Characterising Deprecated Android APIs

Li Li\(^1\), Jun Gao\(^2\), Tegawendé F. Bissyandé\(^2\), Lei Ma\(^3\), Xin Xia\(^1\), Jacques Klein\(^2\)

\(^1\) Faculty of Information Technology, Monash University, Australia
\(^2\) Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
\(^3\) School of Computer Science and Technology, Harbin Institute of Technology, China

\{li.li,xin.xia\}@monash.edu,\{jun.gao,tegawende.bissyande,jacques.klein\}@uni.lu, malei@hit.edu.cn

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 exploratory 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, as well as developer reactions to API deprecation. Experimental results reveal several findings that further provide promising insights for future research directions 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.

ACM Reference format:
Li Li\(^1\), Jun Gao\(^2\), Tegawendé F. Bissyandé\(^2\), Lei Ma\(^3\), Xin Xia\(^1\), Jacques Klein\(^2\).
https://doi.org/10.1145/3196398.3196419

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. [1] 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 [2].

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 \texttt{@Deprecation} Java annotation, or by inserting \texttt{@deprecation} 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 [3]. 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 [4, 5], the deprecate-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 [6, 7]. Others have studied developer reactions to deprecated APIs [4, 5, 8–11]. There have been also various works on automatically migrating client code in response to broken APIs [12–19]. 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 exploratory 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 70% 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. The accessing delay of common libraries however is generally shorter than that of app code, and library developers are more likely to update deprecated APIs than app developers.

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 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.
- 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.

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.

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 8 versions providing progressively 26 API levels. This SDK comes with an online portal that tracks all documentation written by Android maintainers to help developers correctly use the provided APIs.

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 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.
- 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

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 [20]. 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

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

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).

```java
package java.android.net;

/**
 * Deprecated All APIs accepting a network type are deprecated. There should be no need to validate a network type.
 */

@Deprecated
public static boolean isNetworkTypeValid(int networkType)
{
    return MIN_NETWORK_TYPE <= networkType &&
    networkType <= MAX_NETWORK_TYPE;
}

첫 번째 API: isNetworkTypeValid()

```java
package java.android.net;

/**
 * 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.
 */

@Deprecated
@RequiresPermission(android.Manifest.permission.BROADCAST_STICKY)
public abstract void removeStickyBroadcast(@RequiresPermission Intent intent);

두 번째 API: removeStickyBroadcast()

```

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.
- A set of 10,000 apps that are randomly selected from AndroZoo [21]. We sample 5,000 apps from the official Google Play market (GPlay) apps and 5,000 apps from third-party markets (NGPlay).

The Android platform code, hosted in Github since October 2008, is actually a mirror of the Google source code repository maintained by Google. It has since been forked 5,000 times, and has seen the contributions of over 600 developers, while being watched for changes by almost 900 developers. The 109 git development branches have integrated changes from 323,059 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 [22–24].

Over 450 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. Eventually, as illustrated in Table 1, we are able to consider 20 releases (associated to 20 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.

<table>
<thead>
<tr>
<th>API Level</th>
<th>Code Name</th>
<th>Selected Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Oreo</td>
<td>android-8.0.0_r9</td>
</tr>
<tr>
<td>25</td>
<td>Nougat</td>
<td>android-7.1.0_r7</td>
</tr>
<tr>
<td>24</td>
<td>Nougat</td>
<td>android-7.0.0_r7</td>
</tr>
<tr>
<td>23</td>
<td>Marshmallow</td>
<td>android-6.0.1_r9</td>
</tr>
<tr>
<td>22</td>
<td>Lollipop</td>
<td>android-5.1.1_r9</td>
</tr>
<tr>
<td>21</td>
<td>Lollipop</td>
<td>android-5.0.2_r3</td>
</tr>
<tr>
<td>19</td>
<td>KitKat</td>
<td>android-4.4w_r1</td>
</tr>
<tr>
<td>18</td>
<td>Jelly Bean</td>
<td>android-4.3_r3.1</td>
</tr>
<tr>
<td>17</td>
<td>Jelly Bean</td>
<td>android-4.2_r1</td>
</tr>
<tr>
<td>16</td>
<td>Jelly Bean</td>
<td>android-4.1_r2.1</td>
</tr>
<tr>
<td>15</td>
<td>Ice Cream Sandwich</td>
<td>android-4.0_r2.1</td>
</tr>
<tr>
<td>14</td>
<td>Ice Cream Sandwich</td>
<td>android-4.0_r2.1</td>
</tr>
<tr>
<td>13</td>
<td>Honeycomb</td>
<td>android-3.2_r1</td>
</tr>
<tr>
<td>10</td>
<td>Gingerbread</td>
<td>android-2.3_r1</td>
</tr>
<tr>
<td>9</td>
<td>Gingerbread</td>
<td>android-2.3_r2</td>
</tr>
<tr>
<td>8</td>
<td>Froyo</td>
<td>android-2.2.3_r2</td>
</tr>
<tr>
<td>7</td>
<td>Eclair</td>
<td>android-2.1_r2.1</td>
</tr>
<tr>
<td>6</td>
<td>Eclair</td>
<td>android-2.0.1_r1</td>
</tr>
<tr>
<td>5</td>
<td>Eclair</td>
<td>android-2.0_r1</td>
</tr>
<tr>
<td>4</td>
<td>Donut</td>
<td>android-1.6_r2</td>
</tr>
</tbody>
</table>

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 (hereinafter referred as NGPlay) such as AppChina. These apps are randomly selected from the AndroZoo app repository, which contains over 5 million Android apps and is known to be so far the largest app set publicly available.

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

By using gshuf/ head -5000 command.
to our community. Apps from this dataset have been previously leveraged for a variety of research studies [25–28].

![Figure 2: Distribution of randomly selected apps based on their assembled date (i.e., dex date).](image1)

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 compiled 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.

![Figure 3: Distribution of randomly selected apps based on their size (in MB).](image2)

### 3.2 CDA

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

CDA first parses all Java files in a given release of the Android framework code repository and builds a mapping between Java methods and their documentation (cf. line 6). Then, for each method, 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. Thus, in a first phase, CDA can pinpoint inconsistency cases where a deprecated API is documented but not annotated (lines 13-15) or is annotated but not documented (lines 17-19). 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.

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 the different metrics (i.e., the number of files, classes, lines of code, and API methods). Eventually, between level 4 and level 26 (the two extreme API levels in our study), the framework code has substantially grown: the number of classes has almost doubled, while the number of code lines has tripled; the phenomenon is even more acute in methods which have grown 6-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 drastically dropped. Indeed, as shown in Table 2, the total number of APIs in level 21 has almost doubled comparing to that of level 19 while the number of deprecated APIs are more or less kept the same.

### 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?
Characterising Deprecated Android APIs

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

<table>
<thead>
<tr>
<th>API Level</th>
<th># Java Classes</th>
<th># LoC</th>
<th># Total APIs</th>
<th># Public APIs</th>
<th># Static APIs</th>
<th># Deprecated APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>742</td>
<td>3244981</td>
<td>4478</td>
<td>3677</td>
<td>610</td>
<td>133</td>
</tr>
<tr>
<td>25</td>
<td>635</td>
<td>2927464</td>
<td>3972</td>
<td>3341</td>
<td>497</td>
<td>119</td>
</tr>
<tr>
<td>24</td>
<td>623</td>
<td>2864293</td>
<td>3910</td>
<td>3299</td>
<td>491</td>
<td>119</td>
</tr>
<tr>
<td>23</td>
<td>557</td>
<td>2538626</td>
<td>3367</td>
<td>2822</td>
<td>429</td>
<td>101</td>
</tr>
<tr>
<td>22</td>
<td>504</td>
<td>2376430</td>
<td>2993</td>
<td>2460</td>
<td>414</td>
<td>83</td>
</tr>
<tr>
<td>21</td>
<td>490</td>
<td>2333200</td>
<td>2920</td>
<td>2392</td>
<td>413</td>
<td>83</td>
</tr>
<tr>
<td>19</td>
<td>439</td>
<td>1381169</td>
<td>2864</td>
<td>2318</td>
<td>412</td>
<td>98</td>
</tr>
<tr>
<td>18</td>
<td>425</td>
<td>1271452</td>
<td>2765</td>
<td>2197</td>
<td>395</td>
<td>90</td>
</tr>
<tr>
<td>17</td>
<td>419</td>
<td>1248085</td>
<td>2624</td>
<td>2022</td>
<td>387</td>
<td>78</td>
</tr>
<tr>
<td>16</td>
<td>425</td>
<td>1265976</td>
<td>2668</td>
<td>2059</td>
<td>379</td>
<td>73</td>
</tr>
<tr>
<td>15</td>
<td>398</td>
<td>1151084</td>
<td>2466</td>
<td>1862</td>
<td>348</td>
<td>48</td>
</tr>
<tr>
<td>14</td>
<td>397</td>
<td>1137869</td>
<td>2466</td>
<td>1846</td>
<td>346</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>380</td>
<td>1028975</td>
<td>2433</td>
<td>1787</td>
<td>364</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>313</td>
<td>872561</td>
<td>1897</td>
<td>1340</td>
<td>324</td>
<td>58</td>
</tr>
<tr>
<td>9</td>
<td>303</td>
<td>849373</td>
<td>1858</td>
<td>1301</td>
<td>316</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>428</td>
<td>896503</td>
<td>3250</td>
<td>2444</td>
<td>425</td>
<td>68</td>
</tr>
<tr>
<td>7</td>
<td>428</td>
<td>841184</td>
<td>3129</td>
<td>2339</td>
<td>414</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>439</td>
<td>831461</td>
<td>3147</td>
<td>2326</td>
<td>412</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>439</td>
<td>837932</td>
<td>3146</td>
<td>2326</td>
<td>412</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>389</td>
<td>774426</td>
<td>2980</td>
<td>2204</td>
<td>360</td>
<td>54</td>
</tr>
</tbody>
</table>

Figure 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.

- **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?

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

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, users 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.

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

Figure 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 small set of cases where the documentation is not consistent with deprecation annotation presence/absence. Table 3 summarises statistics of cases found in the various framework releases. We note that deprecated APIs are generally well documented as such: Annotated-Not-Documented cases of inconsistencies are limited or nonexistent in the releases. In contrast, there are several cases where an API documented as deprecated is not annotated as such: until API level 15, we could find less than 10 such cases per framework release; later releases contain several more inconsistency cases (up to 8 times more inconsistencies between API level 13 and API level 23). 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.

Finally, 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 latest version of the Android framework base (i.e., master branch). 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\(^1\).

\(^1\)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.
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. 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). Over half of the updates 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.

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

<table>
<thead>
<tr>
<th>Update</th>
<th>Additional Removal</th>
<th>Update</th>
<th>Additional Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4 → L5</td>
<td>13 1</td>
<td>L16 → L17</td>
<td>2 8</td>
</tr>
<tr>
<td>L5 → L6</td>
<td>0 0</td>
<td>L17 → L18</td>
<td>11 4</td>
</tr>
<tr>
<td>L6 → L7</td>
<td>0 3</td>
<td>L18 → L19</td>
<td>8 0</td>
</tr>
<tr>
<td>L7 → L8</td>
<td>5 0</td>
<td>L19 → L21</td>
<td>5 20</td>
</tr>
<tr>
<td>L8 → L9</td>
<td>0 10</td>
<td>L21 → L22</td>
<td>0 0</td>
</tr>
<tr>
<td>L9 → L10</td>
<td>0 0</td>
<td>L22 → L23</td>
<td>18 1</td>
</tr>
<tr>
<td>L10 → L11</td>
<td>0 1</td>
<td>L23 → L24</td>
<td>16 0</td>
</tr>
<tr>
<td>L11 → L12</td>
<td>0 4</td>
<td>L24 → L25</td>
<td>0 0</td>
</tr>
<tr>
<td>L12 → L13</td>
<td>0 0</td>
<td>L25 → L26</td>
<td>27 14</td>
</tr>
</tbody>
</table>

We further go one step deeper to check how deprecated Android APIs are removed from the framework code base. Our investigation reveals that 25 deprecated APIs are not “actually” physically removed from the framework but are only tagged as hidden for app developers. Nevertheless, in this work, we still consider such deprecated APIs as removed. As discussed by Li et al. [22], hidden APIs are also 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.

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”\(^{10}\). It can be observed from the results shown in Figure 6, that most deprecated APIs are not removed immediately in the next release (i.e., generation ≥ 2) and over 90% of deprecated APIs have survived beyond one generation. Such a grace period is understandable, since developers must be given time to take actions. Yet, 6.7% of deprecated APIs are removed after one update. Although this rate is low, we are still surprised that this situation does happen during the evolution of the Android framework code base. Because of the limited time window, app developers may not yet be informed and hence may still leverage those deprecated APIs, resulting in immediate crashes on devices running next framework versions.

Figure 7 presents the violin plot on the life expectancy distribution of deprecated Android APIs. The median number of generations a deprecated API survives in the code base is 5 (mean = 6.2). Given the fact that the Android framework code base evolves at a fast pace (a generation occurs every 3 months[1]), app developers need to react quickly on replacing deprecated APIs in their client code before they become inaccessible in updated devices.

\(^{10}\)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).
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 others?). However, in practice, there is evidence that API elements are usually deprecated without such messages [4–6]: 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 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).

Figure 8 illustrates the distribution of deprecated APIs with/without replacement messages for the considered API level releases. In each release of the framework, a median percentage of 69.35% deprecated APIs have been explicitly documented with replacement messages (mean percentage is 70.05%). The latest release (i.e., level 26) has replacement messages for 62 APIs (i.e., 68.1% of total deprecated methods) in line with average metrics. However, comparing to the study of Brito et al. [6], who has investigated a large-scale study on 661 real-world Java systems and shown that the average replacement rate of deprecated APIs is 64%, the replacement rate of Android framework code is slightly higher, demonstrating that the deprecation-then-updating quality of Android framework (at least for deprecation) is generally above the average of normal Java programs.

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: (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?

We find that: (1) No replacement message will be added to such deprecated APIs that initially have no replacement message; and (2) seldom, an existing replacement message will be updated: we identified only three API cases (cf. Table 6) 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.

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) [29], 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 statements of the analyzed app to check if some used APIs have been deprecated in releases prior to the declared targeted SDK version.

Among our randomly sampled set of 10,000 apps, CDA highlights that 37.87% apps are making use of deprecated APIs. Among the flagged 3,787 apps, the GPlay subset contributes with 2,897 apps while NGPlay contributes with 1,870 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. 9, 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\(^{12}\), where the resulting p-value confirms that there is a significant difference between Google Play and alternative markets apps at a significance level\(^{13}\) of 0.001.

\(^{12}\)We have appended 2,007 (2,897–890) zero to third-party markets (i.e., NGPlay) to balance the number of elements.

\(^{13}\)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.
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\textsuperscript{14}. Table 7 highlights the top five caller packages that have invoked deprecated APIs in Google Play and Third-party market apps, respectively. If we exclude common libraries from consideration, the number of apps leveraging deprecated APIs reduces to 374 and 127 respectively for Google Play and third-party market apps. This evidence 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.

Table 7: The top five packages calling into deprecated Android APIs, which account for 90% and 74% of total deprecation usages in Google Play and Third-party Markets, respectively.

<table>
<thead>
<tr>
<th>Deprecated API</th>
<th>Replacement Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.database.sqlite.SQLiteClosable: void onAllReferencesReleasedFromContainer()</td>
<td>releaseReferenceFromContainer()</td>
</tr>
<tr>
<td>android.webkit.WebSettings: void setDefaultZoom(ZoomDensity)</td>
<td>ZoomDensity#MEDIUM onReceive</td>
</tr>
<tr>
<td>android.app.admin.DeviceAdminReceiver: void onReadyForUserInitialization(Context,Intent)</td>
<td>sendStickyBroadcast</td>
</tr>
<tr>
<td>android.content.Context: void removeStickyBroadcast(Intent)</td>
<td>getTextZoom()</td>
</tr>
<tr>
<td>android.database.Cursor: void deactivate() #query</td>
<td></td>
</tr>
</tbody>
</table>

\textit{deprecationLevel}. This delay represents the number of generations where app developers are still able to call deprecated APIs. The delay between the thousands deprecated APIs called by the 3,787 apps range from 1 to 18 with 5 and 4.9 generations as median and mean, respectively. Fig. 10 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 both Google Play and third-party market apps. This difference is also further confirmed by a MWW test.

Table 6: The updated three replacement messages.

<table>
<thead>
<tr>
<th>Replacement Message (original)</th>
<th>Replacement Message (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#SaCertificate(String, String, Date, Date)</td>
<td>#SaCertificate(X509Certificate)</td>
</tr>
<tr>
<td>#setTextZoom(int)</td>
<td>#setTextZoom</td>
</tr>
<tr>
<td>#getTextZoom() #getTextZoom</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Distribution of the number of deprecated APIs utilised per app.

Figure 10: Distribution of delays between Google Play and third-party market apps. Suffixes _APP and _LIB indicate that the caller of deprecated APIs are from the app code and third-party library code, respectively.

**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. The accessing delay of common libraries however is generally shorter than that of app code, and library developers are more likely to update deprecated APIs than app developers.

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.

⇒ 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. Our research prototype, namely CDA, is actually our first step towards providing such a general tool for helping repository maintainers better deal with API deprecation.

⇒ 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 [31], which however should not contradict the proposed deprecate-replace-hide-remove model as those hidden APIs will unlikely be originated from deprecated ones.

⇒ 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.

⇒ Evolution study on apps dealing with deprecation.

Although we found in our study that most 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, we can investigate how developers react to API alternatives: do developers follow maintainer recommendations? what has impacted API deprecation on app code maintenance? etc. Such a study will complete the view on API deprecation in the Android framework.

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. 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.

Third, our library-based investigation is based on a whitelist provided by Li et al. [22], 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 7 are included).

Fourth, 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.

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 [32–34]. However, many deprecated APIs are not removed despite having remained as deprecated for years. For example, Zhou et al. [32] 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. [34] further reveal that deprecated entities do not always get removed eventually while removed entities are not always deprecated before hand.

For some Java systems on Maven Central Repository, deprecated APIs are even never removed, as discovered by Raemaekers et al. [35]. Unfortunately, in their study, only @Deprecated annotation is considered, i.e., @deprecation 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 base such as the Android framework code base.

Brito et al. [6] 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. [7] 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. [8] 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 be also happened 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.

6.2 API Evolution

McDonnell et al. [1] 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-Vasquez et al. [10] 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 [36]. Furthermore, they also empirically show that change-prone Android APIs are more likely discussed on social media such as Stack Overflow [10].

Li et al. [22] 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 [37–39]. Dagenais and Robillard [37] 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. [38] 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. [39] 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.

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, Android framework ensure that deprecated APIs are commented to provide alternatives, although this documentation is 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.
REFERENCES


