Identifying and Characterizing Silently-Evolved Methods in the Android API

Pei Liu*, Li Li*, Yichun Yan†, Mattia Fazzini†, John Grundy*
*Monash University, Melbourne, Australia
{pei.liu,li.li,john.grundy}@monash.edu
†University of Minnesota, Minneapolis, USA
{yan00104,mfazzini}@umn.edu

***Abstract*—With over 500,000 commits and more than 700 contributors, the Android platform is undoubtedly one of the largest industrial-scale software projects. This project provides the Android API, and developers heavily rely on this API to develop their Android apps. Unfortunately, because the Android platform and its API evolve at an extremely rapid pace, app developers need to continually monitor API changes to avoid compatibility issues in their apps (i.e., issues that prevent apps from working as expected when running on newer versions of the API). Despite a large number of studies on compatibility issues in the Android API, the research community has not yet investigated issues related to silently-evolved methods (SEMs). These methods are functions whose behavior might have changed but the corresponding documentation did not change accordingly. Because app developers rely on the provided documentation to evolve their apps, changes to methods that are not suitably documented may lead to unexpected failures in the apps using these methods.**

To shed light on this phenomenon, we conducted a large-scale empirical study in which we identified and characterized SEMs across ten versions of the Android API. In the study, we identified SEMs, characterized the nature of the changes, and analyzed the impact of SEMs on a set of 1,000 real-world Android apps. Our experimental results show that SEMs do exist in the Android API, and that 957 of the apps we considered use at least one SEM. Based on these results, we argue that the Android platform developers should take actions to avoid introducing SEMs, especially those involving semantic changes. This situation highlights the need for automated techniques and tools to help Android practitioners in this task.

I. Introduction

Mobile applications (or simply apps) are becoming increasingly prevalent in our lives. For instance, in 2017, US-based users spent an average of two hours and 25 minutes per day using mobile apps. This time accounts for more than 80% of the total time spent by the users on their mobile devices [1]. Android apps, which run on the Android operating system (OS), are the most widely used type of mobile apps and account for over 70% of the mobile OS market share [2]. Android apps are not only extremely popular, but their number is also growing at a staggering speed, with about 35,000 new apps released on Google Play every month [3].

One common trait of Android apps is that they rely heavily on the underlying Android OS, as the OS offers access to a large number of popular and essential app services. Apps can access these services using the application programming interface (API) of the OS. On average, 25% of all methods and field references in the apps are uses of the Android API [4]. This characteristic facilitates app development [5], but it also creates a tight coupling between the apps and the version of the API used by the apps.

Unfortunately, the Android OS and its API evolve rapidly [4], [6]–[10], and when a new version of the API is released, app developers need to carefully understand the changes to the API so that they can suitably adapt their apps to also run on the new version of the API. To help app developers in this task, Android provides a curated documentation that app developers can access using the platform website [11]. This documentation is based on the comments associated with the source code of the API, and the comments are created and maintained by the API developers.

When API developers are working on a new version of the API, they not only should create comments that describe newly added API components but they should also update existing comments to report changes in existing API components (e.g., [12]). In the latter task, developers should document both syntactic and behavioral changes introduced in the new version of the API. Although it is important to document both types of changes, it is imperative to document the second class of changes as app developers would otherwise not easily know how to update their apps, and users might experience field failures due to compatibility issues.

Although related work investigated a number of aspects associated with the evolution of the Android API [4], [7], [13]–[16], to the best of our knowledge, no work has systematically analyzed whether there are issues in the documentation of the behavioral changes in the platform API. Additionally, no study identified the extent to which these issues might affect Android apps. A study on these topics would provide insights for building automated techniques that detect the issues and highlight mitigation strategies against the issues.

To fill this gap, we present an extensive empirical study that identifies and characterizes *silently-evolved methods* (SEMs) in the Android API. A SEM is a method whose implementation is different across two subsequent versions of the Android API, while the method's documentation (i.e., the method's comment) is not changed. In the study, we (i) identify SEMs across different versions of the Android API, (ii) report the characteristics of these methods, and (iii) analyze the impact that SEMs might have on real-world apps. Specifically, we

analyzed method updates across ten API releases and manually classified updates to methods whose documentation did not change. In the study, we identified 4,769 SEMs, which include 2,271 publicly-accessible methods. After manually analyzing a statistically significant sample containing 562 SEMs, we found that 363 of the methods include semantic changes. Furthermore, we also analyzed the use of SEMs in a sample of 1,000 real-world Android apps and observed that 957 of these apps use at least one SEM. Interestingly, a number of such usages have been manually mitigated by app developers through API version checks[1], indicating that SEMs could indeed introduce compatibility issues in Android apps. Overall, we believe that our results highlight that Android developers do not always thoroughly document semantic changes in the platform API and that these changes can extensively affect real-world Android apps.

In summary, the main contributions of this paper are:

- A characterization of SEMs across the ten most used versions (at the time we started our study) of the Android API. We analyzed method updates across ten versions of the Android API and manually confirmed that 363 methods out of a sample of 562 publicly-accessible methods contain semantic changes. We also characterized the nature of SEMs and found that developers might have accidentally introduced the majority of them as most of them are evolved only once.
- A study of how SEMs impact Android apps. We investigated whether and how Android apps use SEMs in a sample of 1,000 real-world Android apps. We found that SEMs are commonly used by Android apps, and such usages could indeed lead to compatibility issues as app developers have added checks to prevent the execution of SEMs on certain versions of the Android API.
- A tool to identify SEMs and the experimental data containing the findings of our study. To identify and characterize SEMs, we designed a technique and implemented the approach in a prototype tool called ANDROSEA. ANDROSEA identifies methods across two versions of the Android API that have same signature, same method comment, but different method body. The tool and the experimental data are publicly available at https://github.com/MobileSE/AndroSea.

The remainder of this paper is organized as follows. Section II defines relevant terminology. Section III presents our study methodology. Section IV details the results of the study. We discuss implications for researchers and practitioners in Section V. Section VI outlines related work. Finally, Section VII provides concluding remarks.

[1]Version checks are recommended by Google to tackle API-induced compatibility issues. A typical check conforms to the following structure: if `SDK_INT` $< n$, call a method of the older API, otherwise, do something else. `SDK_INT` is the short version for `android.os.Build.SDK_INT` and this value, at runtime, provides the version of the API on which a certain app is running. This check ensures that the method of the API, which may introduce compatibility issues in the app when the app is running on newer versions of the API, will only be invoked if the app is running on API versions that are older than the one where the change was introduced.

II. TERMINOLOGY AND MOTIVATION

This sections introduces the relevant terminology we will use in the rest of this paper. Consider two API versions (or levels): *old API* = $[m_1, ..., m_k]$ and *new API* = $[m'_1, ..., m'_l]$. A method m_i from the old API is a *silently-evolved method* (SEM) if there is a method m'_i in the new API that shares the same signature and the same comment with m_i, but has a different method body with respect to m_i. Among SEMs, we define those methods that are publicly accessible (i.e., methods that are declared as public) as *publicly-accessible silently-evolved methods* (PASEMs).

Listing 1 provides an example of a PASEM, which shows the evolution of the method `getSqlStatementType` between API level 27 and 28. In the example, the two versions of the method have the same signature (line 7) and the same method comment (lines 1-6), but different method body, as Android developers added new statements to the method in the API level 28 (lines starting with +). In both API versions, the method returns the type of the SQL statement provided as input. However, from API level 28, part of the method behavior is changed. In fact, instead of returning `STATEMENT_ABORT` for any SQL rollback statement as defined in API level 27, the method returns a different value (i.e., `STATEMENT_OTHER`) if the SQL statement aims at rollbacking to a savepoint [17].

```
1  /**
2   * Returns one of the following which represent the
        type of the given SQL statement.
3   * ...
4   * @param sql the SQL statement whose type is
        returned by this method
5   * @return one of the values listed above
6   */
7  public static int getSqlStatementType(String sql) {
8   String prefixSql = sql.substring(0,
        3).toUpperCase(Locale.ROOT);
9   else if (prefixSql.equals("ROL")) {
10 + boolean isRollbackToSavepoint =
        sql.toUpperCase(Locale.ROOT).contains(" TO ");
11 + if (isRollbackToSavepoint) {
12 +   ...
13 +   return STATEMENT_OTHER;
14 + }
15    return STATEMENT_ABORT;
16  }
17  return STATEMENT_OTHER;
18 }}
```

Listing 1. Code snippet extracted by comparing the method `getSqlStatementType` between Android API level 27 and 28.

Unfortunately, due to this behavioral change, apps that use the method could exhibit compatibility issues. Specifically, an app that relies on the method's behavior as implemented in API level 27 might encounter a failure when running on newer versions of the API. Because API developers did not suitably document the change, app developers might not be aware of the change and hence have a low chance of avoiding such compatibility issue. In the rest of this paper, we present a systematic study that carefully analyzes this family of changes across multiple versions of the Android API and investigates to what extent these changes might affect Android apps.

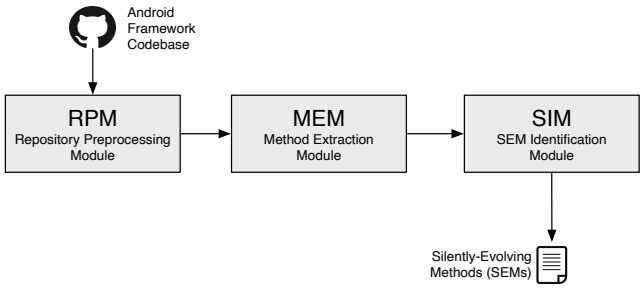


Fig. 1. High-level overview of the ANDROSEA workflow.

TABLE I
ANDROID FRAMEWORK REVISIONS USED IN THE STUDY. WE DID NOT CONSIDER API LEVEL 20 AS THIS VERSION WAS FOCUSING ON CHANGES FOR SUPPORTING ANDROID WEAR.

API Level	Code Name	Release Tag	Distribution
29	Android10	android-10.0.0_r1	8.2%
28	Pie	android-9.0.0_r1	10.4%
27	Oreo	android-8.1.0_r1	15.4%
26	Oreo	android-8.0.0_r1	12.9%
25	Nougat	android-7.1.0_r1	7.8%
24	Nougat	android-7.0.0_r1	11.4%
23	Marshmallow	android-6.0.0_r1	16.9%
22	Lollipop	android-5.1.0_r1	11.5%
21	Lollipop	android-5.0.0_r1	3.0%
19	KitKat	android-4.4_r1	6.9%

III. METHODOLOGY

Our analysis is based on the Android framework codebase. This codebase is one of the largest repository made available on Github and contains over 440,000 commits and nearly one thousand release tags. In this study, we focus our analysis on the ten most recent[2] major version releases of the Android framework, as these versions are the ones that are widely used on user devices. (Older releases were less popular and their distribution accounted for less than 3% of the total distribution.) When selecting a revision for analyzing a major version release, we chose the first release tag associated with each version considered. Table I reports the details of the versions and revisions we considered. Using the versions listed in Table I, we built nine subsequent version pairs (e.g., version 19 and 21 constitute a version pair), and use these pairs to identify SEMs.

To the best of our knowledge, no readily-available tool exists to detect SEMs. For this reason, we implemented a prototype tool called ANDROSEA, which identifies SEMs across different version of the Android framework. At a high-level, the tool takes as inputs the repository containing the codebase of the Android framework and the list of framework version pairs that ANDROSEA should compare. Given this information, the tool analyzes the version control history of the repository to compare methods across different versions of the Android API. In this step, the tool categorizes a method as a SEM if the method has the same signature, the same method comment, but different method body. The output of ANDROSEA is a list of SEMs for each version pair analyzed.

Fig. 1 presents a high-level overview of the ANDROSEA workflow. As the figure highlights, ANDROSEA uses three modules to identify SEMs. The three modules are the *repository preprocessing module* (RPM), the *method extraction module* (MEM), and the *SEM identification module* (SIM). We now present the three modules in detail.

A. Repository Prepocessing Module

Each release of the Android framework codebase contains a large variety of files. In fact, the codebase includes the core implementation of the Android API, the source code of various command-line tools (such as the Android Asset Packaging Tool), the source code of unit tests, and other types of files. Because not all of these files are part of the Android API, the repository prepocessing module analyzes the codebase to identify and select the files that related to the Android API. The repository preprocessing module performs this task by using a whitelist that we manually constructed, and performs this step for each framework version provided as input to ANDROSEA. By performing this operation, ANDROSEA reports only the SEMs that are related to the Android API. ANDROSEA provides the list of relevant files to the method extraction module for further analysis.

[2]When we started the study in March 2020.

B. Method Extraction Module

This module locates and extracts relevant information about the Java methods in the source code files of the Android API. Specifically, for each version of the API, the module creates a set of tuples (*apiInfo*) where each tuple (*mInfo*) represents a method in the API and contains the signature (*signature*), the comment (*comment*), and the body (*body*) of the method. The module uses a Java parser to build the abstract syntax tree (AST) for each of the source code files and identifies the API methods in a file by navigating the AST. After locating a method, the module stores the method signature, the method comment, and the method body in *apiInfo*. ANDROSEA only saves the comments declared through the Javadoc notation (i.e., `/**...*/`) since only these comments will appear in the documentation of the API. The output of this module are the sets of tuples that are associated with the versions of the API considered.

C. SEM Detection Module

The SEM detection module identifies SEMs by comparing the relevant methods in the version pairs provided as input to ANDROSEA. Algorithm 1 describes how the module identifies SEMs. Given the methods' information from two subsequent versions of the Android API (*apiInfo1* and *apiInfo2* in Algorithm 1), the module iterates over the methods in the versions and compares them (lines 1-15). When the algorithm finds methods with matching signatures (i.e., they are the same method in different versions of the API), ANDROSEA first checks whether the methods have a comment associated with them. If either one of the methods does not have a comment,

Algorithm 1: Detecting Silently-Evolved Methods.

```
   Input  : apiInfo1 and apiInfo2: method information from
            two API versions
   Output: sems: set of silently-evolved methods between the
            two Android versions considered
1  for mInfo1 ∈ apiInfo1 do
2      for mInfo2 ∈ apiInfo2 do
3          if mInfo1.signature ≠ mInfo2.signature then
4              continue
5          end
6          if mInfo1.comment.isEmpty ||
             mInfo2.comment.isEmpty then
7              continue
8          end
9          if mInfo1.comment == mInfo2.comment then
10             if mInfo1.body ≠ mInfo2.body then
11                 sems.add(mInfo1, mInfo2)
12             end
13         end
14     end
15 end
16 return sems
```

ANDROSEA will not consider the methods for further analysis (line 7). The rationale behind this decision is that we believe that such methods might not be intended for use by app developers as they often resort to the official documentation to learn how to use the API methods. If both methods have a comment, ANDROSEA checks whether the comments are the same or not by comparing their text. If the comments have the same text, ANDROSEA moves forward and compares their method bodies. For simplicity, ANDROSEA compares the method bodies using the text of their bodies. If the bodies are different, ANDROSEA categorizes the method as a SEM and adds the method information to the set of SEMs computed for the API versions pair under analysis.

IV. EXPERIMENTAL STUDY

This section discusses our empirical study. In the study, we investigated the following research questions

- **RQ1:** To what extent do SEMs appear in Android API?
- **RQ2:** What are the characteristics of SEMs?
- **RQ3:** How do SEMs evolve during the development of the Android API?
- **RQ4:** To what extent are PASEMs used in Android apps?

In the rest of this section, we answer the research questions by presenting our experimental findings.

A. RQ1: SEMs in the Android API

With the first research question, we are interested in quantifying the number of SEMs in the Android API. To this end, we ran ANDROSEA on the source code of the Android framework codebase using the list of release tags shown in Table I. ANDROSEA extracted all the methods in each release and conducted a pairwise comparison between each subsequent version pair (e.g., between *android-4.4_r1* and *android-5.0.0_r1*). Since our study considered ten major version releases of the Android API, ANDROSEA conducted nine pairwise comparisons. In total, ANDROSEA was able to identify 4,769 SEMs and 2,271 of these SEMs are PASEMs.

After identifying SEMs, we also analyzed the modifiers and annotations associated with the methods to determine the potential impact of the methods on client apps. Table II reports the number of SEMs identified by ANDROSEA and categorizes them by their modifiers and annotations. The table is divided into five sections. The first section (API Level) reports the information of the version pair considered in the study. The other four sections group methods according to their Java access modifier. In Table II, the columns labeled with the symbol '-' report the number of SEMs that have the Java access modifier as their only modifier.

The number of SEMs varies quite significantly among the nine updates, i.e., can be as large as 943 methods or as small as 84. Fig. 2 presents the correlation (obtained via Pearson's correlation coefficient) between the total number of SEMs and the difference between the number of commits (Fig. 2 (a)), the number of methods (Fig. 2 (b)), and the number of updated methods (Fig. 2 (c)) in the two releases considered. These correlation results (i.e., Pearson's correlation coefficient R and $p-value$) show that the introduction of SEMs is not strongly correlated with the number of Github commits (which do not necessarily lead to method changes) but strongly correlated with the difference in the number of methods, especially the difference in the number of updated methods, in two subsequent releases.

As shown in Table II, the majority of SEMs are declared as public (i.e., PASEMs) and these methods can be accessed by a number of Android apps. Based on the update types of PASEMs, these apps may be subject to major compatibility issues, which can lead to field failures if the update has changed the method's semantics (e.g., the API method presented in our motivating example and reported in Listing 1). We believe that API developers should pay particular attention to updating these method comments so that app developers can suitably account for the semantic changes affecting their apps.

Answer to RQ1

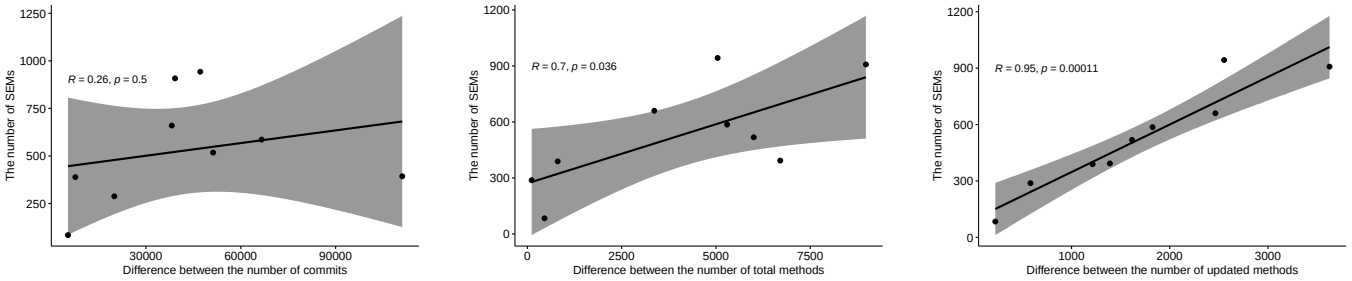
Based on our empirical results, we can confirm that SEMs are present in the Android API. Considering ten major version releases of the API, we were able to identify 4,769 SEMs, including 2,271 PASEMs. These PASEMs could lead to compatibility issues in their client Android apps. Furthermore, the more methods are updated in a version release, the more SEMs can be introduced.

B. RQ2: Understanding SEMs

In the second research question, we are interested in understanding the main purposes behind the updates of SEMs. Specifically, we would like to check whether the updates are related to simple code refactorings that would introduce no harm to the system or involved in semantic changes that could break the execution of existing Android apps. Ideally, we would expect that SEMs should only involve code refactorings

TABLE II
SEMs IDENTIFIED IN OUR STUDY CATEGORIZED BY THEIR MODIFIERS AND ANNOTATIONS.

API Level	public						default						protected						private					
	-	static	final	abstract	hidden	native	-	static	final	abstract	hidden	native	-	static	final	abstract	hidden	native	-	static	final	abstract	hidden	native
19→21	502	92	33	1	142	0	38	6	1	0	6	0	16	0	1	0	24	0	52	5	3	0	39	0
21→22	112	18	4	125	80	0	16	2	1	0	1	0	1	0	1	4	7	0	28	4	1	0	14	0
22→23	279	75	49	0	171	9	15	9	1	0	4	0	7	0	0	0	4	0	63	10	1	0	28	0
23→24	436	80	28	0	336	0	16	9	2	0	5	0	5	0	0	0	4	0	44	13	0	0	15	0
24→25	26	9	5	0	37	0	4	0	0	0	1	0	0	0	0	0	3	0	3	3	0	0	2	0
25→26	233	60	14	0	200	0	19	4	5	0	24	0	3	0	0	0	7	0	32	9	2	0	13	0
26→27	64	25	3	0	193	0	5	2	0	0	5	0	0	0	0	0	0	0	7	2	0	0	2	0
27→28	145	111	15	0	151	0	9	3	0	0	4	0	0	0	0	0	4	0	19	1	0	0	3	0
28→29	166	63	11	0	263	0	4	1	0	0	14	0	2	0	0	0	6	0	29	4	0	0	6	0



(a) (b) (c)

Fig. 2. Correlations between the number of SEMs and the differences of the number of commits, total methods, and updated methods in two subsequent releases, respectively.

such as renaming variables and attributes. They should not include semantic changes as those changes could break the execution of client apps. The comments, especially the Javadocs, of the corresponding methods should have subsequently been updated to properly advise app developers to update their apps so as to be aligned with the changed APIs' new semantics.

```
1 // Between android-8.1.0_r1 and android-9.0.0_r1
2 public CharSequence[] getTextArray(@StyleableRes int
      index) {
3   final TypedValue value = mValue;
4 - if (getValueAt(index * AssetManager.STYLE_NUM_ENTRIES,
      value)) {
5 + if (getValueAt(index * STYLE_NUM_ENTRIES, value)) {
```

Listing 2. An example of PASEM flagged as code refactoring.

Unfortunately, to the best of our knowledge, our community has not yet made available tools for effectively determining whether an update of method code (i.e., method diff) is related to semantic change or not. To this end, in this work, we resort to manual efforts to classify the purposes behind the updates of PASEMs (i.e., the nature of the changes). We choose to classify PASEMs instead of SEMs because only PASEMs could directly impact the execution of client apps available in the wild. Since manual efforts are known to be time-intensive, it becomes impractical for us to manually classify all the PASEMs identified previously. To that end, we resort to randomly sample a set of PASEMs to fulfill the purpose. To ensure that the sampled PASEMs are representative, we turn to the well-known online Sample Size Calculator[3] to determine the number of PASEMs for manual classification (with a confidence level of 95% and margin of error of 10%).

The second column in Table III enumerates the number of samples randomly selected from each framework iteration. For each method, two of the authors of the paper independently categorized the method as either *refactoring* or *semantic change*. If the authors cannot make a decision in 10 minutes, the corresponding PASEM will be flagged as uncertain. After completing the independent manual classification, the two authors then set up meetings to discuss their decisions until consensuses reached. The final results are summarized in the last four columns in Table III. In the manual classification, only 97 out of 562 are classified as different types by the two different authors achieving a high inter-rater reliability of 82.74%. After the meeting, 41 out of 97 are concluded as semantic changes accounting for 42.27% while the refactorings and uncertains are made up of 23.71% and 34.02% respectively. Surprisingly, all in all, slightly more than a quarter of the randomly selected PASEMs are related to code refactorings (Listing 2 presents such an example), and around two-thirds of the randomly selected PASEMs are related to semantic changes (e.g., logic added, removed, or changed that is complicated update including statements added and removed). This result shows that SEMs are a severe problem in the Android framework. The framework maintainers should pay special attention to carefully handle these methods, i.e., avoid introducing SEMs in future releases and document the semantically changed methods.

```
1 // Between android-9.0.0_r1 and android-10.0.0_r1
2 public void setVolumeTo(int value, int flags) {
3   try {
4 - mSessionBinder.setVolumeTo( mContext.getPackageName(),
      mCbStub, value, flags);
5 + // Note: Need both package name and OP package name.
      Package name is used for
6 + // RemoteUserInfo, and OP package name is used for
      AudioService's internal
7 + // AppOpsManager usages.
8 + mSessionBinder.setVolumeTo( mContext.getPackageName(),
```

[3]https://www.surveysystem.com/sscalc.htm

TABLE III
MANUAL CLASSIFICATION ON SELECTED SAMPLE PASEMS.

API Level	Sample	Update type				Refactoring	Uncertain	Disagreement
		Added	Removed	Changed	Sum			
19→21	83	16	2	34	52 (62.65%)	22 (26.51%)	9 (10.84%)	22 (26.51%)
21→22	54	17	1	14	32 (59.26%)	19 (35.19%)	3 (5.56%)	1 (1.85%)
22→23	76	23	1	27	51 (67.10%)	21 (27.63%)	4 (5.26%)	21 (27.63%)
23→24	81	29	1	35	65 (80.25%)	13 (16.05%)	3 (3.70%)	9 (11.11%)
24→25	25	8	1	11	20 (80.00%)	2 (8.00%)	3 (12.00%)	5 (20.00%)
25→26	71	15	2	24	41 (57.75%)	23 (32.39%)	7 (9.83%)	11 (15.49%)
26→27	42	13	0	15	28 (66.67%)	14 (33.33%)	0 (0.00%)	2 (4.76%)
27→28	65	9	0	21	30 (46.15%)	24 (36.92%)	11 (16.92%)	12 (18.46%)
28→29	65	13	2	29	44 (67.69%)	10 (15.38%)	11 (16.92%)	14 (21.53%)
Total	562	143	25	199	363 (64.59%)	148 (26.33%)	51 (9.07%)	97 (17.26%)

```
    mContext.getOpPackageName(), mCbStub, value, flags);
} catch (RemoteException e) {
```

Listing 3. An example of PASEM flagged as uncertain.

As shown in Table III (the fifth column), around 10% of PASEMs are flagged as uncertain. The majority of those methods are related to updates of callee methods, which may further involve complicated changes. Listing 3 presents such an example. The original callee method *setVolumeTo()* called by *mSessionBinder* has been replaced by a new one that involves a new parameter, which is not needed by the original version. The callee method is only defined in a Java interface called *ISessionController*, which is non-trivial for the authors to manually identify its dynamically bound object (in a short time), considering that the Android framework is one of the most complicated open-source projects. Therefore, this update is flagged as *uncertain*.

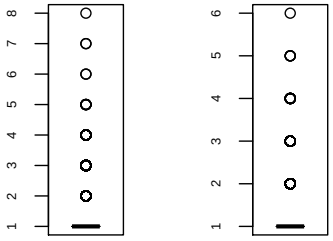
Answer to RQ2

Our manual classification reveals that the majority of PASEMs (over 64.59%) do involve semantic changes, and such changes may involve complicated updates of the code.

C. RQ3: Evolution of PASEMs

In this research question, we are interested in exploring the evolution of SEMs under the evolution of the Android framework codebase. To this end, we first look at the number of times a given method is silently changed. Fig. 3 illustrates the distribution of such times for all the identified 4,769 SEMs and 2,271 PASEMs. Expectedly, the majority of methods (61.12%) that are flagged as SEMs are only silently evolved once, in the meanwhile, 79.03% of PASEMs are silently evolved once, suggesting that SEMs might not be intentionally introduced by the framework developers. Nevertheless, there are several methods that have indeed been repeatedly changed silently. For example, method *loop* of file `core/java/android/os/Looper.java` has been silently updated six times, among the nine considered iterations. The class Looper is used to run a message loop for a thread. For the specific method, it is actually a static public method that is used to run the message queue in this thread.

Among the 4,769 SEMs, interestingly, only 80 of them have their comments updated along with the update of the



(a) SEMs (b) PASEMs

Fig. 3. Distribution of update times of SEMs from level 19 to 29.

implementation in the following up revisions of the Android framework. This experimental result suggests that framework maintainers might not yet be made aware of SEMs or at least are not well-motivated to mitigate the introduction of SEMs. Listing 4 illustrates one of such updated comments. The implementation of the method has been updated in the release android-5.0.0_r1 while its comments are only updated at the revision android-6.0.0_r1. Nevertheless, although rare, the fact that some of the SEMs are indeed resolved by the framework maintainers shows that SEMs are indeed a problem that should be carefully addressed.

```
1  //android.widget.CheckedTextView.setCheckMarkDrawable
2   /**
3  - * Set the checkmark to a given Drawable. This will
       be drawn when {@link #isChecked()} is true.
4  - * @param d The Drawable to use for the checkmark.
5  + * Set the check mark to the specified drawable.
6  + * <p>
7  + * When this view is checked, the drawable's state
       set will include
8  + * {@link android.R.attr#state_checked}.
9  + * @param d the drawable to use for the check mark
10 + * @attr ref
       android.R.styleable#CheckedTextView_checkMark
11   * @see #setCheckMarkDrawable(int)
12   * @see #getCheckMarkDrawable()
13 - * @attr ref
       android.R.styleable#CheckedTextView_checkMark
14   */
```

Listing 4. The comment of SEM *setCheckMarkDrawable* is updated in a future release while its body is not changed.

We further look at the evolution of method modifiers for all the identified SEMs. Fig. 4 summarizes such results in a directed graph. Each node represents a distinct modifier

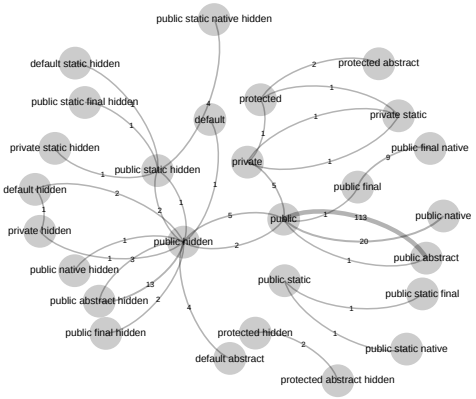


Fig. 4. The update of modifiers. The line in clockwise shows the direction of the update and the weight on the line shows the number of SEMs having such updates.

type, while each arrowed edge represents an evolution, e.g., from a *src* modifier to the *dest* modifier. The weight of each edge subsequently represents the number of times the corresponding evolution happens during the evolution of the Android framework codebase.

In total, we have observed 204 times of modifier changes. Expectedly, none of the changes are made from *public* to other low-level accessibilities (except one from *public static hidden* to *private static hidden*[4]) as such changes would break the compilation rules for Android apps, resulting in explicit compatibility issues. Interestingly, the majority of modifier changes are related to altering *non-abstract* methods to *abstract* methods or exchanging *native* methods with *non-native* methods. Listing 5 represents an example of modifiers update from *public* to *public abstract* extracted between android-5.0.0_r1 and android-5.1.0_r1. The class, the method belongs to, is actually an abstract class. The body of the method is to provide the default behavior, which is to throw `MustOverrideException` if the subclass extends the *WebSettings* but does not explicitly override the method. The update of the method reinforces the need to override the method by declaring it as *abstract*, which utilizes the compile-time check to ensure that the method will be overridden in the subclass. Although this type of change is trivial, it will theoretically cause compatibility issues for existing Android apps. Indeed, previously, even if developers do not override the API method *setBuiltInZoomControls()* when extending the *WebSettings* class, grammatically speaking, there will be no compile error. However, with the latest SDK version, the accessibility of *setBuiltInZoomControls()* has changed to *public abstract*, there will be a compile error if the method is not explicitly overridden. Therefore, ideally, this type of change should be avoided by framework maintainers, and subsequently should be considered by software analyzers aiming at detecting compatibility issues in Android apps.

[4]Hidden APIs cannot be directly accessed by Android apps.

```
1 // This is an abstract base class: concrete
      WebViewProviders must
2 // create a class derived from this, and return an
      instance of it in the
3 // WebViewProvider.getWebSettingsProvider() method
      implementation.
4 public abstract class WebSettings {
5 - public void setBuiltInZoomControls(boolean
      enabled) {
6 -  throw new MustOverrideException();
7 - }
8 + public abstract void
      setBuiltInZoomControls(boolean enabled);
```

Listing 5. An example of modifier update from public to public abstract.

The majority of the updates related to native methods share the same change pattern, which is to provide a wrapper method with the same name while calling the native method to provide the same behavior (cf. (Listing 6). Even though the modifiers were updated from *public native* to *public*, the signatures of the APIs provided to developers are not changed. Practitioners hence can still use the same method signature to implement their intentions. Therefore, this type of update will unlikely to introduce compatibility issues into running Android apps.

```
1 // Code snippet from android-4.4_r1
2 94  public native int getHeight();
3 // Code snippet from android-5.0.0_r1
4 107 public int getHeight() {
5 108  return nativeGetHeight(mNativePicture);
6 109 }
```

Listing 6. An example of modifier update from public native to public.

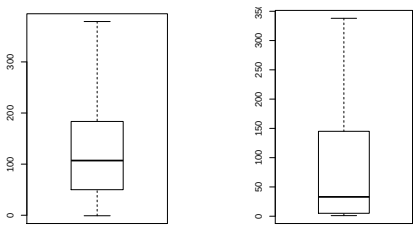
Answer to RQ3

The majority of SEMs is only introduced into the framework once, without following up updates. Moreover, SEMs may involve updating the method's modifiers that could further introduce runtime issues to client Android apps.

D. RQ4: PASEMs in Android apps

Since PASEMs may involve semantic changes, their client apps may suffer from incompatible issues (because of those silent semantic changes) when they are running on devices with different SDK versions. Therefore, in this last research question, we are interested in knowing if PASEMs are used by Android apps. If so, to what extent are they accessed, and what are the potential impacts such usages could bring to Android app developers? To answer these questions, we introduce a simple app scanner to the community, which takes as input an Android APK and the list of PASEMs identified in this work and outputs the list of PASEMs that are actually accessed by the APK. The app scanner is implemented on top of the famous Soot analysis framework [18]. The APIs are identified at the Jimple level, which is one of the intermediate representation types provided by Soot to ease the analysis of Android apps.

Specifically, to fulfill our experiments, we randomly select 1,000 apps published in 2020 from the official Google Play store. All of the selected apps are then sent to the aforementioned app scanner to check whether they have leveraged PASEMs or not. Interestingly and surprisingly, among the



(a) Total PASEMs. (b) Semantically updated PASEMs.

Fig. 5. The distribution of the number of PASEMs used in a set of randomly selected 1,000 APKs published on 2020.

1,000 apps, 957 of them have accessed PASEMs. Moreover, in total, 44.25% PASEMs are accessed. Fig. 5(a) further presents the distribution of the number of PASEMs accessed by these apps, giving a median and average number of PASEMs at 107 and 126, respectively. This experimental result shows that PASEMs have been significantly accessed by Android apps, which could subsequently suffer from "hidden" incompatible issues. This result strongly suggests that the Android framework maintainers should pay special attention to avoid introducing PASEMs. This finding is further backed up by the fact that semantically updated PASEMs are also significantly accessed by Android apps, as illustrated in Fig. 5(b), for which only the 363 manually confirmed PASEMs (involving semantic changes) are considered.

Previous studies, such as the one proposed by He et al. [14], shows that the convention to address the evolution-induced API compatibility issue is to add additional conditions to check the practical running API level of the device (e.g., through the default constant value of `VERSION.SDK_INT`) before invoking the corresponding APIs. In this work, we try to detect how many different PASEMs are called after the API level condition check, where the PASEMs are referred to as protected PASEMs. Interestingly, 469 out of 1,000 Apps indeed contain protected PASEMs, for which there are 134 PASEMs called under protection. Table IV presents the top-10 PASEMs ranked by their protection times. The fact that the top-ranked PASEMs are protected more than 800 times shows that PASEMs could be protectively accessed many times within the same app.

Moreover, we go one step deeper to check the actual API levels leveraged for protecting the invocation of PASEMs. Our experimental results reveal that a large portion of PASEMs (i.e., 45 or 33.58%) are protected by different API levels, indicating possible errors of app developers, although they have attempted to protect the accessed PASEMs. The possible reasons behind these disagreements between developers could be that PASEMs are silently (hence hiddenly) introduced to the framework. There is hence no documentation for developers to correctly use these PASEMs. Subsequently, developers have to independently identify the API levels suitable for protecting PASEMs independently, based on their empirical evidence. The fact that developers have attempted to protect the access of PASEMs (although may incorrectly do so because of lacking documentation) indicates that PASEMs can indeed introduce runtime issues to Android apps. Therefore, we argue that PASEMs should be totally avoided by framework maintainers. There is also a need to introduce automated tools to regulate that. Our approach ANDROSEA could be leveraged to achieve such a purpose.

TABLE IV
THE TOP TEN PROTECTED PASEMS IN 1,000 APKS

PASEM	times
android.content.res.TypedArray: void recycle()	848
android.os.Bundle: void putParcelable(String,Parcelable)	788
android.os.Bundle: void putCharSequence(String,CharSequence)	623
android.text.TextUtils: void writeToParcel(CharSequence,Parcel,int)	384
android.app.AlarmManager: void setExact(int,long,PendingIntent)	359
android.app.AlarmManager: void setExactAndAllowWhileIdle(int,long,PendingIntent)	289
android.app.Activity: boolean navigateUpTo(Intent)	286
android.app.Activity: void startIntentSenderForResult(IntentSender,int,Intent,int,int,int,Bundle)	285
android.view.ViewGroup: void removeView(View)	283
android.os.Bundle: void putAll(Bundle)	273

Answer to RQ4

PASEMs have indeed commonly been accessed by real-world Android apps. Some of them are even accessed with protections (by checking the running API level), indicating that practitioners have realized that those PASEMs could introduce runtime issues to their apps, although they are not documenting.

V. DISCUSSION

We now discuss the potential implications of this study for both practitioners and our fellow researchers, as well as some promising future research directions that could be built on the findings of our research (cf. V-A). After that, we present the potential threats to the validity of this study (cf. Section V-B).

A. Implication for Practitioners and Researchers

As shown in Figure 5, the usages of PASEMs are common in real-world Android Apps, and over 60% of the PASEMs involve truly semantic changes, as demonstrated in Table III. As semantic changes could introduce potential crashes (or security, efficiency issues) in daily use of the Android Apps, we argue that framework maintainers should try their best to avoid introducing PASEMs. This should also apply to the maintenance of any other third-party frameworks or libraries that provide APIs to facilitate the development of client apps. Subsequently, the client app developers should pay special attention to those silently evolved methods when developing their apps.

As shown in Table III, slightly more than a quarter of SEMs do not involve semantic changes but are simply related to code refactorings. When performing compatibility analyses, these methods could be ignored as they will not introduce runtime issues to their users (i.e., client apps). However, to the best of our knowledge, our community has not introduced promising tools to automatically decide if a given code diff involves semantic changes. State-of-the-art refactoring detection tools [19], [20] are not capable of accurately achieving

that. Therefore, we argue that there is a strong need to invent an automated approach to locate semantic changes during the evolution of software systems. With the help of this tool, SEMs with semantic changes could be automatically identified and thereby mitigated by codebase maintainers.

Besides the implementation update of the methods, the methods' modifiers might also be updated along with. The update of the modifiers also could bring in big problems, such as the example in listing 5. Therefore, we argue that our fellow practitioners should pay attention to the updates of methods' modifiers when updating their software systems.

B. Threats to Validity

Our study has carried a number of threats to validity that we have attempted to mitigate. First, SEMs in the paper are extracted from nine Android revision pairs, which do not represent the whole picture of Android evolution. Subsequently, many PASEMs may have been overlooked by our study. Nevertheless, to alleviate this threat, we have considered all the major first releases, and thereby the experimental results obtained in this work should still be representative. Second, our work involves substantial manual tasks, which could introduce errors to the final results since human efforts are known to be error-prone. For example, the work related to understanding why some methods are silently evolved involves manual summarization of the diff code snippets. To mitigate this threat, we have cross-validated the results. Third, the experimental results presented for answering RQ4 might be impacted by the selected apps' representativeness. We have attempted to mitigate this threat by randomly selecting apps from AndroZoo, one of the most comprehensive app datasets made available in our community.

VI. Related Work

Android API analysis has been a hot topic in the research community [21], [22]. We now summarize the representative ones.

A. API evolution

While API evolves to meet new feature requirements, to fix bugs etc., developers need to update their implementation of the Application and publish the newer version to provide a stable running environment for their customers. McDonnell et al. [4] conducted an extensive empirical study on the stability and adoption of Android APIs while focusing on the relationship between the API evolution and client adoption. The authors in the paper confirm that the Android API evolves more frequently than the client adoption and what's more, the more frequent update of the APIs the longer time for client to adopt the ones. Bavota et al. [7] and Linares-Vásquez [6] studied the relationship between the popularity of the Android Applications and the stability of the SDK APIs. Their empirical study reveal that the more enjoyable Android Apps are prone to call the less updated APIs. Works [13] and [14] investigate the API-related compatibility issues. Li et al. [13] present an approach CiD to highlight the API usage that can lead to potential compatibility issues by analysing the framework release history and identifying the methods without API level checking. He et al. [14] investigate the evolution-induced compatibility issues in Android Applications. Their research shows that the Android Support library only provides limited support for the new APIs in each release and the majority of the Applications need to handle the evolution-induced compatibility issues in their own implementation. Different from the existing work focusing on the general APIs, what we do is to disclose the silently evolved methods that always ignored by developers and researchers.

B. API pattern

In addition, researchers propose many different approaches to detect Android malware to address the security problems of remote control, privilege escalation, and privacy leakage etc. Chan et al. [23] proposed a static approach for Android malware detection via extracting permissions and API usage. They confirm that the integration between the feature of permission and API calls can achieve a better precision than just the only feature of permission. Karbab et al. [24] introduced an automatic and effective Android malware detection system, MalDozer, that depends on deep learning techniques and raw sequences of API method calls. Arp et al. [25] proposed a tool DREBIN that builds a SVM based detection model utilizing APIs and other related information. While Ma et al. [26] decompile the Android Apps and construct three different system API data sets: API usage, API frequency, and API sequence to detect malware. To be specific, Linares-Vásquez et al. [27] attempted to reveal the APIs and usage patterns related to energy consumption, and to provide potential guidance for developers to decrease energy consumption.

C. Special APIs

Android APIs typically follow the general *deprecated-replace-remove* evolution cycle. Work [15] introduced prototype tool called CDA to characterize deprecated APIs from different revisions. Their extensive investigation shows that the deprecated APIs are not continuously annotated and documented and over a half of these APIs are commented to provide alternatives but these alternatives are rarely replaced by the developers. Besides the aforementioned general publicly accessible APIs, there exists another type of API referred to as inaccessible API that can be recognized as internal or hidden. Internal APIs are resolved ones for system apps located in the package `com.android.internal` while hidden APIs are methods annotated by the javadoc `@hide`. Li et al. [16] did an extensive investigation to reveal the usability of these APIs. They demonstrate that these inaccessible APIs are continuously implemented in the Android framework and used to access a specific set of features while without any promise of forward compatibility. They also reveal that there exist a plenty of apps are indeed calling these inaccessible APIs and the patterns of usage are quite different between each other.

VII. Conclusion

In this paper, we presented a prototype tool, namely AndroSea, to detect silently-evolved methods in the Android framework. By applying AndroSea to the Android framework codebase, we found that SEMs indeed exist in Android, and many of the identified SEMs can be publicly accessed (referred to as PASEMs) by Android app developers. Through an exploratory study on the identified SEMs, we further empirically found that (1) the majority of PASEMs involves semantic changes that could lead to runtime issues of their client Android apps, (2) the majority of SEMs is only introduced into the framework once, indicating that framework maintainers may not have been aware of that, (3) PASEMs have indeed been recurrently accessed (even with protections) by real-world Android apps.

VIII. Acknowledgments

This work was partially supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, a Discovery project DP200100020, and a gift from Facebook.

References

[1] eMarketer, eMarketer Unveils New Estimates for Mobile App Usage. https://www.emarketer.com/Article/eMarketer-Unveils-New-Estimates-Mobile-App-Usage/1015611, 2020, Last updated: 11-Apr-2017.

[2] Mobile operating systems' market share worldwide from January 2012 to July 2020. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009, 2020, Last updated: 17-Aug-2020.

[3] Number of available applications in the Google Play Store from December 2009 to June 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store, 2020, Last updated: 17-Aug-2020.

[4] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *ICSM*, 2013, pp. 70–79.

[5] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan, "Studying the relationship between source code quality and mobile platform dependence," *Software Quality Journal*, p. 485–508, 2015.

[6] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: a threat to the success of android apps," in *FSE*, 2013, pp. 477–487.

[7] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change-and fault-proneness on the user ratings of android apps," *TSE*, vol. 41, no. 4, pp. 384–407, 2014.

[8] G. Yang, J. Jones, A. Moninger, and M. Che, "How do android operating system updates impact apps?" in *MobileSoft*. New York, NY, USA: ACM, 2018, pp. 156–160.

[9] L. Li, T. Bissyandé, and J. Klein, "Moonlightbox: Mining android api histories for uncovering release-time inconsistencies," in *ISSRE*. IEEE, 2018, pp. 212–223.

[10] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android apis," in *MSR*, 2018.

[11] API reference. https://developer.android.com/reference, 2020, Last updated: 15-Sep-2020.

[12] ConnectivityManager. https://developer.android.com/reference/android/net/ConnectivityManager#getAllNetworkInfo(), 2020, Last updated: 29-Sep-2020.

[13] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *ISSTA*, 2018.

[14] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 167–177.

[15] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Cda: Characterising deprecated android apis," *Empirical Software Engineering (EMSE)*, 2020.

[16] L. Li, T. F. Bissyandé, Y. Le Traon, and J. Klein, "Accessing inaccessible android apis: An empirical study," in *ICSME*. IEEE, 2016, pp. 411–422.

[17] SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements. https://dev.mysql.com/doc/refman/8.0/en/savepoint.html, 2020, Last updated: 29-Sep-2020.

[18] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *CETUS*, 2011.

[19] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[20] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014.

[21] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, 2017.

[22] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, 2018.

[23] P. P. Chan and W.-K. Song, "Static detection of android malware by using permissions and api calls," in *ICMLC*, 2014.

[24] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Android malware detection using deep learning on api method sequences," *arXiv preprint arXiv:1712.08996*, 2017.

[25] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[26] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma, "A combination method for android malware detection based on control flow graphs and machine learning algorithms," *IEEE access*, vol. 7, pp. 21 235–21 245, 2019.

[27] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *MSR*.