Towards Automatically Repairing Compatibility Issues in Published Android Apps

Yanjie Zhao
Monash University
Melbourne, Australia
Yanjie.Zhao@monash.edu

Li Li
Monash University
Melbourne, Australia
Li.Li@monash.edu

Kui Liu
Nanjing University of Aeronautics and Astronautics
Naijing, China
Kui.Liu@nuaa.edu.cn

John Grundy
Monash University
Melbourne, Australia
John.Grundy@monash.edu

ABSTRACT

The heavy fragmentation of the Android ecosystem has led to severe compatibility issues with apps, including those that crash at runtime or cannot be installed on certain devices but work well on other devices. To address this problem, various approaches have been proposed to detect and fix compatibility issues automatically. However, these all come with various limitations on fixing the compatibility issues, e.g., can only fix one specific type of issues, cannot deal with multi-invocation issues in a single line and issues in released apps. To overcome these limitations, we propose a generic approach that aims at fixing more types of compatibility issues in released Android apps. To this end, our prototype tool, *RepairDroid*, provides a generic app patch description language for users to create fix templates for compatibility issues. The created templates will then be leveraged by *RepairDroid* to automatically fix the corresponding issue at the bytecode level (e.g., right before users install the app). *RepairDroid* can support template creations for OS-induced, device-specific and inter-callback compatibility issues detected by three state-of-the-art approaches. Our experimental results show that *RepairDroid* can fix 7,660 out of 8,976 compatibility issues in 1,000 randomly selected Google Play apps. *RepairDroid* is generic to configure new compatibility issues and outperforms the state-of-the-art on effectively repairing compatibility issues in released Android apps.

1 INTRODUCTION

The heavy fragmentation problem of Android – many different Android versions (official and customized) running on different devices released by hundreds of manufacturers – has caused severe compatibility issues for the Android ecosystem. Android phone users often find that certain apps cannot be installed on their devices or can be installed but will crash later on if a specific function is reached, leading to poor user experiences. Actually, as revealed by Byron Muhlberg, there are over a billion Android devices no longer supported by Google. Hence, the users of those devices will likely encounter compatibility issues, especially when they want to leverage the latest Android apps, leading to serious problems in the mobile ecosystem.

To address this problem, approaches have explored various ways to automatically detect compatibility issues in Android apps [4, 8, 10, 11, 13, 17, 20, 23, 25, 34, 35]. For example, Wei et al. [31] empirically looked into a set of fragmentation-induced compatibility issues (including those introduced by third-party manufacturers) in open-source Android apps. They further proposed a tool called FicFinder to detect the previously characterized compatibility issues automatically. Li et al. [19] proposed a generic approach called CiD, which detects API-related compatibility issues based on Android API lifecycle knowledge mined from the official's historical evolution Android framework.

Unfortunately the majority of these works only focus on detecting some compatibility issues, leaving many identified issues still unfixed in real-world Android apps. Updating incompatible APIs is a time-consuming endeavor and app developers are also known to be reluctant to repair their apps for fixing issues yielded by static analyzers [9]. App developers have to learn the usages of new APIs in order to replace the incompatible ones while maintaining backward compatibility with the old version. This greatly increases the learning cost for developers.

To mitigate this, researchers have proposed several automated approaches to repair incompatible APIs for Android apps. Fazzini et al. [8] introduced AppEvolve to update incompatible APIs based on examples of how other developers evolved their apps for the same changes. Similarly, Lamothe et al. [17] proposed an approach called A3 for supporting API migration through patterns mined from source code examples via abstract syntax tree (AST). These approaches require the target code under repair written syntactically similar to the before- and after-update API change examples. This makes them hard to find applicable updates, as claimed by Thung et al. [27]. To this end, Haryono et al. proposed CocciEvolve [10] to only learn updates from a single after-update example. It eliminates the weakness of AppEvolve by normalizing both the after-update example and the target app code. Nevertheless, the serious issue of CocciEvolve is its inability to resolve all the values used as API arguments. These can be expressed in various complex forms, e.g., field access expressions, method invocations, and object creations. Its other drawback is the poor readability of its updated app code results. Its authors further extend their work by proposing AndroEvolve [11], which addresses the limitations of CocciEvolve through the addition of data flow analysis and variable name denormalization.

Unfortunately, all the above approaches focus on repairing the source code of Android apps and hence cannot be applied to directly repair published Android apps. Indeed, as time goes by, many published Android apps on popular app markets such as Google Play, unless being timely fixed, will become obsolete, leading to poor user experiences in the mobile ecosystem. To cope with this, market maintainers could choose to remove those apps. However, this may not be a good business model as it may reduce the market's competitiveness, i.e., its competitors are providing more choices of apps for users to explore. If market maintainers do not remove those apps, certain apps in the market will not be able to be installed on users' devices or will crash after installation. It will also cause poor user experiences and can even harm the reputation of the market and the app developers per se.

As a supplement to existing repair approaches that attempt to help developers in developing higher-quality apps, we believe there is also a need to provide approaches for helping repair published apps (before they are installed on users' devices), at least in the time period before their developers explicitly update the apps. This paper proposes a novel, generic approach for repairing three types of compatibility issues – API, device and callback-induced problems – in published Android apps. Since it is relatively simple to transform Java source code to bytecode and vice versa, the approach targeting published Android apps, with small change, could also be applied to repair apps at the source code level (but not the other way around). Nevertheless, we argue that source code- and bytecode-based repairing approaches are not mutually exclusive. They can co-exist and complement each other. Indeed, approaches targeting published app repair could be leveraged to achieve emergent repair, while source code-based approaches can come in later to gradually fix the issues. For example, market maintainers could leverage published app repairs to ensure the compatibility of their hosted apps (with developers' permission) at app uploading or downloading time. This is extremely useful, especially for legacy apps that are less frequently (will no longer be) maintained by their developers. Correspondingly, end-users are provided options to use apps that could not be running initially on their devices.

In this work, we present a prototype tool, *RepairDroid*, which leverages pre-defined patch templates to instrument Android apps so as to fix compatibility issues. The templates are written by dedicated experts (i.e., app developers do not need to write patches for their apps) based on a structural model that is both descriptive and generic, i.e., a given patch should be applicable to all Android apps. The instrumentation process involves control-flow and data-flow analysis to locate and repair compatibility issues. Experimental results on thousands of real-world Android apps show that *RepairDroid* is effective in automatically repair various types of compatibility issues. In this research we make the following key contributions:

- We have designed a novel app patch description language and demonstrated that it is generic enough to be used to create fix templates for various compatibility issues. The genericity is achieved by allowing users to directly leverage the simple but well-defined Jimple grammar (i.e., a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java/Android bytecode) to describe the patches.
- We have designed and implemented a prototype tool *RepairDroid*, which follows given fix templates to automatically repair published real-world Android apps.
- We have evaluated our approach against 1,000 real-world Android apps. Experimental results show that our approach is effective in repairing Android apps, outperforms the state-of-the-art and achieves 85.34% of successful repairing rate.

Open source. The source code and datasets are all made publicly available in our artifact package [2].

2 MOTIVATION

The heavy fragmentation of the Android ecosystem has induced many types of compatibility issues in Android apps. Our research community has spent lots of effort on disclosing such issues, including at least the following three types:

OS-induced compatibility issues. This is one of the most common types of compatibility issues, where issues are caused by the evolution of the Android framework. During framework evolution, new APIs are regularly added to the framework, while existing APIs are also regularly deprecated and removed. In some rare cases, existing APIs may also be semantically changed, despite keeping the signature of the APIs unchanged. The example given in Listing 1 shows a deprecated API issue.

```
+ if (android.os.Build.VERSION.SDK_INT >= 28) {
+   for (Network nw: cm.getAllNetworks()) {
+     NetworkCapabilities nc =
        cm.getNetworkCapabilities(nw);
+     if (nc != null &&
        nc.hasTransport(NetworkCapabilities.TRANSPORT_WIFI))
+       return true;
+   }
+   return false;
+ } else {
    return an.getType() ==
      ConnectivityManager.TYPE_WIFI;
+ }
```

Listing 1: An example of an OS-induced compatibility issue. API *getType* is deprecated at SDK level 28. On devices running SDK versions larger than 28, it is recommended to use API *hasTransport* instead.

Device-specific compatibility issues. These compatibility issues are associated with specific devices running customized Android systems. The problematic apps will only crash on certain devices while behaving normally on others, despite all the devices running the same Android framework version. Listing 2 presents such an example that was initially reported by Wei et al. [32]. The API *setRecordingHint* depends on a conditional statement that checks the device identifier against "Nexus 4". Only the condition to be true, i.e., the corresponding app is indeed running on "Nexus 4", the API will be executed.

```
  Camera mCamera = Camera.open();
  Camera.Parameters params =  mCamera.getParameters();
  ......
+ if (android.os.Build.MODEL.equals("Nexus 4") {
+   params.setRecordingHint(true);
+ }
  ......
  mCamera.setParameters(params);
  mCamera.startPreview();
```

Listing 2: Patch for Camera Preview Frame Rate Issue on Nexus 4, excerpted from [32].

Inter-callback compatibility issues. This type of compatibility issue is caused by the changes to Android system callbacks (also known as lifecycle methods). Such system callback methods are pre-defined by the Android system and will be directly executed when certain conditions are satisfied. Listing 3 illustrates such an example that was initially reported by Huang et al. [14]. The *onAttach(Context)* callback method is only introduced from API level 23. If this code is running on smartphones with earlier API levels, this callback method will not be executed. Subsequently, the *mActivity* field will not be initialized, and its usage will likely throw *NullPointerExceptions*.

```
1  public void onAttach(Context context) {
2    super.onAttach(context);
3  - mActivity = (BrowserActivity) context;
4  - ......
5  + attachActivity((BrowserActivity) context);
6   }
7  + public void onAttach(Activity activity) {
8  +  super.onAttach(activity);
9  +  if (Build.VERSION.SDK_INT < 23) {
10 +    attachActivity((BrowserActivity) activity);
11 +  }
12 + }
13 + private void attachActivity(BrowserActivity
       activity) {
14 +  mActivity = activity;
15 +  ......
16 + }
```

Listing 3: The Patch for WordPress issue 6906. The compatibility issue is caused by the fact that the callback method *onAttach(Context)* is not yet available before API level 23, excerpted from [14].

All of these compatibility issues are equally critical to mobile apps as all of them will cause apps to crash, leading to poor user experiences. Compatibility issue repairing approaches should aim to fix all of them. However, current state-of-the-art tools only focus on repairing API-induced compatibility issues. Automatically fixing other types of compatibility issues, such as device or callback related ones, has not yet been addressed.

As summarized in Table 1, existing approaches also come with many limitations. For example, CocciEvolve only attempts to fix incompatible APIs within a single method. Their follow-up work AndroEvolve fixes this limitation by additionally introducing data-flow analysis into the fixing process. Compatibility issues with respect to (1) out-of-file variables and (2) multi-invocations in a single line and (3) compatibility issues in released Android apps cannot be resolved by any of the state of the art.

Table 1: Problematic Android compatibility issues addressed by state-of-the-art tools.

Feature	A3	AppEvolve	CocciEvolve	AndroEvolve
Out-of-method (within file) variables	✓	✗	✗	✓
Out-of-file variables	✗	✗	✗	✗
1-to-n replacement	✗	✓	✗	✗
Multi-invocations in a single line	✗	✗	✗	✗
Fix in published Android apps	✗	✗	✗	✗

3 OUR APPROACH: *REPAIRDROID*

We introduce *RepairDroid*, a template-based repair approach, to automatically repair three kinds of compatibility issues in Android apps. Figure 1 presents an overview of *RepairDroid*. It has three key modules: (1) Template Build Module (TBM), (2) Bug Location Module (BLM), and (3) Bug Repair Module (BRM). Below we detail these three modules.

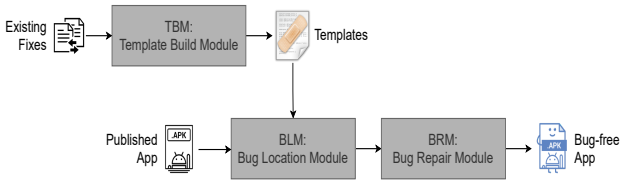


Figure 1: The working process of *RepairDroid*.

3.1 TBM: Template Build Module

This first module of *RepairDroid* aims at preparing a set of semantic templates for subsequent modules to repair published Android apps. The main contribution of this module is a generic language for describing app patch templates. Developers should be able to easily leverage the language to create templates for fixing compatibility issues, based on knowledge learned from the official Android documentation, online question and answer websites such as StackOverflow, or existing fixing samples mined from the evolution of real-world Android apps. The grammar of the language is straightforward. As shown in Figure 2, it contains three blocks, namely variable declaration block, issue location block, and patch block. We now detail these three blocks, respectively.

```
@@ Variable Declaration
$v0 := boolean
$v1 := ANY
[SEARCH] $v2 := <TYPE>

@@ Issue Location
[<ISSUE_TYPE>] <CONDITION>

@@ Patch Denotation
+ //Replacement Statements
- //Original Statements

OR

+ $v0 = <CONDITION EXPRESSION>
+ if $v0 == true
+   //Replacement Statements
+ else
    //Original Statements
```

Figure 2: The language structure for creating patch templates for *RepairDroid*.

Variable Declaration Block. The first block provides a means for developers to specify all the variables involved in the template. At the moment, the language supports three types of variables.

- **Variables to be directly reused from the original context.** These variables appear in the original code statements (used for locating problems) and hence could be directly reused in the new code.

- **Variables to be searched from the original context.** These variables do not appear in the original code statements that are defined to locate the problems but do exist in the app context. Hence, backward data-flow analysis is demanded to search the definition of the variables for direct reuses. These variables will be explicitly marked by keyword [SEARCH], as demonstrated in Figure 2.
- **Variables to be created.** These variables are totally new to the app and hence need to be defined before applying the template.

Issue Location Block. The second block is provided for developers to specify how the compatibility issues can be programmatically located. Developers first specify the issue type (in square brackets) to be fixed to simplify the location process. This is because different types of compatibility issues require different strategies to locate them. As inspired by the three types of compatibility issues summarized in Section 2, the language currently provides three issue types (i.e., [OS], [DEVICE], and [CALLBACK], respectively for OS-induced, device-specific, and inter-callback compatibility issues) to guide *RepairDroid* for locating compatibility issues. More issue types could be added to the language if new types of compatibility issues are identified in the future.

Like the variable declaration block, the issue location block is also relatively easy to configure. In most cases, it only needs one line of statement to specify the issue. For example, the compatibility issues in the three motivating examples shown in Section 2 can be respectively specified by the issue location statements shown in Listing 4.

In addition to the targeted APIs, <CONDITION> also specifies the situation when the issue should not be considered, even if the targeted APIs or call methods are located. Taking Line 1 in Listing 1 as an example, if the invocation of the deprecated API *getType()* is already protected by an SDK version check (i.e., if (Build.VERSION.SDK_INT >= 28)), this target should be ignored as there should have no compatibility issue in such a case.

Template Denotation Block. The last module of the app patch language is used by developers to specify the actual template for fixing the located compatibility issues. The fix template is designed to be written in Jimple-like pseudocode. Jimple is the default intermediate representation of Soot [28], a well-known Java/Android static analysis framework. The reason why we choose Jimple to describe the template is that the instrumentation function of *RepairDroid* is implemented on top of Soot. It allows *RepairDroid* to quickly apply the template to fix compatibility issues in a deployed app[1].

As shown in Figure 2, the typical way to fix an OS-induced compatibility issue is to replace the original problematic statements with corrected new statements. In practice, we recommend the developers to always guard their newly introduced code through a conditional check. When the check returns true, the replacement statements will be invoked. Otherwise, the original statements will be executed and hence the original behaviors are kept.

```
1  [Method] <[android.app.Fragment]: void
       onAttach(android.content.Context)>
2  [Stmt]$r0 := @this: [android.app.Fragment]
3  [Stmt]$a0 := @parameter0: android.content.Context
4  [Stmt]specialinvoke $r0.<android.app.Fragment: void
       onAttach(android.content.Context)>($a0)
5  [CUT] ... when != [END of Method]
6  + $v0 = ([android.app.Activity]) $a0;
7  + virtualinvoke $r0.<[android.app.Fragment]: void
       attachActivity([android.app.Activity])>($v0);
8  + return
9
10 + [NEW Method][public] <[android.app.Fragment]: void
       onAttach(android.app.Activity)>
11 + $r1 := @this: [android.app.Fragment]
12 + $a1 := @parameter0: android.app.Activity
13 + specialinvoke $r1.<android.app.Fragment: void
       onAttach(android.app.Activity)>($a1)
14 + $i1 = <android.os.Build$VERSION: int SDK_INT>
15 + if $i1 >= 23 goto <label_1>
16 + $v1 = ([android.app.Activity]) $a1;
17 + virtualinvoke $r1.<[android.app.Fragment]: void
       attachActivity([android.app.Activity])>($v1);
18 + <label_1>
19 + return
20 + [END of Method]
21
22 + [NEW Method][private] <[android.app.Fragment]: void
       attachActivity([android.app.Activity])>
23 + $r2 := @this: [android.app.Fragment]
24 + $a2 := @parameter0: [android.app.Activity]
25 [PASTE]
26 + return
27 + [END of Method]
```

Listing 5: The template denotation block of the inter-callback example.

Listing 5 shows an example Template Denotation block for the inter-callback example shown in Listing 3. Based on the example in Listing 5, we further introduce several keywords, such as [CUT], [PASTE], to respectively represent cut and paste operations, with which they can batch process statements within the set range, as shown at Line 5 and Line 25. Inspired by SmPL [18], on the basis of ensuring the universality of the language, the templates for these three issues have designed some special symbols, including the use of some reserved words.

$$[CUT]...when! = [End\ of\ Method]$$

The "..." operator in SmPL represents an arbitrary sequence, i.e., any sequence of statements over any control flow path, which is also used in our language. For example, the usage of *when != [End of Method]* at Line 5 means that there should be no occurrences of *[End of Method]* in the matched control-flow path, that is, the matching process continues to the end of the method. Although there are no complicated usages in the repair of the three types of compatibility issues introduced above, our intention for this app patch language design is to ensure the future scalability of *RepairDroid*. To this end, *[End of Method]* can be replaced with other statements (e.g., tag statements such as *<label_1>*) to ensure that *RepairDroid* can be guided to accurately collect statements.

In real-world Android projects, the class inheritance feature has been frequently leveraged. In order to improve the versatility of *RepairDroid*, we use ["Superclass Name"] to specify the superclass of the class to be searched. For example, [android.app.Fragment] means *RepairDroid* needs to search such a class that extends the

[1]Ideally, we would like to support Java as the language for describing the patch directly as it would be more convenient for patch writers. However, this will make the patch parsing step difficult to achieve as essentially it asks for a Java compiler to interpret the (random) Java code. Jimple is a simplified Java representation that could be an ideal trade-off solution, i.e., not very difficult to understand and write but can be interpreted programmatically in practice (thanks to Soot).

```
[OS] <android.net.NetworkInfo: int getType()> Build.VERSION.SDK_INT 28

[DEVICE] <Camera: Camera$Parameters getParameters()> Build.MODEL "Nexus 4"

[CALLBACK] <Fragment: void onAttach(Context)> Build.VERSION.SDK_INT 23
```

Listing 4: Examples of issue location statements for creating templates to fix the compatibility issues listed in the three motivating examples (i.e., Listings 1-3), respectively

superclass named "android.app.Fragment" before performing subsequent work. If the superclass does not need to be restricted, [DECLARING_CLASS], a reserved keyword, can be directly used to indicate the class to which the current method belongs. *RepairDroid* will replace it with the actual value in the app when running on the BRM module.

3.2 BLM: Bug Location Module

The second module of *RepairDroid* takes as input the Android APK to analyze and the semantic templates generated by the first module and outputs the locations (at the statement level) indicating where the templates should be applied. Specifically, this module takes the following three steps to achieve its purpose: *template parsing*, *app pre-processing*, and *bug localization*. We now detail these steps, respectively.

Template parsing. As a prerequisite to the following steps, *RepairDroid* first reads and parses the semantic templates generated by the TBM module. The parsing process utilizes the principle of Finite State Machine to parse the input semantic templates. After the parsing step, each semantic template is stored as a structured object that is readily available for further references.

App pre-processing. This step first transforms the bytecode into an intermediate representation code called Jimple, as it is non-trivial to directly analyze the Dalvik bytecode of Android apps. Jimple is the default intermediate representation format of Soot, a Java/Android app static analysis and optimization framework. In this work, *RepairDroid* leverages Soot to achieve the code transformation and the following-up static analysis of Android apps.

Bug localization. This step traverses the Jimple IRs of the target app and detects the locations that need to be patched according to the specified conditions parsed from the semantic templates above. *RepairDroid* automatically identifies the bug locations (often at the Jimple statement level) from the app code by traversing each method in each class. Subsequently, the located bug statements, along with their belonging methods, will be regarded as a potential bug candidate.

Then for each identified candidate, *RepairDroid* goes one step further to check if it satisfies certain conditions, following what will be also specified in the semantic template. If so, the candidate will be regarded as a true bug and hence will be propagated to the next module for automated repairing. For compatibility issues, the bug locations will often be an API invocation statement (e.g., because the API is no longer available in the latest Android devices or in certain customized Android versions such as Samsung phones). The conditions could be a framework version check or a device manufacturer check. Taking Listing 6 as a simple example, API *abandonAudioFocus* is deprecated in the Android framework version *M* and hence is only recommended to be invoked on devices running lower versions than *M*. However, this method call should not be detected as containing a compatibility issue because the problem has already been protected (i.e., fixed). The conditional check (i.e., against *Build.VERSION_CODES.M*) should have been clearly specified in the semantic template of API *abandonAudioFocus*. To resolve this issue, after locating candidate bugs, *RepairDroid* goes one step deeper to perform an inter-procedural backward control-flow analysis to check if their associated conditions are presented. Only if the incompatible API calls are not already protected, *RepairDroid* will attempt to repair the corresponding issues.

We use the example shown in Listing 6 to illustrate this backward analysis flow. The deprecated API *abandonAudioFocus* and its replacement API *abandonAudioFocusRequest* are called in two separate methods, i.e., *abandonAudioFocus()* (i.e., Line 4) and *abandonAudioFocusRequest()* (i.e., Line 1), and the condition check statement is located in the method *dispose()* (i.e., Line 7). When locating an OS-induced issue, *RepairDroid* can easily pinpoint the deprecated API *abandonAudioFocus* (i.e., Line 5) and its declaring method, *abandonAudioFocus()*. However, there is no conditional check statement for the SDK version in the *abandonAudioFocus()* method. As a result, we need to locate method *dispose()* that calls the method *abandonAudioFocus()* through the backward analysis flow to determine that the current case does not contain an incompatible issue.

```
1  public void abandonAudioFocusRequest() {
2    audioManager.abandonAudioFocusRequest(request);
3  }
4  public void abandonAudioFocus() {
5    audioManager.abandonAudioFocus(this);
6  }
7  public void dispose(){
8    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M){
9      abandonAudioFocusRequest();
10   } else {
11     abandonAudioFocus();
12   }
13 }
```

Listing 6: Code snippets showing the demand for conducting inter-procedural backward flow analysis.

3.3 BRM: Bug Repair Module

The last module of *RepairDroid* is used to repair the located app compatibility bugs by directly updating the code snippets (hereinafter referred to as the *Target*) located by the BLM module, i.e., $Target \xrightarrow{Template} Target'$. The *Template* represents the list of statements used to update the *Target* that has been identified as containing bugs. Subsequently, $Target'$ represents the list of statements having the identified bugs repaired.

Algorithm 1 summarizes the general repairing process implemented in *RepairDroid*. *RepairDroid* first needs to ensure that all the variable keywords[2] involved in the semantic template are available in the code context. It transforms the variable keywords defined as strings in the template to Java objects at the code level. As demonstrated in Lines 2-12, for each variable keyword, *RepairDroid* first

[2]To avoid confusion, we use variable keywords to describe the variables defined as strings in the template. The term variable by itself is kept for referring to Java objects.

checks if its corresponding variable is directly available in the *Target* code. If so, it will be directly reused. If a given variable keyword is marked by keyword [SEARCH], we then consider the variable keyword needs to be searched in the *Target*'s context (the variable should have been defined in the app but has not been leveraged by the *Target* code). Finally, if the variable keyword is defined as a new variable, it will be directly initiated. Taking the code snippet shown in Listing 1 as an example, the variable *cm* (at Line 2) is such a case that needs to be searched backward, and *nw* (at Line 2) and *nc* (at Line 3) are variables that need to be created based on the searched variable *cm*.

When there are variables that do not exist in the original app code statements, *RepairDroid* leverages the *searchLocal* function to conduct backward control-flow analysis so as to select existing variables from the code context. The search process follows the following rules. These are summarized based on our manual observations among various fixes that happened in real-world Android apps. (1) Search backward from the position of *Target* for the variable in the located buggy statement's declaring method. The first seen variable (matched via type) will be considered. (2) If it is not possible to identify the variable in the current method, *RepairDroid* then traverses all the fields declared in the class to which the previously searched method belongs. Next, (3) if it still fails to locate the variable, *RepairDroid* will resort to the whole class, including its inner classes, to search for the variable. Finally, suppose it is still impossible to locate an existing variable after exploring all the aforementioned rules, *RepairDroid* will terminate the repair process and regard the fix as a failure case.

After preparing all the required variables, the next step is to update the *Target* code following the Jimple statements defined in the template. Since the Jimple statements are provided as strings, *RepairDroid* takes additional step to automatically transform it to code snippets (via the BUILD-JIMPLE-STMT function). Recall that the Jimple statements written in the template may contain certain placeholder keywords such as ["Superclass Name"], [CUT], and [PASTE]. When transforming the statements, *RepairDroid* needs to replace them with concrete values.

For example, as shown in Line 1 of Listing 5, *RepairDroid* first collects the actual name of the class inheriting the superclass *android.app.Fragment*, which contains incompatible issues. Then, it replaces [android.app.Fragment] with its real value, e.g., *org.wordpress.android.ui.themes.ThemeBrowserFragment* in the app of Listing 3, which extends the *android.app.Fragment* class, at runtime. Furthermore, some of the searched variables e.g. those returned by Rule (3)), may not be directly accessible in the method under repairing. To this end, *RepairDroid* goes one step further to introduce glue code to change the visibility of the search variables so that they can be freely used to repair the buggy statements.

3.4 Implementation

RepairDroid is implemented on top of Soot and the repair process is done at the Jimple code level. Thanks to Soot, the repaired Jimple code is further transformed back to a new Android app, which could be directly installed and used by users. Although our approach is proposed to repair Android apps directly at the bytecode level, app developers who want to repair their apps at the source code level could also benefit from our approach. By reverse-engineering the repaired Android apps, developers can get fixed source code, which could then be directly ported to the source code project to achieve source code repairs.

Algorithm 1: The repair algorithm of BRM.

Input: $Target$: the located buggy statements.
$Template$: the classes/methods/statements that should be inserted.

```
1  Var2Local = new HashMap;
2  VariableKeywords = getVariableKeywords(Template);
3  Body = Target.getDeclaringMethodBody();
4  for var ∈ VariableKeywords do
5      if var shows in Target then
6          Local = getLocal(Target,var);
7      else if var needs to be searched then
8          Local = searchLocal(Body, var);
9      else
10         Local = new Local(Body, var, Var2Local);
11     Var2Local.add(var, Local);
12 end
13 for Item ∈ Template_before do
14     NewStmt = BUILD-JIMPLE-STMT(Item, Var2Local);
15     insertBefore(NewStmt, Target.get[0]);
16 end
17 for Item ∈ Template_after do
18     NewStmt = BUILD-JIMPLE-STMT(Item, Var2Local);
19     insertAfter(NewStmt, Target.get[Target.size() − 1]);
20 end
```

4 EVALUATION

The goal of this work is to automatically repair compatibility issues in published Android apps. To determine if this objective has been achieved, we look to answer the following four key research questions:

RQ1: How generic is our proposed patch description language?

RQ2: How well does *RepairDroid* perform compared with existing tools?

RQ3: How effective is *RepairDroid* in automatically locating and repairing incompatibility issues in real-world Android apps?

RQ4: What is the time performance of *RepairDroid* in repairing published Android apps?

4.1 RQ1: Genericity

In the first research question, we investigate the genericity of our proposed patch description language, which is one of the core modules supporting the automated repair of Android apps. As discussed in Section 2, there are at least three types of compatibility issues suffered by real-world Android apps. The issues are significantly different from one to another. Even within the same type, the compatibility issues could also be remarkably different. Recall that we aim to design the patch description language to be as generic as possible so that it can be leveraged to describe templates for all the kinds of compatibility issues Android apps may encounter. To this end, we resort to evaluating the language's genericity by directly applying the language to create templates for all the compatibility issues explicitly mentioned in the three articles in which the aforementioned three types of compatibility issues (cf. Section 2) are introduced, respectively.

By manually summarizing the examples detected by the previously mentioned three approaches (i.e., CiD, Pivot, and CIDER,

respectively), we eventually decided to create templates for three OS-induced compatibility issues, seven device-specific compatibility issues, and six inter-callback compatibility issues. In the work of CiD, the authors listed seven issues, of which only three have been confirmed and fixed by the developers. Hence we only take these three issues into consideration. Furthermore, the authors of CiD have further presented another work, CDA [21], that reveals 19 more issues. As a supplement, we decided to also consider them to create OS-induced compatibility issues. However, for five of them, we cannot find intuitive replacements and hence these were excluded before the experiment. These five issues are associated with the Apache HttpComponents project, for which the whole project has now been deprecated by the Android framework. To address them hence requires fundamental code changes – removal of all the usage of Apache HttpComponents – to repair their related app compatibility issues. Such a change is far too difficult to be done automatically. We thus eliminated these five compatibility issues from this experiment. As a result, 17 OS-induced compatibility issues are considered, as enumerated in the second column in Table 2. In the work of Pivot, the authors present ten problematic apps, among which only seven compatibility issues are eventually fixed. These seven issues correspond to three distinct APIs. In the work of CIDER, the authors have reported nine issues, among which only six issues (correspond to four distinct methods) are eventually fixed. To ensure that we can find repair examples that provide specimens demonstrating how these compatibility issues can be fixed and thereby how to describe the templates, we decided to only focus on the three device-specific and four inter-callback compatibility issues that have been fixed or at least confirmed by the developers. The detailed selected issues are also listed in the second column in Table 2.

Table 2: Experimental results demonstrating the genericity of the patch description language.

Paper	Issue	If success	Total
CiD [19] CDA [21]	<Resources: Drawable getDrawable(...)>	✓	15/17
	<Notification: void setLatestEventInfo(...)>	✓	
	<View: void setBackgroundDrawable(...)>	✓	
	<Intent: ClipData getClipData()>	✓	
	<View: void setSystemUiVisibility(...)>	✓	
	<Notification.Builder: Notification.Builder setLocalOnly(...)>	✗	
	<Notification: void <init>(...)>	✓	
	<NetworkInfo: int getType()>	✓	
	<Display: int getWidth()>	✓	
	<Display: int getHeight()>	✓	
	<Resources: int getColor(...)>	✓	
	<PopupWindow: void setWindowLayoutMode(...)>	✓	
	<Activity: void setProgress(...)>	✗	
	<ContentProviderClient: boolean release()>	✓	
	<AccessibilityServiceInfo: String getDescription()>	✓	
	<AccessibilityServiceInfo: boolean getCanRetrieveWindowContent()>	✓	
	<Html: String toHtml(...)>	✓	
Pivot [32]	<DatePickerDialog: DatePickerDialog DatePickerDialog(...)>	✓	3/3
	<View: int getSystemUiVisibility()>	✓	
	<Camera$Parameters: void setRecordingHint(...)>	✓	
CIDER [14]	<Fragment: void onAttach(...)>	✓	4/4
	<WebViewClient: boolean shouldOverrideUrlLoading(...)>	✓	
	<WebViewClient: onReceivedError(...)>	✓	
	<WebViewClient: onReceivedHttpError(...)>	✓	

To evaluate the genericity of *RepairDroid*'s patch description language, for each of the selected compatibility issues in Table 2, the authors manually analysed each issue and its correct fixes and created templates following the rules defined by the language presented earlier. As shown in Table 2, for the 24 considered compatibility issues across three different types, **we successfully created 22 templates for these 24 app compatibility issues**. We will evaluate the correctness of these templates when answering the third research question.

The remaining two issues we could not fix using our *RepairDroid* patch description language. That is because these two issues have not been provided with clear replacement information indicating how they can be avoided on the official Android documentation site. These failures, however, have no connection with the genericity of the language. Indeed, if we are provided with well-structured repair samples, we could still generate templates for these two issues. Overall, this experimental result shows that our patch description language is quite generic and should be capable of describing patterns for finding and fixing a wide range of compatibility issues encountered by Android apps.

➽ RepairDroid*'s patch description language is generic and should be capable of describing most of the compatibility issues available in the Android ecosystem.*

4.2 RQ2: Comparison With State-of-the-art

To answer our second research question, we compare *RepairDroid* with all the state-of-the-art related works. To the best of our knowledge, as listed in Table 1, there are four tools (i.e., A3 [17], AppEvolve [8], CocciEvolve [10], AndroEvolve [11]) proposed by our fellow researchers used to automatically fix OS-induced compatibility issues in Android apps. However, A3 is mainly developed to mine migration patterns from code examples. Although it provides mechanisms to automatically apply the mined migration patterns to fix the problematic APIs, there is no guarantee that such attempts will be correct. We believe it is not fair to compare our approach with A3 and hence exclude A3 from the comparison.

Furthermore, because of certain limitations of AppEvolve, e.g., it cannot handle updates spanning multiple methods, CocciEvolve is proposed to complement AppEvolve. CocciEvolve is able to achieve better performance than AppEvolve on 112 target problems. The authors of CocciEvolve further propose AndroEvolve that extends CocciEvolve to achieve even better performance in automatically repairing incompatible APIs. Therefore, in this work, we compare our approach with AndroEvolve, the most relevant and advanced approach closest to ours.

In order to experimentally compare the performance of *RepairDroid* against AndroEvolve, we need to ensure that these two tools are launched to repair the same set of compatibility issues. We hence made effort to create compatibility patch templates for all of the issues targeted by AndroEvolve. Specifically, AndroEvolve was evaluated against 20 problematic APIs, as shown in the first column in Table 3. Following *RepairDroid*'s description language, we were able to create templates for all the 20 APIs. This also further demonstrates the genericity of *RepairDroid*'s patch description language, complimenting our RQ1 answer above.

We then launch *RepairDroid* and AndroEvolve to repair Android apps that contain the aforementioned 20 compatibility issues. We

build the corresponding projects used in the evaluation of AndroEvolve into apps and conduct experiments based on them, as *RepairDroid* requires published Android apps to check if *RepairDroid* can repair the corresponding issues. Table 3 summarizes the experimental results. ***RepairDroid* is able to successfully repair all the APIs**, including all of the ones that cannot be handled by AndroEvolve. As explicitly acknowledged by Hartono et al. [11], AndroEvolve cannot handle the updates of a single API into multiple APIs. Therefore, AndroEvolve fails to repair *getAllNetworkInfo()* API as its fix requires to access two APIs, i.e., *getAllNetworks()* and *getNetworkInfo()*. Moreover, AndroEvolve cannot deal with the case when multiple API invocations are written in a single line of code. It is indeed non-trivial to resolve this challenge as it may involve complicated operations in that line of code. However, this challenge will not be an issue for *RepairDroid*. Indeed, *RepairDroid* repairs Android apps at the Jimple code level for which the multiple invocations are separated into different lines. These experimental results show that *RepairDroid* goes beyond the state-of-the-art to repair compatibility issues in Android apps.

Table 3: Comparison results between *RepairDroid* and the state-of-the-art AndroEvolve approach.

API	AndroEvolve	RepairDroid
<AccessibilityNodeInfo: void addAction(...)>	✗	✓
<ConnectivityManager: NetworkInfo[] getAllNetworkInfo()>	✗	✓
<TimePicker: Integer getCurrentHour()>	✓	✓
<TimePicker: Integer getCurrentMinute()>	✓	✓
<TimePicker: void setCurrentHour(...)>	✓	✓
<TimePicker: void setCurrentMinute(...)>	✓	✓
<TextView: void setTextAppearance(...)>	✓	✓
<LocationManager: boolean addGpsStatusListener(...)>	✓	✓
<Html: Spanned fromHtml(...)>	✓	✓
<ContentProviderClient: boolean release()>	✓	✓
<LocationManager: boolean addGpsStatusListener(...)>	✓	✓
<WebViewClient: boolean shouldOverrideUrlLoading(...)>	✗*	✓
<View: boolean startDrag(...)>	✓	✓
<AudioManager: int abandonAudioFocus(...)>	✗	✓
<TelephonyManager: String getDeviceId()>	✓	✓
<AudioManager: int requestAudioFocus(...)>	✓	✓
<Canvas: int saveLayer(...)>	✓	✓
<MediaPlayer: void setAudioStreamType(int)>	✗	✓
<Vibrator: void vibrate(long)>	✓	✓
<Vibrator: void vibrate(long[],int)>	✓	✓

* No examples and scripts provided.

➽ RepairDroid *outperforms the state-of-the-art tools by achieving better performance in automatically repairing compatibility issues in Android apps.*

4.3 RQ3: Effectiveness of *RepairDroid*

Our third research question concerns the effectiveness of *RepairDroid* in automatically repairing compatibility issues in published Android apps. To answer this RQ we randomly selected 1,000 real-world Android apps from AndroZoo to form our test dataset. These 1,000 apps were originally collected from the official Google Play store and hence are all published apps (i.e., their source codes are not available anyway). Recall that, when answering RQ1 and RQ2, we have created templates for 42 distinct compatibility issues. We used all of these 42 templates when applying *RepairDroid* to try and detect and repair compatibility issues in the randomly selected apps. Note that at this stage, we do not know yet whether these apps contain true compatibility issues or not.

Table 4 summarizes the experimental results. 714 apps contain potential OS-induced compatibility issues (i.e., 8,519 in total[3]), and 492 apps suffer from potential device-specific issues (i.e., 3,086 in total). Only seven apps contain potential inter-callback compatibility issues (i.e., 31 in total). In this experiment, we consider an app that contains inter-callback compatibility issues only if the unsupported callback methods are explicitly overridden by developers. Among all the identified issues, *RepairDroid* locates that 5,932 OS-induced, 3,042 device-specific, and 2 inter-callback compatibility issues are true issues for which they are not already protected. For each of the located issues, *RepairDroid* then applies its corresponding template to perform the automated repair. Eventually, 4,616 OS-induced, 3,042 device-specific, and 2 inter-callback compatibility issues can be successfully fixed, giving a success rate at 77.82%, 100%, and 100%, respectively. To validate the fixes, we randomly sample 20 apps and leverage Soot's grammar checker to check if their updated code is grammatically correct. Then, we evaluate the repaired app through (1) manually comparing the repaired code with the original buggy code and (2) actually executing the repaired apps (as well as their original counterparts) to verify the fixes of the corresponding compatibility issues. The repaired code of the 20 sampled apps has been manually verified and confirmed to be correct, and all the repaired apps can also be normally installed on Android devices. **Overall, *RepairDroid* is able to achieve an 85.34% of success rate when repairing 1,000 randomly selected Android apps.** Figure 3 plots the distribution of the number of detected and the number of successfully fixed compatibility issues, i.e., the median and mean numbers are 10, 12.77, and 9, 11.12, respectively. The fact that the majority of located issues can be automatically repaired demonstrates the effectiveness of our approach.

The failure cases are mainly caused by the variable search module, for which *RepairDroid* fails to pinpoint the required variables based on the variable keywords leveraged in the template. Taking the code snippet displayed in Listing 1 again as an example, in our experiment, a total of 760 compatibility issues related to API *getType* are located. However, for around 25% at the moment, *RepairDroid* currently fails to search and locate the actual variable for keyword *cm*. Furthermore, due to limitations of Soot, *RepairDroid* also fails to repair a number of issues. Moreover, there might be multiple ways to fix an issue. We respectively provide a general repair template targeting each issue, which may not fully meet all cases' requirements. As of our future work, we commit to continuously improve our approach to increase its success rate in automatically fixing compatibility issues in published Android apps.

Table 4: Performance achieved by *RepairDroid* for repairing 1,000 randomly selected Android apps.

Issue Type	# Apps	# Potential Issues	# Located Issues (No Protection)	# Fixed Issues
OS-induced	714	8,519	5,932	4,616
Device-specific	492	3,086	3,042	3,042
Inter-callback	7	31	2	2
Total	725*	11,636	8,976	7,660 (85.34%)

* One app may suffer from multiple type of issues.

[3]Issues lie in Android framework code are ignored.

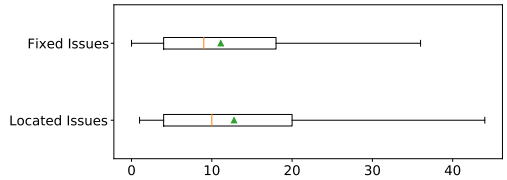


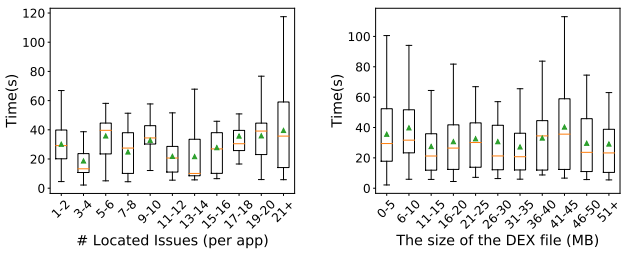
Figure 3: Distribution of the located and successfully fixed compatibility issues in each of the selected apps.

➽ *With an overall 85.34% of success rate, the experimental results show that* RepairDroid *is effective in automatically repairing real-world Android apps.*

4.4 RQ4: Time Performance of *RepairDroid*

In our last research question, we investigate the time performance of *RepairDroid* when applied to repair real-world published Android apps. Taking the same 1,000 apps as leveraged for answering RQ3 as input, Figure 4 presents the distribution of the execution time that *RepairDroid* spends for analyzing an app. The distribution is plotted with respect to the number of located compatibility issues identified in an app and the DEX size of each app, respectively. As the number of located issues per-app increases, the time spent to repair the app also increases. Nevertheless, the increase seems to be gradual. Indeed, the Pearson correlation coefficient confirms that there is only a weak correlation ($r = 0.34, p - value < 0.001$) between them. When DEX size is concerned, we cannot observe direct connections between the DEX size of the app and the time that *RepairDroid* spends to repair the app. Pearson correlation analysis also confirms our observation that there is literally no correlation ($r = -0.07, p - value = 0.05$) between them.

The results suggest that **the time performance of *RepairDroid* is quite stable in analyzing a real-world Android apps**, no matter how large the code size is or how many compatibility issues it suffers from. This evidence further suggests that our approach is suitable to be applied to repair large-scale Android apps.



(a) # Located Issues. (b) DEX Size.

Figure 4: Distribution of the execution time with respect to the number of issues contained per app and its DEX size, respectively.

➽ *The time spent by* RepairDroid *to repair a real-world Android app is stable, with no impact by the app's code size and only a slight impact by the number of compatibility issues that needed to be fixed.*

5 DISCUSSION

5.1 Limitations

While *RepairDroid* aims for precise and sound analysis and repair, it does share some inherent limitations with most other static analysis tools. *RepairDroid* is oblivious to reflective calls, native code, multi-threading features, which may impact the static analysis results and hence leading to inaccurate locations of compatibility issues. Currently *RepairDroid* is only aware of the most common expression types defined in Soot. In some rare cases, when the reserved keywords such as [CUT] and [PASTE] involve unusual expressions, *RepairDroid* may not be able to recognize them and hence will not generate executable statements, resulting in incorrect repairs. However, it is relatively easy to extend *RepairDroid* for including more of Soot's expression types. This limitation could be mitigated in practice. Furthermore, we have only evaluated the correctness of device-specific compatibility issue repairings through manual confirmation because we cannot find the relevant devices for testing. Such manual processes are, however, known to be error-prone. To mitigate the threat, we have cross-validated the results. We have also released our tool and dataset for public reference.

5.2 Mining Fix patterns from existing Android apps

RepairDroid is limited by the set of templates prepared for analysis. The more templates created and included, the more compatibility issues will likely be automatically fixed. However, it might be non-trivial to create templates for some issues as it could require complicated background knowledge to understand the issues and their corresponding fixes. This burden could be significantly mitigated if we can locate real-world code examples relevant to fix compatibility issues. Following the idea of Fazzini et al. [8], who propose to learn fix patterns from the evolution of open-source Android apps, we propose to mine such fix patterns from the evolution of published Android apps. We believe the evolution of published Android apps could provide much more knowledge than mining open-source apps since the majority of apps have only released their published versions (i.e., source code not available).

Table 5: The list of the top-5 API Pairs mined from real-world Android apps.

Incompatible API	Replacement API	# Apps
setBackgroundDrawable	setBackground	5,382
isScreenOn	isInteractive	2,129
getDrawable(int)	getDrawable(int,Theme)	1,543
setOnUtteranceCompletedListener	setOnUtteranceProgressListener	1,240
getColorStateList(int)	getColorStateList(int,Theme)	1,085

To this end, we randomly selected 20,000 apps from AndroZoo [3] and conducted a lightweight static analysis to locate API calls that are protected by SDK version checks (e.g., based on the following pattern: if (condition) $API_{incompatible}$ else $API_{replacement}$). To ensure that the located API calls are indeed relevant to fix compatibility issues, we further resort to the list of 228 API deprecated-and-replacement pairs disclosed by Li et al. [20]. We consider a potential fix pattern is located if the following conditions are satisfied: (1) the API calls are protected by SDK version checks, (2) the problematic API and its replacements are respectively presented in the two branches separated by the SDK version check. Among

the 228 pairs, we are able to locate 34 practical fix patterns, for which the top-5 ones are illustrated in Table 5. Following the located fix patterns, we could indeed easily create templates for the involved problematic APIs. Those templates can subsequently help *RepairDroid* in automatically repairing the pinpointed issues in the corresponding real-world apps. This preliminary study experimentally shows that our approach *RepairDroid* could indeed benefit from existing compatibility issue fixes conducted by developers in practice. This study is our initial attempt and is part of our ongoing efforts at creating more templates for repairing compatibility issues in published apps.

5.3 Supporting automated repair for both Java and Kotlin written Android apps

Since 2017, Google has promoted Kotlin as the official programming language for developing Android apps. In 2019, Google further declared that Android becomes 'Kotlin-first' (i.e., new API, libraries, documentation will target Kotlin first) and hence advised developers to develop new apps using Kotlin instead of Java. Since then, more and more Android apps have completely migrated from Java to Kotlin. However, to the best of our knowledge, all existing app repairing approaches only focus on Java written apps, letting a large number of Kotlin written apps untouched. Indeed, since Java and Kotlin are two different programming languages, code repairing approaches proposed for one language cannot be applied to the other. Analysts have to completely rewrite the code parser and repair module following the new languages' syntax. We hence argue that there is a need to support automated repair for both Java and Kotlin written apps.

Fortunately, no matter which language is leveraged, the published Android apps will be in Dalvik bytecode. Therefore, *RepairDroid* could be directly applied to repair both Java and Kotlin written apps. For example, *RepairDroid* tool can perform successful fixes on the DuckDuckGo-Kotlin [1] app.

6 RELATED WORK

Compatibility Analysis: Compatibility issues have been a key research topic in the Android community [14, 26, 30–32]. To assist developers in exhaustive app testing, Wei et al. [31] empirically study the fragmentation-induced issues to characterize the symptoms and root causes and propose a technique named FicFinder to detect such compatibility issues. After that, the authors [32] further present an API-device correlation extracting and learning approach named Pivot to help detect fragmentation-induced compatibility issues. Huang et al. [14] delve into the callback API evolution induced compatibility issues and provide a technique named CIDER, leveraging a graph-based model to detect two types of callback compatibility issues. Unfortunately, both Pivot and CIDER focus on detecting some types of incompatibility issues instead of repairing them, the motivation of our *RepairDroid*.

The exploration of compatibility issues caused by Android OS evolution is needed as apps are inseparably linked to the official Android APIs. Researchers have put a lot of effort into deprecated APIs [8, 10, 12, 20, 21, 34], which could eventually lead to compatibility issues. Li et al. [20] build a prototype tool called CDA and apply it to different revisions of the Android framework to characterize deprecated Android APIs. Based on an extensive empirical study, He et al. [12] reveal that drastic API changes exist between neighboring Android versions. They have additionally developed a tool named IctApiFinder to detect incompatible API usages. Similarly, Li et al. [19] propose an approach named CiD to model the lifecycle of the Android APIs and flag the error usages capable of causing compatibility issues, the issues declared by which are also regarded as one of our motivations. Xia et al. [33] perform a large-scale study on the practice of handling OS-induced API compatibility issues and their solutions, and propose a tool, RAPID, to ascertain whether a compatibility issue has been addressed.

In non-Android communities, research on API deprecated is also ubiquitous [4, 13, 23, 25, 35]. Zhou et al. [35] study API deprecation usage in open-source Java frameworks and libraries. They also propose a framework to detect deprecated API usages in source code examples on the Web. Brito et al. [4] perform a large-scale analysis of real-world Java systems and reveal that there is almost no significant effort to improve deprecation messages. Some researchers concentrate on the impact of API deprecation [6, 13, 23, 25]. Hora et al. [13] report on an exploratory study that aims to observe API evolution and its impact on the Pharo ecosystem. Sawant et al. [25] extend the study on Java and investigate how many API clients update their dependencies to actively maintain their projects and count the number of affected projects by deprecation.

Program Repair: Program transformations and repairing have been widely researched [5, 7, 15, 16, 18, 22, 24, 29]. For instance, LASE [15, 22] is an example-based program repair tool by learning non-trivial data and its context from multiple editing examples and automatically searching for editing locations to apply customized editing to these locations. Rolim et al. [24] present REFAZER, a technique based on the code edits performed by developers for automatically learning program transformations. Coccinelle [5, 18] is a C-based program matching and source-to-source transformation tool that has been employed for the automatic evolution of the Linux kernel. Coccinelle provides Semantic Patch Language (SmPL) to write its transformation rules. As a Java extension to Coccinelle, Coccinelle4J [16] is designed to apply for Java programs. Similar to Coccinelle, it uses semantic patches written in SmPL.

Recently, studies have begun to focus on the usage of automatically updating incompatible Android APIs. As one of the first tools to implement this goal, AppEvolve [8] using GitHub as the code base to perform API updates by learning examples before and after the update. Haryono et al. [10] improve AppEvolve by proposing CocciEvolve, which uses a single updated example to perform API updates and provides readable and configurable scripts in the form of semantic templates. They further broaden their study by proposing AndroEvolve [11], which addresses the defects of CocciEvolve with data flow analysis and variable name denormalization. Similar to their works, *RepairDroid* also rely on semantic templates to automatically perform API updates. Lamothe et al. [17] leverage the basic *diff* in the version control system to learn API migration patterns, where they use the ASTs to match the API calls in the source code to the code examples. Unlike the above researches that directly act on the Java source code, our study concentrates on the low-level programming language, intending to modify Dex files directly and break the limitation that they can only take effect within the scope of a method or file.

7 CONCLUSION

We have proposed a novel prototype tool *RepairDroid* for automatically repairing compatibility issues in published Android apps. *RepairDroid* provides a patch description language for users to create fix templates for given compatibility issues. *RepairDroid* then applies these created templates directly to the Android app bytecode to repair the corresponding compatibility issues. Experimental results show that the patch description language is generic, being able to correctly describe 42 out of 44 issues, and the repair module is effective, being able to outperform the state-of-the-art approaches and achieve an overall 85.34% of success rate in repairing 1,000 real-word Android apps. *RepairDroid*'s execution time per app is also stable, making it suitable to be applied to conduct market-scale repairings.

ACKNOWLEDGEMENTS

This work is supported by ARC Laureate Fellowship FL190100035, Discovery Early Career Researcher Award DE200100016 and Discovery Project DP200100020.

REFERENCES

[1] 2017. DuckDuckGo Android App. https://github.com/duckduckgo/Android.
[2] 2021. Towards Automatically Repairing Compatibility Issues in Published Android Apps. https://zenodo.org/record/5430715.
[3] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 468–471.
[4] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 2016. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 360–369.
[5] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. 2009. A foundation for flow-based program matching: using temporal logic and model checking. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 114–126.
[6] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.
[7] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael O'Boyle. 2020. M3: Semantic API Migrations. *arXiv preprint arXiv:2008.12118* (2020).
[8] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 204–215.
[9] Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Negative Results on Mining Crypto-API Usage Rules in Android Apps. In *The 16th International Conference on Mining Software Repositories (MSR 2019)*.
[10] Stefanus Agus Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example. *arXiv preprint arXiv:2005.13220* (2020).
[11] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. 2020. AndroEvolve: Automated Android API Update with Data Flow Analysis and Variable Denormalization. *arXiv preprint arXiv:2011.05020* (2020).
[12] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and detecting evolution-induced compatibility issues in android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 167–177.
[13] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 251–260.
[14] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.
[15] John Jacobellis, Na Meng, and Miryung Kim. 2013. LASE: an example-based program transformation tool for locating and applying systematic edits. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1319–1322.
[16] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. 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.
[17] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering* (2020).
[18] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 USENIX Annual Technical Conference*. 601–614.
[19] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.
[20] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 254–264.
[21] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. CDA: Characterising Deprecated Android APIs. *Empirical Software Engineering* (2020), 1–41.
[22] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511.
[23] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[24] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.
[25] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2018. On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering* 23, 4 (2018), 2158–2197.
[26] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 288–298.
[27] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automated Deprecated-API Usage Update for Android Apps: How Far are We?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 602–611.
[28] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[29] Eelco Visser. 2001. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *International Conference on Rewriting Techniques and Applications*. Springer, 357–361.
[30] Haoyu Wang, Hongxuan Liu, Xusheng Xiao, Guozhu Meng, and Yao Guo. 2019. Characterizing Android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 280–292.
[31] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.
[32] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning API-device correlations to facilitate Android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 878–888.
[33] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 886–898.
[34] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. 2018. How do Android operating system updates impact apps?. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 156–160.
[35] Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 266–277.