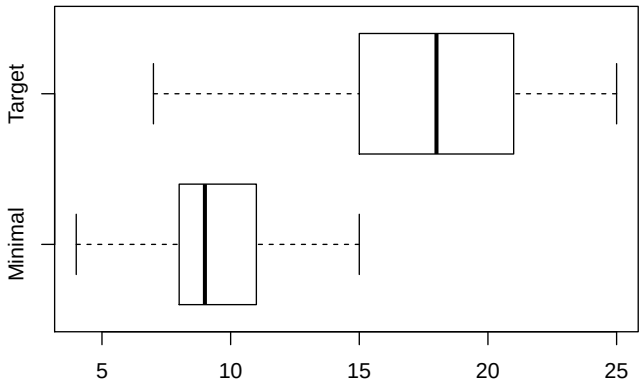

Fig. 14: Distribution of the minimal and targeted SDK versions of the selected Android apps.

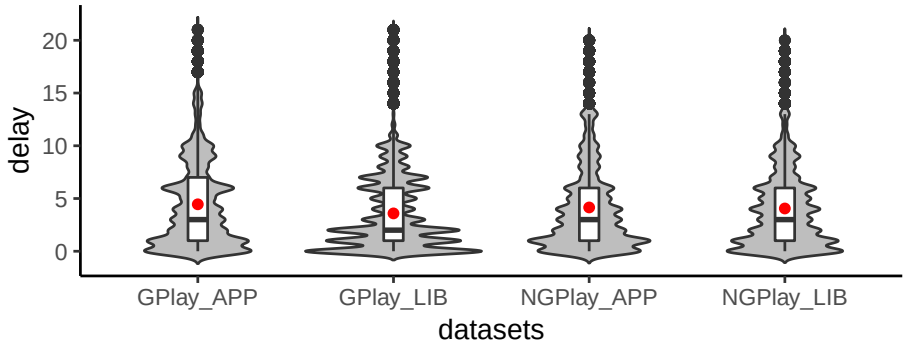

Fig. 15: Distribution of delays between the usage of deprecated APIs in Google Play and third-party market apps. The red line indicates the mean value of each violin plot. Suffixes $\_APP$ and $\_LIB$ indicate that the caller of deprecated APIs are from the app code and third-party library code, respectively.

$generationGap = deprecationLevel - targetSDK$. This gap represents the number of generations that app developers need to be aware of so as to be able to react on the deprecations on time. Fig. 16 illustrates the distribution of the generation gaps of the APIs to-be deprecated. Normally, half of the selected APIs (or eventually deprecated APIs) will be deprecated in less than six generations (between one year to two years). This evidence suggests that app developers should continuously update their apps. Otherwise, even if a given app is well developed at the moment (i.e., it does not access into any deprecated API), it could still become less-maintained. As time goes by, the number of deprecated APIs accessed by Android apps (if without any change) will likely increase, resulting in a bigger probability of being incompatible with the latest devices.

Fig. 17 further illustrates the distribution of the number of APIs that will become deprecated per app. The fact that over half of the apps have accessed into around 95 APIs that will be deprecated eventually emphasises that Android apps need to be continuously updated.

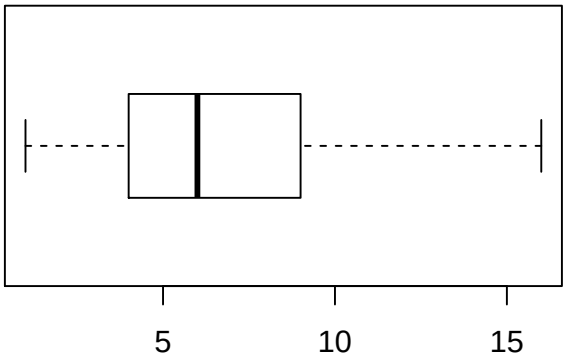

Fig. 16: Distribution of the *generationGap*s among the selected apps.

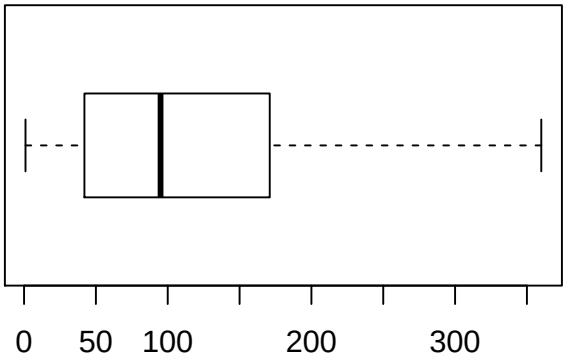

Fig. 17: Distribution of the number of APIs that will become deprecated per app.

RQ-4 Finding

Most deprecated APIs are accessed by app code via popular libraries. Developers should thus pay attention in the library releases used in their app packages.

4.5 Evolution of the Usage of Deprecated APIs

Based on the targeted API level declared in Android apps, our experimental results towards answering the previous research question reveal that app developers may still use such APIs that are already deprecated at the time of implementing. In this research question, we go one step further to investigate how long will it take for app developers to make the update when dealing with deprecated APIs. In other words, we would like to keep track of the code changes during the evolution of Android apps. To do so, we need to collect a set of Android app lineages, where each lineage is formed with the different versions of the same Android app. To this end, we resort to AndroZoo again to harvest such datasets.

We randomly select 500 app lineages from the the dataset provided by (Gao et al., 2018), in which the authors have re-constructed the app lineages by consid-

ering all the AndroZoo apps. The 500 app lineages contain in total 8,989 Android apps. Each lineage contains at least 10 apps that (1) share the same package name; (2) are signed by the same certificate; and (3) are released to the same app market (e.g., Google Play). The apps inside a lineage are also ordered based on their declared versions. In this work, we consider the different versions as generations. For example, given an app lineage *com.facebook.katana* $\{g_1, g_2, ..., g_{32}\}$, we call the first app version as the first generation of the app and the last version as the 32nd generation of the app. The targeted SDK versions of lineage apps can be updated when the app itself is updated. Among the 500 lineage apps, around 18% of them have involved with cases where the targeted SDK versions are updated. Often, the updates in terms of the SDK versions are in a small range. This phenomenon is expected as the targeted SDK version changes can lead to significant refactorings of the app code. App developers may not be interested in doing that as it not only introduces more works to them but also increases the possibility of introducing bugs to the app code. Moreover, the larger changes of the SDK version, the more refactorings might need to be applied to the app code, resulting in even more works for developers to deal with. Finally, since there is no enforcement from the Android system to restrict the usage of deprecated APIs, app developers are not motivated to update the SDK versions. Even without changing the SDK version, most apps should still be able to run on modern devices since the majority of deprecated APIs are not really removed from the framework.

Fig. 18 presents the distribution of the number of generations among the 500 app lineages we have randomly selected for this experiment. The number of app generations ranges from 10 to 108, with a median and mean generations at 14 and 17.9, respectively. This distribution illustrates the diversity of our randomly selected app lineages as well.

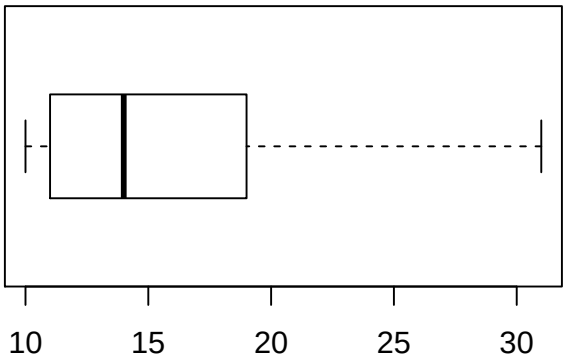

Fig. 18: Distribution of the number of generations in each app lineage.

Based on the selected app lineages, we first look at the problem whether app developers remove deprecated APIs during the evolution of their Android apps. Fig. 19 illustrates the distribution of the number of deprecated APIs that are (1) stayed in the app until the last generation and (2) removed eventually from the app. The median and mean numbers of deprecated APIs are 75.5 and 78.5 for retained ones and 19.5, 34.45 for removed ones, respectively. Cohen's d (equals

to 0.92) suggests a large effective size between these two distributions. Clearly, most deprecated APIs are retained in the app rather than removed, demonstrating that deprecated APIs have not received enough attention from app developers. Similar findings have also been observed by researchers on other platforms such as the JDK (Sawant et al., 2018c) and the Smalltalk ecosystem (Robbes et al., 2012). Furthermore, among the removal cases of deprecated APIs, around 15% of the attempts remove APIs immediately in a subsequent app version and have happened in over half of the selected lineages. Unfortunately, all the involved deprecated APIs, which are removed immediately in some lineages, have appeared to be the cases that the APIs are removed after at least two generations (or app versions). This empirical finding suggests that, at least based on the 500 randomly selected app lineages, we cannot observe any pattern indicating a sense of urgent fixes to deprecated Android APIs.

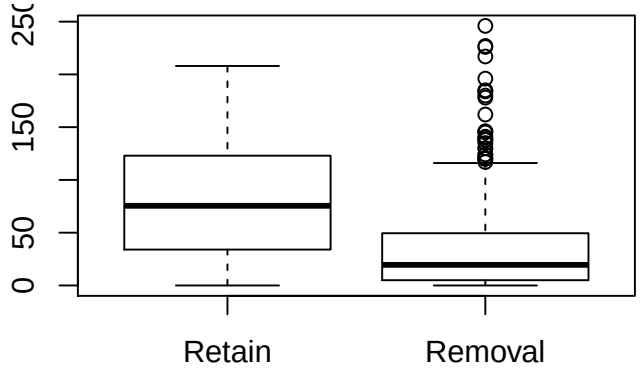

Fig. 19: Distribution of the number of deprecated APIs that are (1) stayed in the app until the last generation and (2) removed eventually from the app.

Table 9: The top 10 retained deprecated APIs.

API	Frequency
<android.view.View: void setBackgroundDrawable(android.graphics.drawable.Drawable)>	409
<android.content.res.Resources: android.graphics.drawable.Drawable getDrawable(int)>	397
<android.app.Notification: void setLatestEventInfo(Context,CharSequence,CharSequence,PendingIntent)>	389
<android.net.NetworkInfo: int getType()>	385
<android.view.Display: int getWidth()>	377
<android.view.Display: int getHeight()>	373
<android.content.res.Resources: int getColor(int)>	347
<org.apache.http.params.HttpConnectionParams: void setConnectionTimeout(org.apache.http.params.HttpParams,int)>	353
<org.apache.http.params.HttpConnectionParams: void setSoTimeout(org.apache.http.params.HttpParams,int)>	350
<android.content.res.Resources: int getColor(int)>	347

Table 9 and Table 10 further respectively list the top 10 retained and removed deprecated APIs, among the evolution of the 500 selected app lineages. The fact that the two lists of APIs are totally different from each other shows that app developers may have special focuses when dealing with the replacement of deprecated APIs. There are various reasons that may attract the attention of app developers for those deprecated APIs that are actively updated. For example, it could be the case that those APIs have clear alternative APIs mentioned in the Android documentation, or those APIs are later removed from the framework so

Table 10: The top 10 removed deprecated APIs.

API	Frequency
<android.app.Notification: void <init>(int,java.lang.CharSequence,long)>	160
<android.widget.PopupWindow: void setWindowLayoutMode(int,int)>	111
<android.app.Activity: void setProgress(int)>	107
<org.apache.http.params.HttpConnectionParams: void setSocketBufferSize(org.apache.http.params.HttpParams,int)>	106
<org.apache.http.conn.ssl.SSLSocketFactory: org.apache.http.conn.ssl.SSLSocketFactory getSocketFactory()>	106
<android.content.ContentProviderClient: boolean release()>	100
<org.apache.http.params.HttpParams: org.apache.http.params.HttpParams setParameter(java.lang.String,java.lang.Object)>	99
<android.accessibilityservice.AccessibilityServiceInfo: java.lang.String getDescription()>	99
<android.accessibilityservice.AccessibilityServiceInfo: boolean getCanRetrieveWindowContent()>	99
<android.text.Html: java.lang.String toHtml(android.text.Spanned)>	96

that developers have to remove them in order to make their app compatible to the latest devices.

Towards verifying these hypotheses, we conduct two more experiments attempting to understand the rationale behind. Our observation reveals that, although the top 10 retained and removed lists are different, the majority of APIs (over 90%) are actually shared by these two sets, i.e., it is likely that a given deprecated API is retained by some developers while removed by others. Apart from the majority APIs that are both retained and removed by app developers, the numbers of retained and removed APIs that have been documented with replacements are more or less the same (also around 80%), indicating that having clear alternatives is not the main reason for app developers to address deprecated APIs.

In terms of actual removal of deprecated APIs, in our experiments, there are only 20 APIs that are accessed by the apps of the selected lineages that have been removed from the framework. Interestingly, all the 20 APIs have been involved with removal in the selected app lineages. This evidence suggests that app developers are more likely to deal with such deprecated APIs that are eventually removed from the framework. Surprisingly, 17 out of the 20 APIs have been also retained by some app lineages. We speculate that this might be correlated to the quality of app lineages. Indeed, as argued by (Gao et al., 2019), some app developers tend to write poor quality apps. Even with critical features such as crypto-API usages, developers are frequently making mistakes.

Fig. 20 further illustrates the distribution of generations that app developers take to remove deprecated APIs. All the 10 APIs share more or less a similar trend: for at least half of the cases, app developers take around five generations to remove a deprecated API, suggesting that app developers do not frequently update deprecated APIs. Furthermore, the average generation (indicated by the red dots) that the API is removed is always bigger than the median generation. Except for the top 10 APIs, this trend is also generally true for all the deprecated APIs, as illustrated in Fig. 21.

RQ-5 Finding

During the evolution of Android apps, deprecated APIs are likely to stay in the app. For the cases where deprecated APIs are frequently removed from the apps, they are generally different from the ones that are constantly retained.

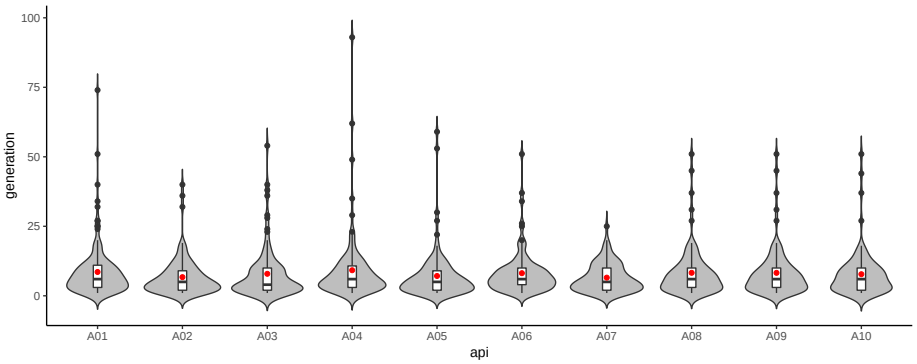

Fig. 20: Distribution of generations app developers take to remove deprecated APIs (only the top 10 deprecated APIs shown in Table 10 are illustrated). The APIs illustrated in this figure (i.e., A01-A10) follow the same sequence as that enumerated in the table (e.g., A01 stands for method *<init>()* of class *Notification* while A10 stands for method *getTypeName()* of class *NetworkInfo*.

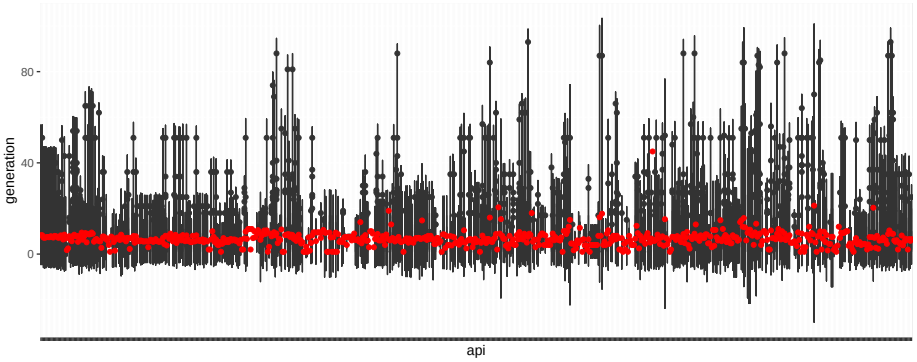

Fig. 21: Distribution of generations app developers take to remove deprecated APIs (all the involved APIs are considered).

4.6 Updating Deprecated APIs

Although we found in our study that over half of the deprecated APIs come with replacement messages indicating alternatives, we have no confirmation that the proposed alternatives are indeed suitable for app developers and the scenarios in which they used the deprecated APIs. Building on a large dataset of apps with several release versions per app (i.e., app lineages), we can investigate how do app developers deal with deprecated APIs, e.g., the deprecated APIs are simply removed from the app or will developers actually replace them with their alternatives recommended by the document?

Towards answering the aforementioned research question, we conduct another exploratory study of the previously selected lineage apps. In this term, we conduct pairwise comparisons between two subsequent app versions in a lineage, aiming to extract code changes (i.e., *diff*s) that have involved with the removal of deprecated APIs. Given a pair of two subsequent apps (e.g., $g_x \rightarrow g_{x+1}$), if a deprecated API

is removed from the body of a method m in g_x and the method m is still presented in g_{x+1}, we consider that the developer has been aware of the deprecated nature of the API and has performed a dedicated treatment. Consequently, we extract the changed code and represent it into a diff snippet.

Listing 2 illustrates an example of an extracted diff snippet, which are extracted from app lineage *air.com.playsino.bingo.thanksgiving*. Because API *getWidth()* of class *android.view.Display* is deprecated, developers remove it from the app (cf. Line 9). As demonstrated in the diff code, the similar functionality is achieved by accessing into another API called *getSize()* of the same class (cf. Line 13 and then line 10). It is worth mentioning that the replacement API, namely *getSize()*, is actually recommended and explicitly highlighted in the Android documentation.

```
1 -    $i3 = virtualinvoke $r6.<android.view.Display: int getHeight()>();
2 +    $r7 = new android.graphics.Point;
3 +    specialinvoke $r7.<android.graphics.Point: void <init>()>();
4 +    $z0 = $r0.<com.adobe.air.AIRWindowSurfaceView: boolean
       mIsFullScreen>;
5 +    if $z0 == 0 goto label04;
6 +    label01:
7 +    $i3 = $r7.<android.graphics.Point: int y>;
8      $r0.<com.adobe.air.AIRWindowSurfaceView: int mBoundHeight> = $i3;
9 -    $i3 = virtualinvoke $r6.<android.view.Display: int getWidth()>();
10 +   $i3 = $r7.<android.graphics.Point: int x>;
11     $r0.<com.adobe.air.AIRWindowSurfaceView: int mBoundWidth> = $i3;
12 +  label04:
13 +   virtualinvoke $r6.<android.view.Display: void
       getSize(android.graphics.Point)>($r7);
14 +   goto label01;
```

Listing 2: An example of code diff mined from lineage *air.com.playsino.bingo.thanksgiving* (versions *AC1752* and *24C7BA*).

Overall, among the 500 app lineages, we are able to extract 6,043 code diffs that have involved in removing 360 deprecated APIs. Among the 360 APIs, 232 of them further target the removal of such APIs that have possible replacement messages mentioned in the Android documentation, suggesting that app developers are more likely to deal with such deprecated APIs that have recommended alternatives.

However, among the 232 APIs, because of class-level deprecation, we are only able to locate recommended replacement methods for 105 of them. If an API is deprecated at the class level, its replacement message is likely given at the class level as well and the recommended replacement is likely classes that do not provide explicit replacement messages for specific methods.

Among the 6,043 code diffs involved in removing deprecated APIs, 2,890 of them are relevant to the 105 APIs that have recommended alternatives highlighted in the Android documentation. We then go one step deeper to check to what extent app developers use the recommended alternative methods to replace the deprecated ones in order to remove them. Unfortunately, only 31 out of the 2,890 cases (less than 1%) have actually replaced the deprecated APIs with the recommended ones. This evidence suggests that app developers are not really (or at least are unlikely) following the recommendation of the official documentation to deal with deprecated APIs.

Furthermore, we compare the 31 cases (contributed by 13 distinct APIs), for which the deprecated APIs have been replaced with their recommended alternatives (under the same caller methods), with the top 10 removed deprecated APIs shown in Table 10. Surprisingly, none of the top 10 APIs has appeared in the

31 code diffs. This result once again confirms our previous observation that app developers are not likely replacing the deprecated APIs with their alternatives (at least not in the same caller methods) following the official documentation.

Towards understanding why deprecated APIs are removed while their recommended alternatives are not leveraged, we manually look into some samples. Our preliminary investigation finds that the aforementioned issue might be caused by the following reasons.

- R1: Replaced by library or wrapper code. Instead of directly using the recommended alternatives to replace the deprecated counterparts, to simplify the updates, app developers might directly leverage library methods or wrap the changes into independent methods. In this case, since the recommended alternatives are not explicitly presented in the code diff, our naive approach will not be able to spot that. Listing 3 presents such an example, where deprecated API *getOrientation()* is actually replaced by a method from an *amazon ad library*.
- R2: Alternative implementation. In addition to the recommended alternatives, there will be other means that app developers can achieve the same function (remove the deprecated APIs) while not using the recommended APIs. For example, as shown in Listing 3, deprecated API *getOrientation()* is not replaced by its recommended alternative method (which is *getRotation()*) but by a direct access to a field.
- R3: Function no longer supported. In many cases, the deprecated features are simply removed. For example, as also demonstrated in Listing 3, *getDescription()* is a deprecated API that is removed during an app update. However, without giving any alternative implementation about the deprecated feature (i.e., getDescription()), the update even changes the return value to always be *null*, indicating that app developers are no longer interested in this method.

```
1  //R1: Replaced by library or wrapper code.
2  //From pair FD2A08-D307D8
3  - $i0 = virtualinvoke $r5.<android.view.Display: int getOrientation()>();
4  + $i0 = staticinvoke <com.amazon.device.ads.AndroidTargetUtils: int
       getOrientation(android.view.Display)>($r5);
5
6  //R2: Alternative implementation
7  //From pair DD669C-E486E2
8  - $i0 = virtualinvoke $r2.<android.view.Display: int getOrientation()>();
9  + $i0 = $r2.<android.content.res.Configuration: int orientation>;
10
11 //R3: Function no longer support
12 //FACD24-1F4480
13 - public static java.lang.String
       getDescription(android.accessibilityservice.AccessibilityServiceInfo)
14 + public java.lang.String
       getDescription(android.accessibilityservice.AccessibilityServiceInfo)
15   {
16     ... ...
17 -   $r1 = virtualinvoke
        $r0.<android.accessibilityservice.AccessibilityServiceInfo:
        java.lang.String getDescription()>();
18 -   return $r1;
19 +   return null;
20   }
```

Listing 3: Sample code snippets demonstrating how are deprecated APIs removed without leveraging the recommended alternatives.

We would like to remind the readers that the approach we leveraged to check if a given deprecated API is replaced by its recommended alternative is quite naive, i.e., only the caller method of the removed deprecated API is checked. It is highly likely that the replacement may be put in other methods that are overlooked by our approach (cf. R1 in Listing 3 would be one of such examples). Indeed, among 2000 updates (i.e., app pairs) randomly sampled from the 500 app lineages, we found that around 70% of them have included the replacement in the updates with different locations (via a global analysis), which is much larger than that of same locations where deprecated APIs are accessed into. These replacement methods additionally included in the updated app version may not be the cases of replacing deprecated APIs but due to the introduction of new features, which is unfortunately non-trivial to confirm. Nonetheless, we believe this rate presents at least an upper bound of possible replacements to deprecated APIs. We encourage our fellow researchers to invent advanced techniques to improve the precision of identifying actual patches applied to update deprecated Android APIs.

Moreover, our straightforward approach only checks the syntactic similarity of the deprecated APIs and their recommended counterparts, semantic changes will, unfortunately, be missed. Let us take *getOrientation()* API in Listing 3 again as an example, except for the two samples (cf. R1 and R2), our preliminary investigation has also found actual fixes for this API, i.e., it is replaced by API *getRotation()*, as recommended by the Android documentation (cf. Listing 4). The fact that *getOrientation()* is fixed in different means (either follow the recommendation or not) suggests that the replacement of deprecated APIs is unlikely achieved through a systematic approach.

```
1 //From pair 855C79-FDF66C
2 $r4 = interfaceinvoke $r3.<android.view.WindowManager:
      android.view.Display getDefaultDisplay()>();
3 - $i0 = virtualinvoke $r4.<android.view.Display: int getOrientation()>();
4 + $i0 = virtualinvoke $r4.<android.view.Display: int getRotation()>();
5 return $i0;
6 }
```

Listing 4: Sample code snippets demonstrating how is deprecated API *getOrientation()* replaced by its recommended alternative (i.e., getRotation()).

Online Web Service. Based on these 6,043 code diffs, which have involved in removing 360 deprecated APIs, we further present to the community an online web service aiming at helping developers understand how other developers deal with deprecated APIs. As demonstrated in Fig. 22, the online web service takes as input a deprecated API and outputs a list of code diffs (13 diffs for API *android.net.wifi.WifiManager.startScan* as shown in the screenshot), for which app developers can leverage to understand quickly how the searched API is removed by other developers in practice. As of future work, we commit to harvesting more code diffs from a large set of app lineages.

RQ-6 Finding

When dealing with deprecated APIs during the evolution of Android apps, app developers are unlikely replacing the deprecated APIs (at the same place) with their alternatives recommended by the official Android documentation.

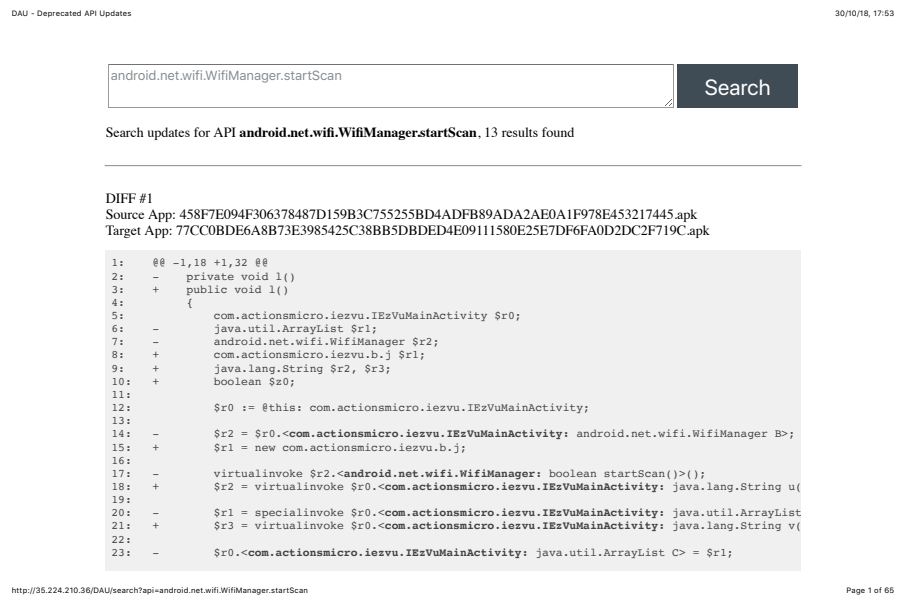

Fig. 22: A Sample Usage of the Online Web Service.

5 Discussion

This section discusses implications of this study and promising research directions that could be built on the characterization of Android APIs (cf. Section 5.1). We also enumerate some potential threats to validity in our findings (cf. Section 5.2).

5.1 Implications

The findings of this study raise a number of issues and opportunities for the research and practice communities.

$\Longrightarrow$ **Tool support for deprecating APIs**.

As unveiled by our investigations and reported in Section 4.1, deprecated APIs suffer from inconsistency issues in documentation and annotation. Most probably, API deprecation remains a manual process undertaken by framework developers. Given the consequences of inconsistency issues in practice for app developers, it is necessary that Android maintainers adopt specific tools to deal with API deprecation. Generally, it is important for not only the maintainers of Android framework base but also for the maintainers of any other repositories that need to deal with API deprecation to request tool support. It is non-trivial to devise a single tool that can fully solve the problem of API deprecating (Henkel and Diwan, 2005). Our community might need to split the problem into small tasks and implement dedicated tools to resolve them separately. Our research prototype,

namely CDA, is actually our first step towards providing such a tool set for helping repository maintainers better deal with API deprecation.

$\Longrightarrow$ **A *deprecate-replace-hide-remove* model**.

So far, the practice in dropping legacy APIs from the code base consists in applying the so-called *deprecate-replace-remove* model, where the legacy APIs are eventually removed after a certain period of time. This model appears to be suitable for most cases, but would still lead to crashes for some legacy client apps which still call into removed APIs. In order to avoid such unnecessary crashes, the Android framework base has introduced another means to deal with deprecated APIs. That is, instead of directly removing deprecated APIs, it first flags them as hidden APIs that can still live for a while in the framework side (i.e., available in the runtime virtual machine) but are no longer available in the client SDK. Thus, legacy apps, which still call into hidden APIs (removed from the SDK), can successfully run on updated devices. Meanwhile, new apps that are developed based on latest SDK would not face the problem of accessing "removed" APIs because those APIs are indeed removed from the developer's point of view. This scheme has already been shown to be effective for other APIs in the Android framework code base. Thus, we recommend that the community adopts a new process model for deprecating APIs, namely *deprecate-replace-hide-remove* model. We remind the readers that hidden APIs could be promoted to public APIs eventually (Hora et al., 2016), which however should not contradict the proposed *deprecate-replace-hide-remove* model as those hidden APIs will unlikely be originated from deprecated ones.

It is worth mentioning that app developers may be interested in using hidden APIs, e.g., a dedicated library has been provided to the community for app developers to access hidden APIs,[21] simply removing hidden APIs may result in problems to the apps developed by such developers. Nevertheless, as argued by (Li et al., 2016c), hidden APIs should be avoided in the first place. Therefore, while applying the *deprecate-replace-hide-remove* model, the usage of hidden APIs should also be regulated. Actually, starting in Android 9 (API level 28), the Android platform restricts the usage of certain hidden APIs[22]. If a given app attempts to access a hidden API that is restricted for the app's target API level, the Android system will throw an error. enforcing developers to avoid the usage of hidden APIs when developing new apps. However, to allow the execution of historical apps, the same hidden API might be still accessible if the app targets a lower API level.

$\Longrightarrow$ **Advanced fix mining for dealing with deprecated APIs**.

In this work, we attempt to automatically mine fixes of deprecated APIs from app lineages that contain the practical changes made by app developers. So far, our approach only look at the evolution of the caller method that has accessed into deprecated APIs. Given two subsequent app versions (a_x, a_{x_1}) in a lineage, deprecated APIs could indeed be removed from a_x by developers but their fixes may not necessarily be placed in the same place (i.e., the same caller method). As a result, our current fix mining approach may have overlooked a lot of true fixes. Therefore, we argue that there is a need to design advanced fix mining approaches towards learning the practical fixes from app developers.

[21] https://github.com/anggrayudi/android-hidden-api

[22] https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces

$\Longrightarrow$ **Recommendation System for fixing deprecated APIs.**.

Ideally, if we apply our fixing mining approach to a large set of app lineages, we would be able to collect a large set of code diffs demonstrating how are deprecated APIs fixed by developers in practice. The large set of code diffs can further be leveraged to implement a recommendation system for automatically recommending appropriate fix templates for helping developers fix deprecated APIs. Towards building a reliable recommendation system, one research challenge that is still needed to be addressed is to invent a new algorithm to rank the results, i.e., the most relevant code diff should be recommended first.

$\Longrightarrow$ **Automatic fix of deprecated APIs usage in apps**.

Our study in this work constructs a mapping between deprecated APIs and their replacement alternatives. An opportune research direction could be to invent an automated approach for fixing the usage of deprecated APIs across apps in the wild. This direction involves challenges beyond simple refactoring of API call sites: indeed, alternatives can be other API methods with different parameters (how to initialize arguments based on context variables?), suggested classes (how to infer object initialization and specific internal method calls?), or fields of existing objects (how to identify the right object, and use the appropriate field in replacement code?). Nevertheless, we believe that leveraging the mapping produced in this work and a large dataset of apps (with millions of code samples) can help systematically learn patterns for fixing the usage of deprecated APIs.

5.2 Threats to Validity

First, our investigation is conducted based on a subset of selected releases of the Android framework base, where the selected subset of releases may not be representative for the whole evolution of deprecated APIs and hence introduce threats into the external validity. Nevertheless, to alleviate this threat, we have considered all the possible API level releases.

Second, the representability of our approach could potentially be also impacted by the selection of app sets and lineages. For example, there is a chance that dormant apps may be selected to our dataset. Because dormant apps may come with low-quality and are not under active development, their developers may not react to the usage of deprecated APIs or care about the affection of deprecation, resulting in bias in our experimental results. Nonetheless, this threat is mitigated by performing random sampling from so far the largest and most up-to-date research dataset (a.k.a. AndroZoo) in our community. It is worth to mention that even with reputed apps, as disclosed by (Gao et al., 2019), their lineages may not be always good for supporting evolutionary studies such as mining usage patterns of cryptographic APIs. We hence encourage our fellow researchers in the community to working on this problem and inventing reliable means for supporting representative evolutionary studies in Android.

Third, our library-based investigation is based on a whitelist provided by (Li et al., 2016c), where certain libraries could be still missing, making our corresponding findings biased to some extent. Nevertheless, the whitelist we have leveraged contains over 1,000 libraries including at least the popular ones (e.g., all the popular libraries presented in Table 8 are included).

Fourth, the replacement messages of deprecated APIs are inferred via a heuristic-based approach, where the heuristics are summarised based on manual observation. Despite that, we have added more conservative rules to the heuristics, our approach is still subject to mistakes that may further introduce to both false positive and false negative results. The underline challenge prevents from properly inferring replacement messages is that Android developers do not follow a single means to provide replacement messages. Even for some parts of the APIs, where developers do follow similar patterns to introduce replacement messages, they also frequently make mistakes, making it also difficult to automatically infer replacement messages. In this work, we aim to ensure the correctness of the inferred replacement messages via manual verification, which however is non-trivial to achieve in practice. As of our future work, we plan to explore new possibilities to automatically and correctly infer replacement messages for deprecated Android APIs.

Fifth, the developer reactions study is conducted based on the *targetedSDK* version, which has been used by app developers to test against the functionality of the apps, resulting in a limited view of the use of deprecated APIs as ideally the full range of supported SDK versions should be considered. Nevertheless, our empirical findings should not be significantly impacted as the *targetedSDK* version generally represents the framework version the corresponding app is developed upon.

Sixth, the deprecated API update study is based on a naive assumption that app developers will replace deprecated APIs with their recommended alternatives at the same place where the deprecated APIs are accessed into (i.e., under the same caller method). Unfortunately, there is no guarantee that this assumption will be true in practice. Also, it is non-trivial to locate the code that updates the deprecated APIs outside of their caller methods, where comprehensive control-flow and data-flow analyses are expected. We hence left it for our future work.

Finally, our empirical investigations are performed purely on software artefacts (e.g., the source code and documentation of the Android framework base, or the bytecode of Android apps), the corresponding findings may only reflect the output of those artefacts and hence may not reflect the opinions of framework maintainers and app developers. To alleviate this, in our future work, we plan to contact both framework maintainers and app developers for a more comprehensive understanding on how are deprecated APIs treated in practice.

6 Related Work

Recent studies have explored the problem of deprecating APIs from various aspects. In this section, we discuss some of the most representative ones.

6.1 API Deprecation

As a common knowledge, deprecated APIs should follow the *deprecate-replace-remove* cycle where an API is first marked as deprecated and then replaced by a new API and eventually removed from the source code base (Zhou and Walker, 2016) (Dig and Johnson, 2005) (Kapur et al., 2010). However, many deprecated

APIs are not removed despite having remained as deprecated for years. For example, (Zhou and Walker, 2016) present a retrospective analysis of deprecated APIs and find that the traditional *deprecate-replace-remove* cycle is often not respected in open source Java frameworks and libraries. They also argue that, because of API deprecation, coding examples on the web can easily become outdated. Consequently, they present a prototype tool named *Deprecation Watcher* to automatically flag coding examples of deprecated APIs so that developers can be informed of such usages before spending time and energy into interpreting them. (Kapur et al., 2010) further reveal that deprecated entities do not always get removed eventually while removed entities are not always deprecated beforehand.

By analysing the Javadoc messages, source code, issue tracker and commit histories, Sawant et al. have observed 12 reasons that may trigger API producers to deprecate a feature (Sawant et al., 2018b). Furthermore, towards understanding developers' needs on API deprecation, the authors have conducted semi-structured interviews and surveys with Java producers and developers. Their experimental results disclose that the current deprecation mechanism in Java is not sufficient to address all the needs of Java developers (Sawant et al., 2018a). As one of the largest Java projects, the experience we obtained through mining the Android framework code base can be also useful to complement their work towards better understanding the developers' needs of deprecated API features.

For some Java systems on Maven Central Repository, deprecated APIs are even never removed, as discovered by (Raemaekers et al., 2014). Unfortunately, in their study, only *@Deprecated* annotation is considered, i.e., *@deprecated* Javadoc tag is ignored, which could have missed some deprecated APIs. As demonstrated in this work, it is quite common that these inconsistencies appear in Java source code repository such as the Android framework code base.

(Brito et al., 2016) argue that APIs should always be deprecated with clear replacement messages so that client systems can correspondingly update. However, based on their investigation, this philosophy is not always respected. Similarly, (Ko et al., 2014) investigate the relationship between API documentation quality and the resolved deprecated APIs. Their empirical investigation reveals that deprecated APIs with documented replacement messages are more likely to be updated comparing to such deprecated APIs that have no documentation indicating their alternatives.

(Espinha et al., 2014) provide a systematic and extensible study on the deprecation of web APIs. Their experimental results show that many web developers are not able to keep their app up-to-date even with a long deprecation time given. Taking Google Maps API version 2 as an example, Google gives three years for its developers to upgrade but turns out that three years are not enough. The authors then argue that three years are rather short but too long that leaves developers too relaxed to migrate their code. This interesting finding could also happen in Java-based systems including the Android framework code base. However, to explore this direction is out of the scope of this work, we therefore consider it as our future work.

Similar to the study of (Espinha et al., 2014), other researchers have also worked in this direction attempting to understand developer reactions to deprecated APIs (Hou and Yao, 2011) (Robbes et al., 2012) (Hora et al., 2015). For example, (Sawant et al., 2016) investigated more than 25,000 clients of five popular Java APIs on Github. They empirically found that client project maintainers

did not update their API versions as long as the execution is not broken. This finding is actually in line with ours where app developers are not motivated to update the target SDK version of their apps as long as the apps work fine in modern mobile devices.

6.2 API Evolution

(McDonnell et al., 2013) investigate the stability and adoption of Android APIs and find that Android APIs evolve fast and app developers do not follow the evolution momentum. For example, they disclose that around 28% of APIs used by Android apps are outdated where the median lagging time is 16 months. (Linares-Vásquez et al., 2014) further explore the relationship between fault- and change-prone APIs and the success of Android apps and empirically demonstrates that there is a negative impact between these two parts (Bavota et al., 2015). Furthermore, they also empirically show that change-prone Android APIs are more likely discussed on social media such as Stack Overflow (Linares-Vásquez et al., 2014).

(Li et al., 2016c) explore the evolution of inaccessible Android APIs, where both internal and hidden APIs are considered. Like our approach, they also investigate the inaccessible APIs based on the historical changes of the Android framework code base. They have taken into account 17 prominent releases and reveal that inaccessible APIs are commonly implemented in the Android framework. In this work, we find another reason, which is yet not disclosed by their approach, that certain deprecated APIs are eventually marked as hidden. This modification is quite intelligent as from app developer's point of view those deprecated APIs have been removed from the SDK while from the framework's point of view those deprecated APIs are still retained to avoid potential compatibility issues.

In addition to Android framework code base, several approaches are also proposed to investigate the evolution of general framework code (Dagenais and Robillard, 2011) (Wu et al., 2010) (Meng et al., 2012) (Hou and Yao, 2011) (Dig and Johnson, 2006) or library code (Cossette and Walker, 2012). For example, (Hou and Yao, 2011) are interested in exploring the Intent behind API evolution, so as to counter the negative impacts of API evolution. (Dagenais and Robillard, 2011) present a client-server tool called SemDiff that automatically recommends adaptations such as replacing no longer existed methods to client programs by mining the evolution of framework changes. Similarly, (Wu et al., 2010) introduce AURA, a hybrid approach that integrates call dependency analysis with text similarity comparison together, to automatically identify change rules to further benefit client programs to keep their code up-to-date. (Meng et al., 2012) present a novel approach named HiMa, which performs pairwise comparisons for each consecutive revisions recorded in the evolutionary history and aggregates revision-level rules to construct framework-evolution rules. Although HiMa takes more computing powers than AURA, it achieves higher precision and recall in most circumstances.

Finally, our fellow researchers are also interested in automatically migrating client code to cope with evolving APIs (Dig et al., 2008) (Štrobl and Troníček, 2013) (Bogart et al., 2016) (Brito et al., 2018a). For example, (Chow and Notkin, 1996) propose a semi-automated approach for updating client projects in response to library changes. Their approach presents a toolset that relies on changed functions annotated by library maintainers to automatically update client projects.

The authors further introduce the so-called twinning technique for allowing programmers to specify a class of program changes (i.e., a mapping) without modifying the target program directly (Nita and Notkin, 2010). This mapping can then be leveraged to transition a program from using one API to using an alternative API. Instead of manually annotating the changes of given libraries, (Henkel and Diwan, 2005) presents a prototype tool called Catchup!, which aims at capturing and replaying refactoring actions within an integrated development environment. (Xing and Stroulia, 2007) attempt to automatically recognise the API changes and proposes plausible replacements to the "obsolete" API based on working examples of the framework code base. All of these approaches have proposed promising techniques to handle deprecation in the evolution of software frameworks. Specifically, we believe these approaches can be also applied to resolve the deprecation problem of Android APIs, i.e., to automatically update deprecated APIs in Android apps.

7 Conclusion

In this work, we have conducted an exploratory study of deprecated Android APIs. In particular, we have built a prototype research tool called CDA and applied it to different revisions (i.e., releases or tags) of the Android framework code base to investigate all the deprecated APIs (how are they annotated and documented? or how are they cleaned up or survived during the evolution of the framework base?) and infer the mapping with their potential replacement alternatives. Finally, we explore a set of real-world Android apps attempting to understand the reaction of app developers to deprecated Android APIs.

Our experimental investigation eventually finds that (1) Deprecated Android APIs are not always consistently annotated and documented, which can have severe consequences in app development and user experience; (2) The Android framework code base is regularly cleaned-up from deprecated APIs, often in a short period of time; (3) In general, over half of the deprecated APIs in the Android framework are commented to provide alternatives, although they will be rarely updated. (4) In practice, most usage sites of deprecated APIs in app code are located in popular libraries, although, library developers are more likely to update deprecated APIs than app developers. (5) During the evolution of Android apps, deprecated APIs are likely retained rather than removed from the app code. (6) For the cases app developers do remove deprecated APIs from the app, they are unlikely replacing the deprecated APIs with their alternatives recommended by the official documentation, at least not directly at the same place (e.g., under the same caller method).

Acknowledgements

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments to the conference version of this extension.

References

Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.

Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E Hassan, Juergen Dingel, and James R Cordy. Analyzing a decade of linux system calls. *Empirical Software Engineering*, pages 1–33, 2017.

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. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015.

Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2016.

Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why and how java developers break apis. *arXiv preprint arXiv:1801.05198*, 2018a.

Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 360–369. IEEE, 2016.

Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. On the use of replacement messages in api deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321, 2018b.

Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *icsm*, volume 96, page 359, 1996.

Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 134–145. IEEE, 2015.

Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 55. ACM, 2012.

Barthélémy Dagenais and Martin P Robillard. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):19, 2011.

Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.

Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 389–398. IEEE, 2005.

Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of the 29th international conference on Software Engineering*, pages 427–436. IEEE Computer Society, 2007.

Danny Dig, Stas Negara, Ralph Johnson, and Vibhu Mohindra. Reba: refactoringaware binary adaptation of evolving libraries. In *In ICSE08: Proceedings of the 30th International Conference on Software Engineering*. Citeseer, 2008.

Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 84–93. IEEE, 2014.

Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. On vulnerability evolution in android apps. In *The 40th International Conference on Software Engineering, Poster Track (ICSE 2018)*, 2018.

Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. Negative results on mining crypto-api usage rules in android apps. In *The 16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.

Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 236–247. IEEE, 2015.

Johannes Henkel and Amer Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 274–283. IEEE, 2005.

André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 251–260. IEEE, 2015.

Andre Hora, Marco Tulio Valente, Romain Robbes, and Nicolas Anquetil. When should internal interfaces be promoted to public? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 278–289. ACM, 2016.

Daqing Hou and Xiaojia Yao. Exploring the intent behind api evolution: A case study. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 131–140. IEEE, 2011.

Puneet Kapur, Brad Cossette, and Robert J Walker. *Refactoring references for library migration*, volume 45. ACM, 2010.

Deokyoon Ko, Kyeongwook Ma, Sooyong Park, Suntae Kim, Dongsun Kim, and Yves Le Traon. Api document quality for resolving deprecated apis. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 2, pages 27–30. IEEE, 2014.

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 *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

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), 2016a.

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)*, 2016b.

Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016c.

Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017a.

Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017b.

Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018a.

Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018b.

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.

Tyler McDonnell, Baishakhi 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.

Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 353–363. IEEE, 2012.

Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

Marius Nita and David Notkin. Using twinning to adapt programs to alternative apis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 205–214. IEEE, 2010.

Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software*, 137:143–162, 2018.

Jeff H Perkins. Automatically generating refactorings to support api evolution. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 111–114. ACM, 2005.

Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and*

Manipulation, pages 215–224. IEEE Computer Society, 2014.

Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular java apis. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 400–410. IEEE, 2016.

Anand Ashok Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 561–571. IEEE, 2018a.

Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. Why are features deprecated? an investigation into the motivation behind deprecation. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–24. IEEE, 2018b.

Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018c.

Roman Štrobl and Zdeněk Troníček. Migration from deprecated api in java. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 85–86. ACM, 2013.

Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82. ACM, 2015.

Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *The 2018 Internet Measurement Conference (IMC 2018)*, 2018.

Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 325–334. ACM, 2010.

Zhenchang Xing and Eleni Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. Characterizing malicious android apps by mining topic-specific data flow signatures. *Information and Software Technology*, 2017.

Jing Zhou and Robert J Walker. Api deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 266–277. ACM, 2016.