AndroEvolve: automated Android API update with data flow analysis and variable denormalization

Stefanus Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, Gilles Muller

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

AndroEvolve: Automated Android API Update with Data Flow Analysis and Variable Denormalization

Stefanus A. Haryono · Ferdian Thung · David Lo · Lingxiao Jiang · Julia Lawall · Hong Jin Kang · Lucas Serrano · Gilles Muller

Received: date / Accepted: date

Abstract The Android operating system is frequently updated, with each version bringing a new set of APIs. New versions may involve API deprecation; Android apps using deprecated APIs need to be updated to ensure the apps' compatibility with old and new Android versions. Updating deprecated APIs is a time-consuming endeavor. Hence, automating the updates of Android APIs can be beneficial for developers. CocciEvolve is the state-of-the-art approach for this automation. However, it has several limitations, including its inability to resolve out-of-method variables and the low code readability of its updates due to the addition of temporary variables. In an attempt to further improve the performance of automated Android API update, we propose an approach named AndroEvolve, that addresses the limitations of CocciEvolve through the addition of data flow analysis and variable name denormalization. Data flow analysis enables AndroEvolve to resolve the value of any variable within the file scope. Variable name denormalization replaces temporary variables that may present in the CocciEvolve update with appropriate values in the target file. We have evaluated the performance of AndroEvolve and the readability of its updates on 372 target files containing 565 deprecated API usages. Each target file represents a file from an Android application that uses a deprecated API in its code. AndroEvolve successfully updates 481 out of 565 deprecated API invocations correctly, achieving an accuracy of 85.1%. Compared to CocciEvolve, AndroEvolve produces 32.9% more instances of

Stefanus A. Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Hong Jin Kang
School of Information Systems, Singapore Management University
E-mail: {stefanusah,ferdianthung,davidlo,lxjiang, hjkang.2018}@smu.edu.sg

Julia Lawall, Gilles Muller
Inria, France
E-mail: {Julia.Lawall,Gilles.Muller}@inria.fr

Lucas Serrano
Sorbonne University/Inria/LIP6, France
E-mail: Lucas.Serrano@lip6.fr

correct updates. Moreover, our manual and automated evaluation shows that AndroEvolve updates are more readable than CocciEvolve updates.

Keywords Program transformation · Android · data flow analysis · readability · API deprecation · API update

1 Introduction

Android is currently one of the most prominent operating systems (OS) due to its vast amount of users. The Android OS is frequently updated to add new features or to fix bugs. Each new version includes changes in its APIs. Changes to Android APIs may deprecate older versions and render them unusable in the newer versions of the OS. While the Android OS is frequently updated, not all users adopt its latest version, resulting in many different versions of the Android OS being concurrently used. This problem, termed Android fragmentation (Han et al. 2012; Wei et al. 2016), occurs frequently in the whole Android ecosystem. To prevent errors caused by such API deprecation, developers need to constantly update deprecated-API usages in their code, while still maintaining backward compatibility with older Android versions. Aside from being cumbersome and time-consuming to mitigate, Android fragmentation also introduces security risks (Zhou et al. 2014).

Due to the nature of Android fragmentation, updating usages of deprecated Android APIs has become a priority (McDonnell et al. 2013; Wei et al. 2016). To help developers, several studies have proposed approaches that ease the process of updating Android API usages (Fazzini et al. 2019; Haryono et al. 2020; Li et al. 2018a). These approaches propose various heuristics to generate update patches; they are neither sound nor complete. Their goal is to provide recommendations that developers can either directly accept, accept with modifications, or reject. Simply put, they are developer-in-the-loop solutions rather than fully automated ones.

For example, CiD (Li et al. 2018a) automates the detection of API-related compatibility issues in Android apps. CiD analyzes the history of framework releases and use a static analyzer to locate code using a user-specified API to flag potential compatibility issues without proposing update alternatives. AppEvolve (Fazzini et al. 2019) provides an automated update for Android deprecated APIs. It uses both before- and after-update code examples to learn the update automatically. However, while it is able to provide an applicable update for some examples, it was found to have several weaknesses. Specifically, a replication study by Thung et al. (2020) demonstrated that the target file in AppEvolve, i.e., the file that contains usages of a deprecated API, needs to have a similar coding style with the code example used. Note that, different from the existing line-of-work on automated Java API update, e.g., (Li et al. 2015; Xi et al. 2019; Štrobl and Troníček 2013), the aforementioned Android-specific approaches consider the unique features of Android apps – the primary one being that an app typically needs to support multiple versions of Android OS due to Android fragmentation.

CocciEvolve, the latest approach to update deprecated Android API usages, is built on Coccinelle4J (Kang et al. 2019), a program transformation tool for Java that is based on Coccinelle (Lawall and Muller 2018). The main improvements that CocciEvolve provides compared to previous tools are: (1) generating *readable* update-scripts in Coccinelle4J's Semantic Patch Language (SmPL) (Lawall and Muller 2018), (2) generating updates by using only a single after-update example, and (3) being able to update multiple instances of deprecated API invocations within a single file. CocciEvolve shows better performance than AppEvolve on 112 target files. The code examples used in CocciEvolve must be in the form of an `if` statement that checks the Android version as its condition, and contains the updated API in the `then` statement and the old API in the `else` statement, or vice versa. A sample of such an after-update code example can be seen in Figure 1. In this after-update example code for the `getCurrentMinute()` API, an `if` statement checks the Android version of the application (line 1). The deprecated API `getCurrentMinute()` is within the `else` statement (line 4), while the updated API `getMinute()` is within the `then` statement(line 2). This form of code is common in Android applications.

```
1  if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.M) {
2      minutes = picker.getMinute();
3  } else {
4      minutes = picker.getCurrentMinute();
5  }
```

Fig. 1 An example of after-update code for `getCurrentMinute()` API

CocciEvolve's Limitations. CocciEvolve has two major limitations: (1) its inability to resolve variables defined outside of the method containing the invocations to the deprecated API (so called *out-of-method variables*); and (2) the presence of temporary variables in the updated code. The out-of-method variables problem limits the set of update examples that can be utilized by CocciEvolve. The presence of temporary variables significantly reduces the readability of the update results that CocciEvolve produces.

Out-of-method variables. An after-update example for the `requestAudioFocus(...)` API is shown in Figure 2. In this example, the deprecated API is `requestAudioFocus(...)` on line 77. The updated API and its argument are shown on lines 74–75. In this example, the method invocation argument for the updated API is the `request` object, shown on line 75. This object needs to be defined by invoking the method `audioFocusRequestOreo.getAudioFocusRequest()` (line 74). The variable `audioFocusRequestOreo` is defined outside of the method at line 41. Furthermore, this variable creates an object of the `AudioFocusRequestOreo` class, which is defined in lines 110–133. These variables and their definitions are new arguments for the updated API invocation that are not yet defined in the deprecated API invocation, thus they are not present in the target file. Since CocciEvolve only performs an

```
41 AudioFocusRequestOreo audioFocusRequestOreo = new AudioFocusRequestOreo();
   ...
67 public void tryToGetAudioFocus() {
68     OnAudioFocusChangeListener listener = this;
69     int result;
70     int type = AudioManager.STREAM_MUSIC;
71     int duration = AudioManager.AUDIOFOCUS_GAIN;
72     if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
74         AudioFocusRequest request = audioFocusRequestOreo.getAudioFocusRequest();
75         result = audioManager.requestAudioFocus(request);
76     } else {
77         result = audioManager.requestAudioFocus(listener, type, duration);
78     }
79 }
   ...
110 private class AudioFocusRequestOreo {
111     public AudioFocusRequest getAudioFocusRequest() {
            ...
132     }
133 }
```

Fig. 2 Sample out-of-method argument for an API invocation `requestAudioFocus(...)`

```
1 public void setTimeH(TimePicker tp, int hour) {
2     int parameterVariable0 = hour;
3     TimePicker classNameVariable = tp;
4     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
5         classNameVariable.setHour(parameterVariable0);
6     } else {
7         classNameVariable.setCurrentHour(parameterVariable0);
8     }
9 }
```

Fig. 3 An example of temporary variables in the output of CocciEvolve after updating the `setCurrentHour(...)` API invocation.

intra-procedural analysis of update examples and only considers line 74–75 for the update script creation, the variables related to the `request` object (lines 41 and 110–133) cannot be resolved, hence creating a non-working update.

Temporary variables in update results. Temporary variables are introduced in CocciEvolve to ease the process of code update by removing the variability in the code due to different syntax. However, these variables remain in the updated code, affecting its readability. Furthermore, these variables only refer to other variables that are already defined in the target file. Consider the sample updated code in Figure 3, where there exist two temporary variables, `parameterVariable0` (line 2) and `classNameVariable` (line 3). These variables refer to parameters of the `setTimeH` method, and can be removed and replaced by their definitions.

In this work, we propose AndroEvolve, an improved automated Android API usage update tool that addresses the limitations of CocciEvolve, with two new features: data flow analysis and variable name denormalization. AndroEvolve is built on top of CocciEvolve and serves as an extended and better update tool for deprecated Android API usages. Similar to CocciEvolve, AndroEvolve focuses on deprecated APIs for which the migration follows a

one-to-one API mapping, where one deprecated API is replaced with one updated API. During the update-script creation, data flow analysis is used to resolve the values used as API arguments, including all variables from the example that are defined outside of the method containing the deprecated API invocations to be updated. Definitions of such out-of-method variables are located and used to replace the variables in the API invocations. Variable name denormalization converts the normalized code introduced by AndroEvolve, which uses temporary variables, back to its original form *after* updates have been performed. Variable name denormalization improves the readability of the updated code as a *post-processing step* of the update script application process.

To evaluate the performance of AndroEvolve, we have conducted an experiment using a dataset of 372 target files containing 565 deprecated API usages from 20 different Android APIs. We have compared the performance of AndroEvolve against CocciEvolve by counting the number of successful updates produced by each tool. AndroEvolve successfully updates 481 out of 565 deprecated API usages, while CocciEvolve is only able to update 362 deprecated API usages. AndroEvolve produces successful updates for all cases where CocciEvolve is successful, while producing 119 (32.9%) more successful updates compared to CocciEvolve. To measure readability, we have compared the update results of AndroEvolve and CocciEvolve both automatically by using a popular readability scoring tool (Scalabrino et al. 2018) and manually by asking the opinions of two experienced Android developers. The measurements highlight that AndroEvolve produces updates that are more readable than CocciEvolve in both the automated and manual measurements. Specifically, the manual measurement reveals that the average readability score given by the Android developers for code updated by AndroEvolve is 4.533 (out of 5), with a median score of 5.0. Meanwhile, CocciEvolve updated code only received an average score of 1.10 (out of 5), with a median score of 1.0. This demonstrates that the readability of AndroEvolve's updated code has an appreciable impact on Android developers.

The main contributions of our work are as follows:

1. Different from the existing work on Android deprecated API updates (e.g., CocciEvolve), AndroEvolve analyzes data flows in a file-scope to infer more accurate transformations, as compared to previous tools which only work at the method-scope.
2. AndroEvolve is the first automated update tool for Android which considers the readability of the updated code. None of the previous tools have been evaluated in terms of readability. Our experiments demonstrate that AndroEvolve's output is significantly more readable and natural than those produced by the existing tools.
3. Our work substantially expands the evaluation dataset used to assess the performance of automated tools that update for Android deprecated API usages. The AndroEvolve evaluation dataset consists of 372 target files with 565 deprecated API usages collected from 311 distinct apps, a significant

improvement compared to AppEvolve's dataset (20 deprecated API usages from 15 apps) and CocciEvolve's dataset (112 API usages from 86 apps).

A tool demo paper for AndroEvolve is available at `https://arxiv.org/abs/2012.07259`. The tool demo paper contains a brief description of AndroEvolve, its usage guide, and examples for developers who are interested in using AndroEvolve in their work environment. This tool demo paper serves as a supplement of this paper, mainly for people who are more interested in applying AndroEvolve to their code.

The rest of this paper is organized as follows. Section 2 provides preliminaries on CocciEvolve, data flow analysis, and code readability. Section 3 discusses our approach in creating AndroEvolve as the upgraded version of CocciEvolve. Section 4 presents the experiments and their results. Section 5 discusses related work on API deprecation and program transformation. Section 6 discusses the threats to validity of our work. Lastly, Section 7 concludes our work and discusses future plans.

2 Preliminaries

CocciEvolve (Haryono et al. 2020) is the state-of-the-art tool on automatic Android API usage update. It distinguishes itself from other Android automatic update tool by only requiring an after-update example, providing a readable update-script, and introducing code normalization that tolerates some syntactic differences during code updates. Firstly, by requiring only an after-update example, CocciEvolve reduces the amount of examples that need to be provided by its users, compared to its predecessor AppEvolve (Fazzini et al. 2019) that requires both a before- and an after-update example. Secondly, a readable update-script is achieved through the use of Coccinelle4J (Kang et al. 2019). Coccinelle4J is a program matching and transformation tool for the Java language, ported from Coccinelle (Lawall and Muller 2018; Padioleau et al. 2008) for the C language. Coccinelle4J describes its transformation using the Semantic Patch Language (SmPL), which has similar syntax as *diff*. Having a patch that describes the program transformation helps the developers to better understand the updates and transformations applied. Finally, by normalizing both the after-update example and the target file, CocciEvolve minimizes the syntactic differences between them that may cause a failed update. Syntactic differences occur when the after-update and the target file API invocation arguments are expressed in a different coding style. This was one of the main limitations of AppEvolve, as shown in a replication study by Thung et al. (2020).

Given an after-update example containing the deprecated and updated API invocation within an `if` statement, CocciEvolve creates an update patch in SmPL that can be used to automatically update the usages of the deprecated APIs. Consider an after-update example for the `getCurrentMinute()` API shown in Figure 1. CocciEvolve normalizes the after-update example, resulting

in the normalized code shown in Figure 4. The update patch is created using this normalized code; the result can be seen in Figure 5.

```
1  if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.M) {
2      TimePicker classNameVariable = picker;
3      int tempFunctionReturnValue;
4      tempFunctionReturnValue = classNameVariable.getMinute();
5      minutes = tempFunctionReturnValue;
6  } else {
7      TimePicker classNameVariable = picker;
8      int tempFunctionReturnValue;
9      tempFunctionReturnValue = classNameVariable.getCurrentMinute();
10     minutes = tempFunctionReturnValue;
11 }
```

Fig. 4 Normalized after-update example for the code shown in Figure 1

```
1  @bottomupper_classname_assignment@
2  expression exp0;
3  identifier classIden;
4  @@
5  + if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
6  + tempFunctionReturnValue = classIden.getMinute();
7  + } else {
8  tempFunctionReturnValue = classIden.getCurrentMinute();
9  + }
```

Fig. 5 Update patch created from the normalized code shown in Figure 4

CocciEvolve can then apply this update patch to a target file using the `getCurrentMinute` deprecated API. Consider the example target file in Figure 6. The deprecated API `getCurrentMinute` is assigned to the variable `minute` in line 4. This target code is first normalized. Then, CocciEvolve applies the update patch into the normalized target code, creating the updated code shown in Figure 7. The deprecated API usage has been replaced with the temporary variable `tempFunctionReturnValue` (line 12), which is defined in the `if` statement containing the deprecated and updated API usages (lines 7–11).

```
1  public class AlarmCreator extends AppCompatActivity {
2      protected void onCreate(Bundle savedInstanceState) {
3          timePicker = (TimePicker) findViewById(R.id.timePicker);
4          int minute = timePicker.getCurrentMinute();
5      }
6  }
```

Fig. 6 Example target file containing a `getCurrentMinute` deprecated API usage

```
1  public class AlarmCreator extends AppCompatActivity {
2      protected void onCreate(Bundle savedInstanceState) {
3          timePicker = (TimePicker) findViewById(R.id.timePicker);
4          int minute;
5          int tempFunctionReturnValue;
6          TimePicker classNameVariable = timePicker;
7          if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
8              tempFunctionReturnValue = timePicker.getMinute();
9          } else {
10             tempFunctionReturnValue = timePicker.getCurrentMinute();
11         }
12         minute = tempFunctionReturnValue;
13     }
14 }
```

Fig. 7 Update code result for the target code shown in Figure 6

Data flow analysis, such as def-use analysis (Khedker et al. 2009), is an analysis of the data within the code based on the control flow paths taken by the program. For each given expression at a point inside a program, data flow analysis can locate and resolve the definition of the expression. Sample uses of data flow analysis are dead code elimination, variable value prediction, and program slicing (Bodík and Gupta 1997; Ghandour et al. 2010; Gupta et al. 1997; Hoffmann et al. 2013a; Weiser 1984). For our work, data flow analysis is mainly used to determine the values of variables used in the arguments for API invocations and to conduct program slicing (Hoffmann et al. 2013b).

Code readability is a measure of how easy it is to read a piece of code. It is an important code feature, especially for code that needs to be maintained for a long time, or code that is touched by multiple developers. Having readable code makes it easier for developers to understand and modify the code. Studies on code readability have been conducted extensively (Buse and Weimer 2010; Mi et al. 2018; Posnett et al. 2011; Scalabrino et al. 2016, 2017; Scalabrino et al. 2018). These studies define the metrics and features that are considered as important factors in determining code readability. These features include structural features (e.g. number of lines of code, length of each line of code, etc.), textual features (e.g. name of variables, consistency between comments and variable names, etc.), and entropy-based features (i.e., the amount of information in a piece of code, calculated based on the number of terms and the number of unique terms). Aside from these handcrafted features, a method that estimates code readability through CNN (Convolutional Neural Networks) has also been proposed by Mi et al. (2018).

3 Approach

AndroEvolve provides automated update for deprecated API usages where a deprecated API is replaced with one updated API including necessary changes to its arguments and surrounding code. The workflow of AndroEvolve is comprised of two main functionalities, update-script creation, and update-script

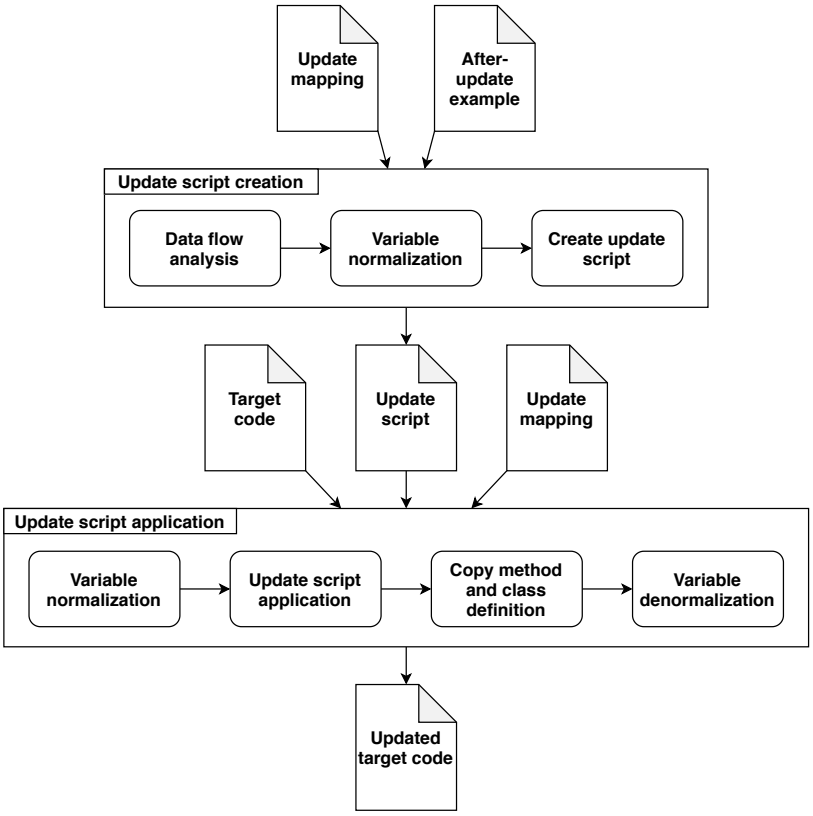

Fig. 8 Summary of the AndroEvolve workflow

application. Figure 8 provides a graphical description of this workflow. Update script creation takes as input the API update mapping and the after-update example of the API to create the update script. The API update mapping defines the mapping between the deprecated API and the updated API. Within the update script creation process, several components are at work. First, data flow analysis in the after-update example is used to resolve any out-of-method variables used by the updated API arguments. Following the data flow analysis, source file normalization is done on the code block containing the API invocation. Then, we take the `if` statement of this normalized code which contains the deprecated and updated API invocations to be used as the example to create the update script. Using this `if` statement, AndroEvolve automatically creates an update patch in the Semantic Patch Language (SmPL) that can be used to update the deprecated API usages.

This update script, along with the API update mapping and the target file, is the input for the update application process. Within the update application process, there are also several steps that are applied. First, source file normalization is applied to the target file containing the deprecated API

```
   // First Example
1  public class AudioPlayer {
2      AudioAttributes.Builder builder = new AudioAttributes.Builder();
3      AudioAttributes attributes = builder.build();
       ...
8      private void setAttributes() {
9          if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
10             mMediaPlayer.setAudioAttributes(attributes);
11         } else {
               ...
           }
       }
   }

   // Second Example
20 public class AudioPlayer {
       ...
24     private void setAttributes() {
25         if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
26             mMediaPlayer.setAudioAttributes(new AudioAttributes.Builder().build());
27         } else {
               ...
           }
       }
   }
```

Fig. 9 An example of different forms of argument for `setAudioAttributes` method invocation

usages. The update script is then applied to the normalized code. After the update is done, we copy the methods and class definitions that are used by the method invocation or object creations that are used in the updated API arguments. Finally, we apply variable name denormalization to remove temporary variables introduced by the source file normalization and replace their usage with the original expressions used as the API arguments.

One caveat of using AndroEvolve is that users need to provide the deprecated and updated API mappings, and an after-update example. To improve the usability of AndroEvolve, complementary approaches could be integrated to AndroEvolve's workflow. For example, an automated approach can be used to infer the deprecated and updated API mappings, as has been proposed by several prior works (Gokhale et al. 2013; Pandita et al. 2017). Meanwhile, to automate the process of obtaining the after-update example, automated code search methods such as AUSearch (Asyrofi et al. 2020) can be used together with AndroEvolve. These complementary approaches are not within the scope of our current work. Their integration into AndroEvolve is left for future work.

3.1 Data Flow Analysis

API method invocations can be written in multiple forms due to differences in the code style of the invocations and their arguments. Figure 9 shows examples of different forms of arguments for `setAudioAttributes` method invocations. In the first example, `Builder` and `AudioAttributes` objects are first instantiated and assigned into variables (lines 2 and 3) before being used

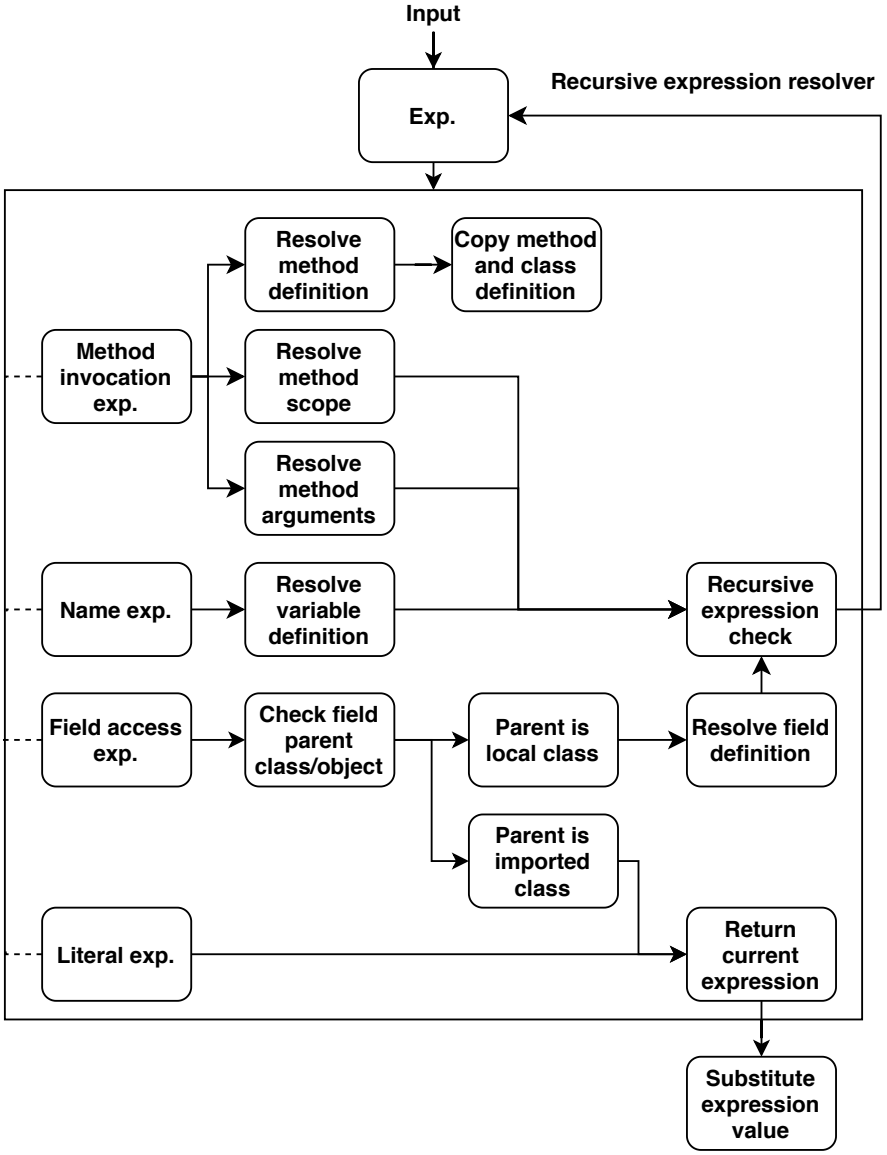

Fig. 10 Overview of the data flow analysis workflow

as the argument for the `setAudioAttributes` method invocation (line 10). The API invocation argument in this first example is an out-of-method variable, due to it being defined outside of the method containing the API invocation (defined in lines 2 and 3). The second example shows a code fragment where a complex expression is directly put as an argument of the second `setAudioAttributes(...)` method invocation (line 26). Contrary to the first

```
43  private int duration = 9;
44  private int frequency = 3;
45  public int amplitude = duration / frequency;
46  public VibrationEffect createVibration(int time, int amplitude) {
47     return VibrationEffect.createOneShot(time, amplitude);
48  }
...
69  public void onCreate() {
70      if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
71          vibrator.vibrate(createVibration(3, amplitude));
72      } else {
73          vibrator.vibrate(50);
74      }
75  }
```

Fig. 11 Update example code for `vibrate(long)` API

example, the argument used in this example is bound (i.e., argument that is locally defined or passed in via a method parameter), as the argument is defined in the API invocation argument. Given an update-example in the first form, AndroEvolve needs to resolve the value of the out-of-method variable used as the API invocation argument.

To alleviate this problem, AndroEvolve utilizes data flow analysis (DFA) to predict and resolve the values of any variable, method, or other Java expressions at any given point in the code. Hence, it can derive replacement values for any expressions used in the API invocation arguments that are only located within the update example file.

We made a custom lightweight DFA for this purpose by using the symbol resolver from Java Symbol Solver, which is a part of Javaparser (van Bruggen et al. 2020). This DFA conducts a bottom-up search from the bound variables or expressions used as the API method invocation arguments and expands the search scope until it finds the value or the method definition referred by the expressions, or until it explores the entire file. Using this approach, we can predict the value of out-of-method variables that are referred to by the variables used as the API invocation arguments. Values and method invocations that are found by this analysis are used to replace the original expressions. These replacements are done to ensure that the expressions used as API invocation arguments are in the form of literal expressions, static class members, method invocations, or object creations.

The workflow for this DFA is shown in Figure 10. The DFA receives as an input the expression to be resolved. This expression is used as an API invocation argument and can be in the form of any Java expressions. Each form of expression will require a specific processing as given in the workflow diagram. AndroEvolve runs its data flow analysis as a preprocessing step before the update script creation. As a preprocessing step, this feature does not change the internal working behavior of CocciEvolve but instead adds a layer of functionality that modifies the input code. To give a better understanding of this workflow, assume the example code provided in Figure 11.

In this update example, we can see that the updated `vibrate` API invocation uses a method invocation expression as its argument (line 71). According to the workflow, we resolve the method definition, `createVibration(...)` (line 46), and process it using the copy method and class definition to copy the method definition to the target file where the update will be applied. Then, we resolve the scope of the method. However, since `createVibration(...)` is a public method that can be referenced directly, no object or class is used in its invocation, thus resulting in no scope to resolve. Next, we resolve the arguments of the method invocation. The first argument, `3` is an integer literal expression, thus no replacement is needed. However, the second argument, `amplitude` is a variable, so we need to resolve its definition, producing `duration / frequency` (line 45). Since this definition contains expressions in the form of variables, we further resolve their values recursively. From this process, we found the literal expressions of `9` (line 43) for duration and `3` (line 44) for frequency. These expressions are used to replace their values in the `amplitude` variable definition in line 45 definition resulting in:

```
45 public int amplitude = 9 / 3;
```

In the end, we replace the second argument of `createVibration` with this definition of the `amplitude` variable in line 71, resulting in this updated API:

```
71 vibrator.vibrate(createVibration(3, 9 / 3));
```

It is possible that using the literal values of variables/constants as illustrated above may not be preferred due to the introduction of *magic numbers*. To alleviate this problem while keeping the version that use the literal values, we implement two approaches of variable replacement for AndroEvolve data flow analysis:

1. **AndroEvolve with simplified variable replacement:** AndroEvolve replaces variables/constants with their literal values;
2. **AndroEvolve with verbose variable replacement:** Based on the variables used in the updated API invocation, AndroEvolve introduces new variables in the update.

For AndroEvolve with verbose variable replacement, instead of using the resolved literal values, AndroEvolve will introduce them as new variables in the target file where AndroEvolve's update is applied. Using this approach, on the update example shown in Figure 11, AndroEvolve will keep track of the following variables and will add them as new variables in the target file:

```
43 private int duration = 9;
44 private int frequency = 3;
45 public int amplitude = duration / frequency
```

Note that AndroEvolve does not necessarily use the initialization values of public variables, as this is likely to introduce bugs if the values are updated. Rather, AndroEvolve will attempt to use the "latest" value assigned to the variable by performing a bottom-up analysis starting from the location of the

updated API usage. For this purpose, AndroEvolve takes into account all the lines preceding the updated API usage in the file. This means that if a value assigned to a public variable is changed, AndroEvolve will attempt to uncover this new value rather than simply using the variable initialization value. Still, AndroEvolve's heuristic may not cover all possible scenarios. Also note that when introducing a new variable to the target file in the update application process, AndroEvolve checks whether a variable with the same name already exists in the target file. If a variable with the same name already exists, AndroEvolve edits the name of the variable that it will introduce by appending a number to its name (e.g., `int currentTime` → `int currentTime_1`). This is done to ensure that there are no variables with duplicate names in the updated target file, which may introduce errors.

In the evaluation of AndroEvolve, the correctness of its produced updated code is determined by checking it against manually constructed updated code. We did not find any errors related to the updating of public variables.

3.2 Source File Normalization

Following the approach taken by CocciEvolve, AndroEvolve also uses source file normalization in its workflow. Source file normalization is used to mitigate the problem of semantically-equivalent code being expressed in different forms, which can cause a failed API update. Source file normalization in AndroEvolve is focused on the part of the file that contains the API invocations defined in the API update mapping, along with their arguments. Given an API invocation, source file normalization normalizes the code in three steps:

1. Extract a deprecated API invocation located within a compound expression or statement into a variable assignment.
2. Extract the arguments of the deprecated API invocation into separate variable assignments.
3. Extract the returned value of the deprecated API invocation into a variable assignment.

Consider a code fragment containing a `vibrate(...)` API invocation as shown in the before normalization example of Figure 12. Source file normalization will convert the code given in the before normalization example into the normalized form given in the after normalization example. First, all arguments, including the class or object used in the API invocation, are extracted. This extraction introduces the variables `parameterVariable0` (line 12), containing the argument of the API invocation, and `classNameVariable` (line 13), containing the class used in the API invocation. These temporary variables refer to the original parameter and class/object used in the API invocation, and replace their usages in the API invocation (line 14).

```
1  // Before normalization
2  public void Once(long milliseconds) {
3      MyVibrator = (android.os.Vibrator) MyActivity.getSystemService(
4              android.content.Context.VIBRATOR_SERVICE);
5      MyVibrator.vibrate(milliseconds);
6  }
7
8  // After normalization
9  public void Once(long milliseconds) {
10     MyVibrator = (android.os.Vibrator) MyActivity.getSystemService(
11             android.content.Context.VIBRATOR_SERVICE);
12     long parameterVariable0 = milliseconds;
13     Vibrator classNameVariable = MyVibrator;
14     classNameVariable.vibrate(parameterVariable0);
15 }
```

Fig. 12 An illustration of the source file normalization result

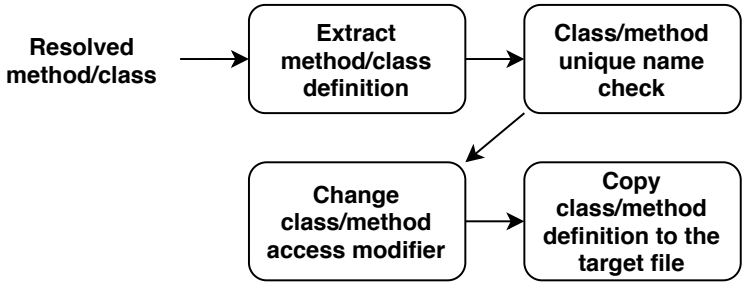

Fig. 13 Overview of the copy method and class definition workflow

3.3 Copying Method and Class Definition

To provide a correct update, substituting the expressions into the resolved value is insufficient if the expressions are in the form of method invocations or object instantiations. This is because these expressions require the associated method and class definitions. Accordingly, we must copy the method and class definitions used by the resolved expression from the update example.

There are several important points to be considered for this feature. First, the copied class or method should be defined within the file containing the after-update example. This is due to AndroEvolve's limitation as a tool that works on a file scope. Therefore, if the class or method is defined outside of the after-update example file, AndroEvolve will not be able to resolve them. Second, the copied class or method must be given an unique name, as required by Java. Lastly, the class and method that is copied must be placed in a scope that is accessible by the API invocation in the target file.

The workflow of this feature can be seen in Figure 13. First we extract the definitions of the methods and classes referenced by the expression from the code. This extraction process is done recursively, extracting all the methods and classes including those that are being used or accessed by a previously extracted class or method. Therefore, if the body of an extracted class accesses another class's methods or fields, the other class will also be copied to ensure

```
1  public class AudioManager {
2      public void requestAudio() {
3          if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
4              AudioFocusRequest request = audioFocusRequestOreo.getAudioFocusRequest();
5              result = audioManager.requestAudioFocus(request);
6          } else {
7              result = audioManager.requestAudioFocus(listener, type, duration);
8          }
9      }
10 }
```

Fig. 14 Updated code example for `requestAudioFocus()` deprecated API containing an unresolved variable

that they can be accessed. Following the extraction process, we make sure that in the target file, there is no class or method with the same name as the extracted class and method. If a duplicate name is detected, the copied class or method name is modified by adding a number to the name. After validating the name, we then modify the access modifier of the class and method to `public` to make sure that the API invocation in the target file can access them. Finally, we copy the class and method definition to the end of the target file.

As an illustration, consider the updated file shown in Figure 14. Lines 3–8 show an example for `requestAudioFocus(...)` API. The updated API uses an `AudioFocusRequestOreo` object (line 4–5) as an argument but this class is not defined in the target file. To correct the update, we must resolve and copy the relevant method and class definition, and instantiate the `AudioFocusRequestOreo` object. The resulting updated code is shown in Figure 15. Lines 4–5 show the resolved variable values, which are obtained by creating an `AudioFocusRequestOreo` object and using the object to invoke the `getAudioFocusRequest` method. The relevant class and method are also copied into the updated code (lines 20–36).

3.4 Variable Name Denormalization

Addressing the problem of temporary variables introduced during the update process improves the readability and ease of understanding of the updated code. For this purpose, AndroEvolve uses variable name denormalization to remove unnecessary temporary variables and replace them with their values or referred variables. Consider the updated code containing temporary variables shown in Figure 16. In lines 11–17, the temporary variables clutter the update result since they only refer to other existing variables. Through the application of variable name denormalization in this example, AndroEvolve removes the temporary variables and replaces them with their relevant values. This results in a shorter, more concise, and more understandable code, as can be seen in Figure 17.

The steps in the denormalization process are shown in Figure 18. In Step (1), AndroEvolve matches the signatures of the deprecated and updated API with the signatures of the method invocations detected in the target file to

```
1  public class AudioManager {
2      public void requestAudio() {
3          if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
4              AudioFocusRequest request = new AudioFocusRequestOreo(this).
                       getAudioFocusRequest();
5              result = audioManager.requestAudioFocus(request);
6          } else {
7              result = audioManager.requestAudioFocus(listener, type, duration);
8          }
9      }
10 }
   ...
20 class AudioFocusRequestOreo {
21     private AudioFocusRequest audioFocusRequest;
22     public AudioFocusRequestOreo(AudioManager.OnAudioFocusChangeListener listener) {
           ...
30     }
31     public AudioFocusRequest getAudioFocusRequest() {
           ...
35     }
36 }
```

Fig. 15 Updated code with resolved variable, method, and class definition for the code shown in Figure 14

```
   ...
10 public int saveLayer(float left, float top, float right, float bottom,
           Paint paint, int saveFlags) {
11     float parameterVariable0 = left;
12     float parameterVariable1 = top;
13     float parameterVariable2 = right;
14     float parameterVariable3 = bottom;
15     Paint parameterVariable4 = paint;
16     int parameterVariable5 = saveFlags;
17     Canvas classNameVariable = mCanvas;
18     int tempFunctionReturnValue;
19     if (VERSION.SDK_INT >= 21) {
20         tempFunctionReturnValue = classNameVariable.saveLayer(parameterVariable0,
               parameterVariable1, parameterVariable2,
               parameterVariable3, parameterVariable4);
21     } else {
22         tempFunctionReturnValue = classNameVariable.saveLayer(parameterVariable0,
               parameterVariable1, parameterVariable2, parameterVariable3,
               parameterVariable4, parameterVariable5);
23     }
24 }
   ...
```

Fig. 16 Example update result for deprecated API `saveLayer`

find the locations where the denormalization should be applied. In Step (2), for each of the found API invocations, AndroEvolve identifies whether the API invocation uses temporary variables as its parameters. This is done by checking whether each of the parameter names contain the prefix "parameterVariable", since the result of the variable normalization done by AndroEvolve only introduces new parameters with this specific prefix. If the parameter is found to be a temporary variable, a bottom-up search to find the definition for the temporary variable is conducted from the API invocation position. This search is done by matching the name of any found variable with the name used

```
   ...
10 public int saveLayer(float left, float top, float right, float bottom,
          Paint paint, int saveFlags) {
11      int tempFunctionReturnValue;
12      if (VERSION.SDK_INT >= 21) {
13          tempFunctionReturnValue = mCanvas.saveLayer(left, top, right, bottom, paint);
14      } else {
15          tempFunctionReturnValue = mCanvas.saveLayer(left, top, right, bottom,
              paint, saveFlags);
16      }
17 }
   ...
```

Fig. 17 Denormalized code of the one shown in Figure 16

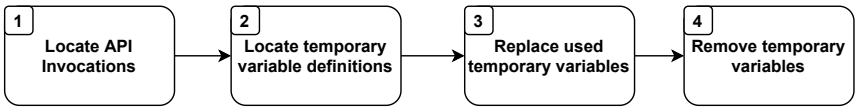

Fig. 18 Overview of the variable denormalization workflow

by the identified temporary variable. Then, in Step (3), AndroEvolve replaces the usages of the temporary variables with their corresponding definitions. Finally, in Step (4), the declarations and definitions of the temporary variables are deleted from the code as they are no longer needed.

4 Experiment

We conduct an experiment to evaluate the performance of AndroEvolve and compare it to the current state-of-the-art Android API automated update tool, CocciEvolve. First, we define the research questions that we aim to answer. Then, we describe the dataset that we use. Finally, we report the results and findings from the experiment.

4.1 Research Questions

4.1.1 RQ1: How many updates can AndroEvolve apply correctly?

We assess update accuracy by counting the number of correct updates produced. A correct update is an update that contains the deprecated and replacement API method in the form of an `if` code block, alongside all the methods and classes needed by the replacement API method. We compare the update accuracy of AndroEvolve and CocciEvolve. We also ask an experienced Java and Android developer, who is not an author of this work, to check the measured update accuracy and verify the correctness of the updated code. Through this assessment, we attempt to get a better understanding of the performance of AndroEvolve on a real-world dataset, and how it compares to the previous state-of-the-art tool.

For the validation conducted by the experienced developers, several steps are taken. First, the developers are asked to parse the updated file (e.g., using JavaParser) to check if it is syntactically correct. Then, the developers are asked to read and study the specification of the deprecated and updated APIs to ensure that they understand the API and how the update should be performed. The developers are also given the after-update example to be used as a guide. Afterwards, we ask the developers to check whether the update is applied correctly in the updated target file, which must contain the following: (1) the `if` statement containing the Android version check; (2) the deprecated API invocation; (3) the updated API invocation; and (4) new parameters that are used by the updated API. Finally, the developers should compare the original target file and the updated target file to check whether there are any unwanted differences.

4.1.2 RQ2: How readable is the updated code produced by CocciEvolve and AndroEvolve?

We measure and compare the readability of the updated code produced by AndroEvolve with simplified variable replacement (AndroEvolve simplified), AndroEvolve with verbose variable replacement (AndroEvolve verbose), and CocciEvolve. Variable name denormalization is applied to both AndroEvolve's updated code. Through this comparison, we investigate if the different variable replacement approaches and the variable name denormalization used in AndroEvolve affects code readability. In order to get a better insight into the readability of the update, we conduct an automatic and a manual measurement.

In the automatic measurement, we utilize a state-of-the-art code readability scoring tool proposed by Scalabrino et al. (2018). The tool outputs a code readability score in a scale of 0.0 to 1.0 with a higher score indicating better readability. As the readability score is affected by the length of the source code file, we performed a static slicing to obtain the parts of the code that are affected by the update. The static code slicing is done using JavaParser (van Bruggen et al. 2020) by first locating the deprecated and updated Android APIs based on their API signatures. After these APIs are found, we extract the API method invocations and all the variables that are used in the invocations. The sliced code is then put into a artificial class and method to allow readability measurement using the tool. An example of the generated file is shown in Figure 19.

In the manual measurement, we ask two experienced Android developers, who are not authors of this work, to score 90 examples of updated code, with 30 examples each from CocciEvolve and AndroEvolve. They are not told which tool is used to produce the updated code. The developers are also not aware of the behaviors of AndroEvolve and CocciEvolve. The first developer has 5 years of experience with Android, while the second developer has 3 years of experience. For each updated code example, the developers are asked to

```
class MainActivity {
    public static void main() {
        int parameterVariable0 = lastHour + 1;
        TimePicker classNameVariable = timePicker;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            classNameVariable.setHour(parameterVariable0);
        } else {
            classNameVariable.setCurrentHour(parameterVariable0);
        }
    }
}
```

Fig. 19 Sliced code example for deprecated `setCurrentHour(...)` API

thoroughly review the code and provide a score on the Likert scale of 1–5 for the readability of the code with the following score descriptions:

1. Hard to read and understand the code
2. Slightly hard to read and understand the code
3. Indifferent. The code is neither easy nor hard to understand.
4. Slightly easy to read and understand the code
5. Easy to read and understand the code

Similarly, the developers are also asked to give a score on the Likert scale of 1–5 for the naturalness of the code with the following score descriptions:

1. Very confident that the code is not made by human
2. Slightly confident that the code is not made by human
3. Unsure whether the code is made by human or not
4. Slightly confident that the code is made by human
5. Very confident that the code is made by human

A higher score in the readability aspect indicates higher readability of the code, while a higher score in the naturalness aspect indicates higher confidence that the code resembles a piece of code that is produced by manually. For each pair of updated code examples, the developers are also asked to determine which code that they prefer.

4.1.3 RQ3: How efficient is AndroEvolve in producing updates?

We measure the time needed for AndroEvolve to update a target file. Specifically, we measure the time to perform update script creation and update application. The measurement is conducted in a MacBook Pro with a 2.3 GHz Intel Core i5-7360U processor, and 8 GB 2133 MHz random access memory. The system runs Java SE 11, with OpenJDK version 11.0.6. We run AndroEvolve in a Docker container running Debian GNU/Linux 10 (buster) environment that is not working or running other task. For each of the considered APIs, we run our experiment 30 times and measure the average and median time taken for the update script creation and update application. Through this time measurement, we evaluate how fast the update process is done by AndroEvolve and the variation of time required between different APIs.

4.2 Dataset

Our dataset comprises after-update examples, target files to update that contain usages of the deprecated APIs, and API mappings from the deprecated APIs to the replacement APIs. In our dataset, a deprecated API is one that is marked as such in the Android SDK documentation. We consider the 20 deprecated APIs that were used in the evaluation of both CocciEvolve (Haryono et al. 2020) and AppEvolve (Fazzini et al. 2019). These APIs are collected from three different Android API levels (23, 24, 26) and have one-to-one mappings (i.e., a deprecated API is replaced with a single updated API) except for the `getAllNetworkInfo` deprecated API. The three API levels correspond to Android OS version 7.0 (Nougat), 7.1 (Nougat), and 8.0 (Oreo). For the after-update examples, we used the same set of 19 after-update examples used in the evaluation of AppEvolve. They consist of the after-update examples for the 20 considered APIs aside from the `shouldOverrideUrlLoading` API which after-update example is not found.

In our evaluation, we aim to replicate the real-life distributions of the considered APIs; specifically, a more popular API should be represented by more target files in our evaluation dataset. The target files are collected from randomly selected GitHub repositories obtained using AUSearch (Asyrofi et al. 2020), a tool to search Github repositories for Android API usages. AUSearch detects and locates whether each project contains invocations of the deprecated APIs. For each project, we only consider its latest version, as currently AUSearch is limited to only collecting API usage examples from the latest commit of a repository. From the 20 considered APIs, a total of 12,008 files are collected. From these 12,008 files, we perform a stratified random sampling, and for each API (each stratum), we collect a statistically representative sample (at 95% confidence level and 5% margin of error).[1] We end up with a collection of 372 target files for the 20 APIs. The detailed statistics of the target files are shown in Table 1.

We also analyzed the repositories from which the target files are retrieved. We found a total of 311 distinct repositories from which the target files are taken. For each repository, there are 1 to 6 target files. For the 311 repositories, the average number of stars is 321 and the average number of commits is 18,991. Thus, our target files come from various repositories including those that are large and popular.

[1] We use the Sample Size Calculator from the Australia Bureau of Statistics that supports stratified random sampling. The calculator is available from: https://www.abs.gov.au/websitedbs/D3310114.nsf/Home/Sample+Size+Calculator+Stratification+Examples

Table 1 Number of targets in our evaluation dataset

API Level	Class Name	Deprecated API	Updated API	#File
23	android.app.Notification. Builder	addAction(int, charSequence, PendingIntent)	addAction(Action)	2
23	android.net.ConnectivityManager	getAllNetworkInfo()	getAllNetworks(), getNetwork-Info(Network)	29
23	android.widget.TimePicker	getCurrentHour()	getHour()	30
23	android.widget.TimePicker	getCurrentMinute()	getMinute()	30
23	android.widget.TimePicker	setCurrentHour(Integer)	setHour(int)	30
23	android.widget.TimePicker	setCurrentMinute(Integer)	setMinute(int)	30
23	android.widget.TextView	setTextAppearance (Context, int)	setTextAppearance (int)	10
24	android.location.LocationManager	addGpsStatusListener (Listener)	registerGnssStatus-Callback(Callback)	21
24	android.text.Html	fromHtml(String)	fromHtml(String, int)	23
24	android.content.ContentProviderClient	release()	close()	4
24	android.location.LocationManager	removeGpsStatusListener(Listener)	unregisterGnssStatus-Callback(Callback)	21
24	android.webkit.WebViewClient	shouldOverrideUrl-Loading(WebView, String)	shouldOverrideUrl-Loading(WebView, WebResourceRequest)	1
24	android.view.View	startDrag(ClipData, DragShadowBuilder, Object, int)	startDragAndDrop (ClipData, DragShadowBuilder, Object, int)	2
26	android.media.AudioManager	abandonAudioFocus (OnAudioFocus-ChangeListener)	abandonAudioFocus-Request(AudioFocus-Request)	25
26	android.telephony.TelephonyManager	getDeviceId()	getImei()	24
26	android.media.AudioManager	requestAudioFocus (OnAudioFocusChange-Listener, int, int)	requestAudioFocus (AudioFocusRequest)	28
26	android.graphics.Canvas	saveLayer(float, float, float, float, Paint, int)	saveLayer(float, float, float, float, Paint)	17
26	android.media.MediaPlayer	setAudioStreamType (int)	setAudioAttributes (AudioAttributes)	5
26	android.os.Vibrator	vibrate(long)	vibrate(Vibration-Effect)	21
26	android.os.Vibrator	vibrate(long[], int)	vibrate(Vibration-Effect)	19

Table 2 Detailed AndroEvolve update results for each API

API Description	# API Usages	# Correct Update	# False Update
addAction(...)	3	0	3
getAllNetworkInfo()	29	0	29
getCurrentHour()	48	45	3
getCurrentMinute()	55	51	4
setCurrentHour(...)	68	68	0
setCurrentMinute()	57	57	0
setTextAppearance(...)	12	12	0
addGpsStatusListener(...)	21	0	21
fromHtml(...)	72	70	2
release()	4	4	0
removeGpsStatusListener(...)	21	0	21
shouldOverrideUrlLoading(...)	1	0	1
startDrag(...)	2	2	0
abandonAudioFocus(...)	32	32	0
getDeviceId()	31	31	0
requestAudioFocus(...)	41	41	0
saveLayer(...)	22	22	0
setAudioStreamType(...)	5	5	0
vibrate(long)	22	22	0
vibrate(long[], int)	19	19	0
Total	**565**	**481**	**84**

4.3 Results

4.3.1 RQ1: Code Update Accuracy

From the 372 target files, a total of 565 deprecated API invocations are found. AndroEvolve and CocciEvolve correctly updated 481 and 362 out of the 565 deprecated API usages, respectively. Only a small number of cases cannot be updated successfully by AndroEvolve, amounting to 84 out of 565 deprecated API usages. AndroEvolve outperforms CocciEvolve by 32.9%. We also conduct a statistical analysis using Fisher's exact test (Upton 1992) to check whether the differences between AndroEvolve and CocciEvolve are significant. We found that the two-tailed P value is less than 0.0001, indicating that the difference between the effectiveness of the two tools is significant. Analysis of the results shows that the inclusion of data flow analysis improves the update results of AndroEvolve significantly, specifically in the `vibrate(long)`, `vibrate(long[],int)`, and `requestAudioFocus(...)` APIs. After-update examples for these APIs include usages of out-of-method variables that are not handled by CocciEvolve. AndroEvolve has similar performance as CocciEvolve for APIs that do not use out-of-method variables in the after-update example.

The update results for each API are shown in Table 2. At the API level, we can see that 11 APIs are updated 100% correctly by AndroEvolve. Instances of `getCurrentHour()` and `getCurrentMinute()` are mostly updated correctly, with only a small number of cases having incomplete update. Meanwhile the

Table 3 Updated code automated readability scores

API	# Code	Average Score		
		AndroEvo Simpified	**AndroEvo Verbose**	**Cocci Evolve**
addAction(...)	3	0	0	0
getAllNetworkInfo()	29	0	0	0
getCurrentHour()	48	0.6009	0.6009	0.5071
getCurrentMinute()	55	0.5996	0.5996	0.5172
setCurrentHour(...)	68	0.6006	0.6006	0.3904
setCurrentMinute(...)	57	0.5766	0.5766	0.3962
setTextAppearance(...)	12	0.5615	0.5615	0.2947
addGpsStatusListener(...)	21	0	0	0
fromHtml(...)	72	0.4143	0.4143	0.2593
release()	4	0.8311	0.8311	0.6890
removeGpsStatusListener(...)	21	0	0	0
shouldOverrideUrlLoading(...)	1	0	0	0
startDrag(...)	2	0.4516	0.4516	0.1440
abandonAudioFocus(...)	32	0.2511	0.3297	0.2257
getDeviceId()	31	0.4545	0.4545	0.3974
requestAudioFocus(...)	41	0.2462	0.2595	0.2341
saveLayer(...)	22	0.4115	0.4115	0.1011
setAudioStreamType(...)	5	0.5082	0.4342	0.3238
vibrate(...)	22	0.4151	0.3524	0.2432
vibrate(..., ...)	19	0.5536	0.5348	0.3728

APIs `addAction(...)`, `getAllNetworkInfo(...)`, `addGpsStatusListener(...)`, and `removeGpsStatusListener(...)`, cannot to be updated due to failure in the update patch creation process due to the usage of out-of-file variables.

4.3.2 RQ2: Code Readability

To understand the readability of the resulting code, we used a readability scoring tool proposed by Scalabrino et al. (2018) to automatically compute the readability scores for all updated code fragments and average the scores for each API. The resulting averages are shown in Table 3. Code updated by AndroEvolve has higher scores for all considered APIs. Further analysis of the updated code shows that a bigger improvement is observed for APIs with multiple arguments (e.g. `saveLayer(...)`, `startDrag(...)`, etc.). Comparing the readability scores, we find that for APIs with addition of a single literal or method invocation as new parameter (e.g., `vibrate(...)`, `setAudioStreamType(...)`), AndroEvolve with simplified variable replacement (AndroEvolve simplified) has a higher average score. Meanwhile, for APIs where there are additions of two or more literals or method invocations in the API parameter (e.g., `requestAudioFocus(...)`, `abandonAudioFocus(...)`), AndroEvolve has a higher readability using verbose variable replacement (AndroEvolve verbose).

The manual readability measurement strengthens the above findings. The average score given by the developers for code updated by AndroEvolve simpli-

fied is 4.5 with a median score of 5.0. The average score given by the developers for code updated by AndroEvolve verbose is 4.633 with a median score of 5.0. The average score given by the developers for code updated by CocciEvolve is 1.133 with a median score of 1.0. Improvement can also be seen in the scores for code naturalness: Updated code produced by AndroEvolve simplified received an average score of 4.533 with a median score of 5.0. Updated code produced by AndroEvolve verbose received an average score of 4.667 with a median score of 5.0. Meanwhile, updated code produced by CocciEvolve only received an average score of 1.10 with a median score of 1.0. We also calculated the IRR (inter-rater reliability) for both the readability and the naturalness of the code. To calculate the IRR, we use weighted Cohen's kappa score (Cohen 1968), which is recommended when the ratings are ordered (Bakeman and Gottman 1986) – in our case, ratings are in the range of 1 to 5, where 1 is closer to 2 than to 5. For the code readability ratings, the two developers weighted Cohen's kappa score is 0.684. For the code naturalness ratings, the two developers weighted Cohen's kappa score is 0.720. Following the interpretation of Cohen's Kappa score proposed by Landis and Koch (1977),[2] these Cohen's Kappa scores indicate that there is substantial agreement between the two evaluators in terms of their code readability and code naturalness ratings.

4.3.3 RQ3: Update time

The average and median time of AndroEvolve's update script creation and update application processes are shown in Table 4. Both the update-script creation and update-script application steps in AndroEvolve took an average of less than 15 seconds to execute. Across the various APIs, the distribution of time needed for the update application process has a wider variation than that for the update script creation process. The amount of time needed for the update application process is influenced by the complexity of the required update and the number of invocations of the deprecated API within the target file. Still, for all APIs, given an after-update code example, a target file, and an API update mapping, AndroEvolve can update the API usages in the target file in less than 15 seconds. When processing multiple files containing the same API usage, the time can be further shortened by using the same update script.

4.4 AndroEvolve's Limitations

To get a better understanding of the current state of AndroEvolve, we analyze the cases where it failed to create successful update. We found the following reasons :

Update requiring inter-procedural data flow analysis. AndroEvolve performs its data flow analysis within the file scope and does not consider other

[2] Landis and Koch (1977) define a kappa score in the range of 0.00-0.20 as slight agreement, 0.21-0.40 as fair agreement, 0.41-0.60 as moderate agreement, 0.61 and 0.80 as substantial agreement, and 0.81-1.00 as almost perfect agreement, respectively.

Table 4 Time measurement results of AndroEvolve (in seconds)

API	Update Creation		Update Application	
	Median	**Average**	**Median**	**Average**
addAction(...)	8.174	8.167	3.609	3.606
getAllNetworkInfo()	-	-	-	-
getCurrentHour()	7.452	7.448	4.005	3.406
getCurrentMinute()	7.409	7.396	4.196	4.326
setCurrentHour(Integer)	6.718	6.716	3.704	3.546
setCurrentMinute(Integer)	6.736	6.726	3.504	3.527
setTextAppearance(...)	6.689	6.674	3.597	3.357
addGpsStatusListener(...)	8.168	8.165	3.529	3.428
fromHtml(...)	7.494	7.493	3.989	3.719
release()	8.824	8.817	3.822	3.740
removeGpsStatusListener(...)	-	-	-	-
shouldOverrideUrlLoading(...)	-	-	-	-
startDrag(...)	8.425	8.394	3.604	3.595
abandonAudioFocus(...)	10.410	10.394	4.009	3.766
getDeviceId()	8.177	8.144	3.958	3.714
requestAudioFocus(...)	7.291	7.284	4.058	3.747
saveLayer(...)	7.829	7.817	3.849	3.721
setAudioStreamType(...)	7.106	7.093	3.133	3.038
vibrate(long)	6.730	6.723	3.166	3.045
vibrate(long[], int)	6.928	6.915	3.457	3.321

files that may be used by the API invocation (e.g., public methods and fields of imported classes). This may cause the generation of incorrect update patches. We analyze the prevalence of such cases in our dataset. And we find that from the 565 API usages in our dataset, 481 of them (85.1%) are correctly updated by AndroEvolve. We manually analyze the remaining 84 API usages and find that for 45 of them – which amount to 8.0% of the API usages in our dataset – inter-procedural data flow analysis is needed to generate correct update patches.

Variables defined in the event listener. Android applications may utilize event listeners, which are methods that will be called by the Android framework based on the user interaction with the application interface. An example of this event listener is the `onClick()` callback method, which will be triggered when a user touches an item in the application interface. AndroEvolve's data flow analysis is unable to resolve the values of variables that are changed through this event listener as they are dependent on user inputs. As an example, let us consider the after-update example code shown in Figure 20. In this example, the updated API invocation at line 12 uses the `AMPLITUDE` variable. However, at lines 5 – 7 of the code, an event listener which will change the value of the `AMPLITUDE` variable if the button is pressed is defined. AndroEvolve will not detect the value change of the `AMPLITUDE` variable that occurs in this method and instead will resolve the value of the variable to `AMPLITUDE = 20` based on its definition in line 3. While we find no such case in our evaluation, it may occur in other Android API migrations that we have yet to consider.

```
1  public final class AppVibrator {
2      public static int DURATION = 10;
3      public static int AMPLITUDE = 20;
4
5      public void onClickBtn(View v) {
6          AMPLITUDE = 200;
7      }
8
9      public static void itemActivated(Context context) {
10         Vibrator vibrator = context.getSystemService(Context.VIBRATOR_SERVICE);
11         if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
12             vibrator.vibrate(VibrationEffect.createOneShot(DURATION, AMPLITUDE));
13         } else {
14             vibrator.vibrate(DURATION);
15         }
16     }
17  }
```

Fig. 20 An example of a variable assignment in an event listener

Variables with values that cannot be determined by AndroEvolve's analysis. To generate a semantic patch, AndroEvolve's data flow analysis tries to resolve the variables that appear in the after-update example to their values. If AndroEvolve's analysis fails to determine the value of a variable (e.g., if a variable value depends on the users' input), AndroEvolve will terminate its data-flow analysis and will not replace variables with concrete values. As the patches generated by AndroEvolve may not be perfect, developers may need to manually make further modifications. Developers can make the necessary changes either to the semantic patch (if the variable can be resolved to an expression that can generalize to all locations) or to the individual locations that are updated by the semantic patch (if the variable needs to be resolved to different expressions at different locations, e.g., the relevant user input is given using a button widget for some cases and a textbox widget for some other cases, etc.).

1:N API update. AndroEvolve cannot handle updates that involve an update of a single API into multiple APIs, such as updating the `getAllNetworkInfo()` API to the `getAllNetworks()` and `getNetworkInfo(...)` APIs. Updating this API involves adding a new control flow in the form of a loop that iterates over the `Network` object returned by `getAllNetworks()` to receive the corresponding `NetworkInfo` object by using `getNetworkInfo(...)` method.

Multiple API invocation in a single line. AndroEvolve also cannot update multiple invocations of an API method written in a single line of code. This problem occurs in the update process of `getCurrentHour()`, and `getCurrentMinute()` APIs. While uncommon, this problem has been found in some target files, resulting in an incomplete update. An example of this problem is shown in Figure 21. We can see that `getCurrentHour()` (line 7) is invoked multiple times in the last line, causing only the first invocation to be updated. This problem occurs due to the current limitation of Coccinelle4J which is the transformation tool used by AndroEvolve. When applying a trans-

```
1   int tempFunctionReturnValue;
2   if (android.os.Build.VERSION.SDK_INT >= 23) {
3       tempFunctionReturnValue = timePickerBegin.getHour();
4   } else {
5       tempFunctionReturnValue = timePickerBegin.getCurrentHour();
6   }
7   dateTime = tempFunctionReturnValue + ":" + timePickerBegin.getCurrentMinute() + "-" +
          timePickerEnd.getCurrentHour() + ":";
```

Fig. 21 An example of multiple invocations of `getCurrentHour()` method in a single line

formation to multiple of the same program element within a single line of code, Coccinelle4J will only transform the first program element.

5 Related Work

API deprecation. Studies about API deprecation have been done frequently (Brito et al. 2016; Fazzini et al. 2019; Haryono et al. 2020; Hora et al. 2015; Li et al. 2018b; Robbes et al. 2012; Sawant et al. 2018; Yang et al. 2018; Zhou and Walker 2016). Li et al. (2018b) proposed a tool called CDA to characterize deprecated Android APIs. They found inconsistent annotation and documentation on deprecated APIs, and that most deprecated APIs are used in popular libraries. Zhou and Walker (2016) examined the usages of deprecated APIs in 26 open-source Java frameworks and libraries and found that many of these APIs were never updated. They proposed Deprecation Watcher, a tool to detect deprecated Android API usages from code examples on the web. Brito et al. (2016) conducted a large scale analysis on Java systems to measure the usages of deprecation messages. Their analysis showed that a number of deprecated APIs did not use these replacement messages. Yang et al. (2018) investigated the impact of Android OS updates on Android apps. They presented an automatic approach to detect parts of Android apps affected by an OS update.

Some studies focus on the effect of API deprecation (Hora et al. 2015; Robbes et al. 2012; Sawant et al. 2018). Robbes et al. (2012) conducted a case study on the Smalltalk ecosystem and found that API deprecation messages are not always helpful. Hora et al. (2015) conducted a case study on the Pharo ecosystem on the impact of API evolution, finding that API changes can have a large impact in the client systems, methods, and developers. They also found that API replacements can not be resolved uniformly. Sawant et al. (2018) replicated the study on Java. They found that only a small fraction of developers react to API deprecation and most of these developers prefer to remove usages of deprecated APIs rather than migrating them to the updated APIs.

Our study also deals with API deprecation. It focuses on automatically updating the usages of deprecated Android APIs. Similar studies on this topic have been done recently. AppEvolve (Fazzini et al. 2019) is one of the first tools proposed for this purpose. It performs API updates by learning from

both before- and after- update example. CocciEvolve (Haryono et al. 2020) is the current state-of-the-art tool for automated update of deprecated Android API usage. CocciEvolve improves on AppEvolve by only using an after-update example to perform API update and providing a highly readable and configurable update API update script in the form of semantic patches. CocciEvolve also solves the problem of failure to update code with form or coding style differences between the update example and the target file which occurs in AppEvolve, as highlighted by the replication study by Thung et al. (2020). To mitigate this problem, CocciEvolve utilizes source file normalization to normalize the deprecated and updated API invocations in the after-update example and in the target code.

Program transformation. Program transformations have been studied extensively (Brunel et al. 2009; Jacobellis et al. 2013; Kang et al. 2019; Lawall and Muller 2018; Meng et al. 2013; Rolim et al. 2017; Visser 2001). Stratego (Visser 2001) is a language for program transformation based on the paradigm of rewriting strategies. Stratego performed transformation following the written transformation rules. LASE (Jacobellis et al. 2013; Meng et al. 2013) is an example based program transformation tool that is capable of locating and applying systematic edits. LASE provides users with a view of the syntactic edit and its corresponding contexts, allowing users to review and correct the edit suggestions. Rolim et al. (2017) proposed REFAZER, a technique for automatically learn program transformations by observing code edits performed by developers.

Coccinelle (Brunel et al. 2009; Lawall and Muller 2018) is a C-based program matching and transformation tool that has been utilized for the automated evolution of Linux kernel. Coccinelle allows developers to write their transformation rules using Semantic Patch Language (SmPL). Recently, Kang et al. (2019) proposed Coccinelle4J, a port of Coccinelle for the Java language. Coccinelle4J allows the transformation of Java program using the same method as Coccinelle, through the use of semantic patches written in the Semantic Patch Language. In our work, AndroEvolve applies program transformation to update deprecated Android API usages. It uses SmPL to write the transformations and Coccinelle4J to apply them.

6 Threats to Validity

External Validity. Threats to external validity relate to the distribution of APIs in our evaluation dataset; 19 have a one-to-one mapping (i.e., an API `x` can be replaced with a single new API `y`), while only one of them does not. One may wonder why a balanced dataset (i.e., half one-to-one mappings and half one-to-many/many-to-one mappings) is not used instead. To investigate this concern, we have conducted an empirical study on the API migration changes between Android API level 22 and 28 by manually checking their API difference reports (e.g. https://developer.android.com/sdk/api_diff/23/changes). From our analysis, we find a total of 328 deprecated APIs with only 6 (1.8%)

of them having one-to-many mappings. Because of this, we believe that the current selection of APIs considered in the evaluation – 19 (95%) correspond to one-to-one mappings, and 1 (5%) corresponds to one-to-many mappings – is reasonable, as it closely follows the distribution of migrations needed for updating deprecated APIs in Android.

Another concern related to external validity is the fact that we only consider 20 APIs in our evaluation. The 20 APIs were originally picked by the authors of AppEvolve (Fazzini et al. 2019). In many past studies, evaluation datasets are often reused to provide a fair evaluation and comparison between tools (Cui et al. 2020; Kim et al. 2019; Zhang et al. 2019). Thus, in this work, we also reuse the evaluation dataset of the prior work. It is unclear how the authors of AppEvolve picked these APIs. However, as the authors of AppEvolve are not the authors of this paper, the selection of the APIs are not cherry-picked to unfairly demonstrate the strength of AndroEvolve as compared to AppEvolve.

Another threat to external validity is related to the variable replacement step. We have implemented two approaches of variable replacement, simplified variable replacement, and verbose variable replacement. Simplified variable replacement uses the literal values of the variables, while verbose variable replacement introduces new variables in the update. In our experiments, we find that AndroEvolve with verbose variable replacement produces more readable code. However, it is possible that AndroEvolve with simplified variable replacement produces updated code that is equally readable on code that is not part of our experiments. Nevertheless, as our experiments on 372 target files indicate that AndroEvolve with verbose variable replacement produces more readable code, we believe that this threat is minimal.

Another concern is related to the code readability evaluation of AndroEvolve. We have conducted extensive evaluations on code readability by comparing the readability between code updated with AndroEvolve and CocciEvolve. However, it is not known whether the readability of AndroEvolve updated code is better than the ones produced by AppEvolve. To mitigate this threat, we have compared the readability of the updated code that is produced by AndroEvolve with simplified variable replacement (AndroEvolve simplified) and AppEvolve. As AppEvolve requires extensive manual configuration to update deprecated API usages within a project, we conduct a partial evaluation using 20 files used in the evaluation of CocciEvolve (Haryono et al. 2020) that have been configured to work with the AppEvolve. We measure readability using the same automatic readability measurement method that is described in Section 4.1.2. From the 20 files that are measured, AndroEvolve simplified achieves an average readability score of 0.6919 with a median score of 0.7370. Meanwhile, AppEvolve achieves an average readability score of 0.5379 with a median score of 0.5611. We also find that for each of the 20 considered files, AndroEvolve simplified has a higher readability score. Considering this result, we believe that there is limited threat to external validity for this potential issue.

Internal Validity. Threats to internal validity relate to the target files that we use in our evaluation. One concern is that the API levels (aka. SDK versions) that we consider in our evaluation (i.e., 23, 24, and 26) might be outside the range of API levels targeted by the apps. To investigate this potential threat, we have manually checked the listed `targetSdkVersion` and `minSdkVersion` in the 311 projects' manifest files. And we find that for 81.7% of the projects, the API levels that we consider (23, 24, and 26) are in between the declared `minSdkVersion` and `targetSdkVersion`. For the remaining 18.3% of the projects, the API levels that we consider are beyond the `targetSdkVersion`. Note that the manifest files are not static; they are often updated when developers wish to support newer Android OS versions. Hence, for the 18.3% of the projects, developers may still wish to use AndroEvolve to update the code when they want to migrate it to a more recent Android OS version. Therefore, we believe there is a limited threat to internal validity related to this potential issue.

7 Conclusion and Future Work

Updating the usages of deprecated Android APIs is a priority to ensure the functionality of Android apps in the current and previous versions of the Android OS. However, performing such updates is time-consuming and labor-intensive. In this work, we proposed AndroEvolve, an automated Android API usage update tool. AndroEvolve uses data flow analysis to resolve the values of out-of-method variables, allowing AndroEvolve to work on the file scope. AndroEvolve also performs variable denormalization to produce updated code with good readability. We evaluated AndroEvolve using a dataset of 372 target files containing 565 deprecated API usages, from which it managed to produce 481 successful updates, achieving an accuracy of 85.1%. On the same dataset, CocciEvolve, the previous state-of-the-art tool, only managed to produce 362 successful updates. We also evaluate the readability of the updated code using both manual and automatic measurements. In the manual measurement, we asked two developers for their opinion of the readability of the updated code, while in the automatic measurement, we used a code readability scoring tool. In both measurements, AndroEvolve outperforms CocciEvolve.

For future work, we plan to increase the capability of AndroEvolve. First, we plan to improve the data flow analysis to allow resolving values that are located in other files within the same project. This addition will make AndroEvolve capable of handling out-of-file variables. In our dataset, however, this case is uncommon. Second, we also plan to handle more complex Android API updates, especially for cases where a single API is updated into several different APIs. This case is uncommon, but such capability may be needed for future versions of Android. Finally, as briefly mentioned in Section 3, we plan to improve the usability of AndroEvolve by integrating other existing automated methods, e.g., (Asyrofi et al. 2020; Gokhale et al. 2013; Pandita

et al. 2017), to reduce the remaining manual steps (i.e., identifying mappings between deprecated and updated APIs and providing after-update example).

The implementation and replication package of AndroEvolve are available at `https://github.com/soarsmu/AndroEvolve`

Declaration

Funding This research is supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and the ANR ITrans project.

Conflicts of Interest/Competing Interests The authors have no conflicts of interest to declare that are relevant to this paper.

References

Asyrofi MH, Thung F, Lo D, Jiang L (2020) AUSearch: Accurate API usage search in GitHub repositories with type resolution. In: IEEE International Conference on Software Analysis, Evolution and Reengineering

Bakeman R, Gottman J (1986) Observing interaction: An introduction to sequential analysis

Bodík R, Gupta R (1997) Partial dead code elimination using slicing transformations. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA, PLDI '97, p 159–170

Brito G, Hora A, Valente MT, Robbes R (2016) Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In: SANER, IEEE, vol 1, pp 360–369

van Bruggen D, Tomassetti F, Howell R, Langkabel M, Smith N, Bosch A, Skoruppa M, Maximilien C, ThLeu, Panayiotis, (@skirsch79) SK, Simon, Beleites J, Tibackx W, L JP, Rouél A, edefazio, Schipper D, Mathiponds, you want to know W, Beckett R, ptitjes, kotari4u, Wyrich M, Morais R, bresai, Ty, Lebouc R, Implex1v, Haumacher B (2020) javaparser/javaparser: Release javaparser- parent-3.15.22

Brunel J, Doligez D, Hansen RR, Lawall JL, Muller G (2009) A foundation for flow-based program matching: using temporal logic and model checking. In: Principles of Programming Languages (POPL), ACM, pp 114–126

Buse R, Weimer W (2010) Learning a metric for code readability. Software Engineering, IEEE Transactions on 36:546–558

Cohen J (1968) Weighted kappa: nominal scale agreement with provision for scaled disagreement or partial credit. Psychological bulletin 70 4:213–20

Cui Z, Jia M, Chen X, Zheng L, Liu X (2020) Improving software fault localization by combining spectrum and mutation. IEEE Access 8:172296–172307, DOI 10.1109/ACCESS.2020.3025460

Fazzini M, Xin Q, Orso A (2019) Automated API-usage update for Android apps. In: ISSTA, ACM, pp 204–215

Ghandour WJ, Akkary H, Masri W (2010) The potential of using dynamic information flow analysis in data value prediction. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, Association for Computing Machinery, New York, NY, USA, PACT '10, p 431–442

Gokhale A, Ganapathy V, Padmanaban Y (2013) Inferring likely mappings between APIs. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, ICSE '13, p 82–91

Gupta R, Benson DA, Fang JZ (1997) Path profile guided partial dead code elimination using predication. In: Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques, pp 102–113

Han D, Zhang C, Fan X, Hindle A, Wong K, Stroulia E (2012) Understanding Android fragmentation with topic analysis of vendor-specific bugs. In: 2012 19th Working Conference on Reverse Engineering, IEEE, pp 83–92

Haryono SA, Thung F, Kang HJ, Serrano L, Muller G, Lawall J, Lo D, Jiang L (2020) Automatic Android deprecated-API usage update by learning from single updated example. In: IEEE International Conference on Program Comprehension

Hoffmann J, Ussath M, Holz T, Spreitzenbarth M (2013a) Slicing droids: Program slicing for Smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA, SAC '13, p 1844–1851

Hoffmann J, Ussath M, Holz T, Spreitzenbarth M (2013b) Slicing Droids: Program slicing for Smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA, SAC '13, p 1844–1851, DOI 10.1145/2480362.2480706, URL `https://doi.org/10.1145/2480362.2480706`

Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT (2015) How do developers react to API evolution? The Pharo ecosystem case. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 251–260

Jacobellis J, Meng N, Kim M (2013) LASE: An example-based program transformation tool for locating and applying systematic edits. In: 2013 35th International Conference on Software Engineering (ICSE), pp 1319–1322

Kang HJ, Thung F, Lawall J, Muller G, Jiang L, Lo D (2019) Semantic patches for Java program transformation (experience report). In: 33rd European Conference on Object-Oriented Programming (ECOOP 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

Khedker U, Sanyal A, Karkare B (2009) Data Flow Analysis: Theory and Practice, 1st edn. CRC Press, Inc., USA

Kim Y, Mun S, Yoo S, Kim M (2019) Precise learn-to-rank fault localization using dynamic and static features of target programs. ACM Transactions on Software Engineering and Methodology 28:1–34, DOI 10.1145/3345628

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174

Lawall J, Muller G (2018) Coccinelle: 10 years of automated evolution in the Linux kernel. In: USENIX Annual Technical Conference, pp 601–614

Li J, Wang C, Xiong Y, Hu Z (2015) SWIN: Towards type-safe Java program adaptation between apis. pp 91–102, DOI 10.1145/2678015.2682534

Li L, Bissyandé TF, Wang H, Klein J (2018a) Cid: Automating the detection of API-related compatibility issues in Android apps. In: ISSTA, ACM, pp 153–163

Li L, Gao J, Bissyandé TF, Ma L, Xia X, Klein J (2018b) Characterising deprecated Android APIs. In: Proceedings of the 15th International Conference on Mining Software Repositories (MSR), ACM, pp 254–264

McDonnell T, Ray B, Kim M (2013) An empirical study of API stability and adoption in the Android ecosystem. In: 2013 IEEE International Conference on Software Maintenance, IEEE, pp 70–79

Meng N, Kim M, McKinley KS (2013) LASE: Locating and applying systematic edits by learning from examples. In: ICSE, IEEE Press, pp 502–511

Mi Q, Keung J, Xiao Y, Mensah S, Gao Y (2018) Improving code readability classification using convolutional neural networks. vol 104, DOI 10.1016/j.infsof.2018.07.006

Padioleau Y, Lawall J, Hansen RR, Muller G (2008) Documenting and automating collateral evolutions in Linux device drivers. In: European Conference on Computer Systems (EuroSys), ACM, pp 247–260

Pandita R, Jetley R, Sudarsan S, Menzies T, Williams L (2017) TMAP: Discovering relevant API methods through text mining of api documentation. Journal of Software: Evolution and Process 29:e1845, DOI 10.1002/smr.1845

Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability. pp 73–82, DOI 10.1145/1985441.1985454

Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to API deprecation?: the case of a Smalltalk ecosystem. In: FSE, ACM, p 56

Rolim R, Soares G, D'Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B (2017) Learning syntactic program transformations from examples. In: ICSE, IEEE Press, pp 404–415

Sawant AA, Robbes R, Bacchelli A (2018) On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. EMSE 23(4):2158–2197

Scalabrino S, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2016) Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp 1–10

Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2017) Automatically assessing code understandability: How far are we? In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 417–427

Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D (2018) A comprehensive model for code readability. Journal of Software: Evolution and Process 30

Thung F, Haryono SA, Serrano L, Muller G, Lawall J, Lo D, Jiang L (2020) Automated deprecated-API usage update for Android apps: How far are we? In: IEEE International Conference on Software Analysis, Evolution and

Reengineering
Upton G (1992) Fisher's exact test. Journal of the Royal Statistical Society Series A (Statistics in Society) 192:395–402, DOI 10.2307/2982890
Visser E (2001) Stratego: A language for program transformation based on rewriting strategies. In: Proceedings of the 12th International Conference on Rewriting Techniques and Applications, Springer-Verlag, RTA '01, pp 357–362
Wei L, Liu Y, Cheung SC (2016) Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In: ASE, IEEE, pp 226–237
Weiser M (1984) Program slicing. IEEE Transactions on Software Engineering SE-10(4):352–357
Xi Y, Shen L, Gui Y, Zhao W (2019) Migrating deprecated api to documented replacement: Patterns and tool. pp 1–10, DOI 10.1145/3361242.3361246
Yang G, Jones J, Moninger A, Che M (2018) How do Android operating system updates impact apps? In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, ACM, pp 156–160
Zhang M, Li Y, Li X, Chen L, Zhang Y, Zhang L, Khurshid S (2019) An empirical study of boosting spectrum-based fault localization via pagerank. IEEE Transactions on Software Engineering PP:1–1, DOI 10.1109/TSE.2019.2911283
Zhou J, Walker RJ (2016) API deprecation: a retrospective analysis and detection method for code examples on the web. In: ICSE, ACM, pp 266–277
Zhou X, Lee Y, Zhang N, Naveed M, Wang X (2014) The peril of fragmentation: Security hazards in Android device driver customizations. Proceedings - IEEE Symposium on Security and Privacy pp 409–423
Štrobl R, Troníček Z (2013) Migration from deprecated API in Java. pp 85–86, DOI 10.1145/2508075.2508093