How Android Developers Handle Evolution-induced API Compatibility Issues: A Large-scale Study

Hao Xia*
Fudan University
haoxia17@fudan.edu.cn

Yuan Zhang*
Fudan University
yuanxzhang@fudan.edu.cn

Yingtian Zhou
Fudan University
yingtianzhou17@fudan.edu.cn

Xiaoting Chen
Fudan University
xtchen16@fudan.edu.cn

Yang Wang
Fudan University
16307130325@fudan.edu.cn

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Shuaishuai Cui
Fudan University
sscui16@fudan.edu.cn

Geng Hong
Fudan University
ghong17@fudan.edu.cn

Xiaohan Zhang
Fudan University
xh_zhang@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

ABSTRACT

As Android platform evolves in a fast pace, API-related compatibility issues become a significant challenge for developers. To handle an incompatible API invocation, developers mainly have two choices: merely performing sufficient checks to avoid invoking incompatible APIs on platforms that do not support them, or gracefully providing replacement implementations on those incompatible platforms. As providing more consistent app behaviors, the latter one is more recommended and more challenging to adopt. However, it is still unknown how these issues are handled in the real world, do developers meet difficulties and what can we do to help them.

In light of this, this paper performs the first large-scale study on the current practice of handling evolution-induced API compatibility issues in about 300,000 Android market apps, and more importantly, their solutions (if exist). Actually, it is in general very challenging to determine if developers have put in countermeasure for a compatibility issue, as different APIs have diverse behaviors, rendering various repair. To facilitate a large-scale study, this paper proposes `RAPID`, an automated tool to determine whether a compatibility issue has been addressed or not, by incorporating both static analysis and machine learning techniques. Results show that our trained classifier is quite effective by achieving a F1-score of 95.21% and 91.96% in the training stage and the validation stage respectively. With the help of `RAPID`, our study yields many interesting findings, e.g. developers are not willing to provide alternative implementations when handling incompatible API invocations (only 38.4%); for those incompatible APIs that Google gives replacement recommendations, the ratio of providing alternative implementations is significantly higher than those without recommendations; developers find more ways to repair compatibility issues than Google's recommendations and the knowledge acquired from these experienced developers would be extremely useful to novice developers and may significantly improve the current status of compatibility issue handling.

*co-first authors

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; **• General and reference → Empirical studies**.

KEYWORDS

Compatibility Issues, API Evolution, Android App Analysis

ACM Reference Format:
Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. 2020. How Android Developers Handle Evolution-induced API Compatibility Issues: A Large-scale Study. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380357

1 INTRODUCTION

Recent years have witnessed the rapid growth of Android-powered devices. To keep up with the emerging needs for new functionalities and improve user experience, Google frequently updates its Android operating system. During the evolution of Android platform, many new APIs are introduced, while some APIs are removed. Google designates a dedicated API level for each major OS release to label the APIs that can be used on this OS release [8]. To safely invoke an Android API, developers should check the current API level of the running platform (which can be accessed via *android.os.Build.SDK_INT*, *SDK_INT* for short), before the invocation. Otherwise, invoking an unsupported API will lead to a "*NoSuchMethodError*" and crash. This kind of issue is called an *evolution-induced API compatibility issue* [16, 24].

With the recent release of Android P, there are 28 different API levels [10] in total and each level defines tens of thousands of Android APIs. Obviously, it is quite challenging and error-prone for Android developers to manage app compatibility against different API levels [18, 35]. Even worse, Android Lint (which is the default app bug detector integrated by the official Android App IDE, Android Studio) is ineffective in helping developers perform API compatibility checks, because it performs context-insensitive analysis and does not cover the integrated external libraries. As a result, evolution-induced API compatibility issues are prevalent. For example, by querying Google with "*Android NoSuchMethodError*", 193,000 entries return (till April 22, 2019). A recent study shows that 91.84% of FDroid apps write specific code to perform API level checks and 25.33% of them have evolution-induced incompatible API usage [16]. To help developers to test whether they perform sufficient compatibility checks, Li et al. [24] and He et al. [16] systematically model the lifecycle of every Android API and leverage heavyweight static analysis.

However, performing sufficient compatibility checks is not enough to deal with evolution-induced API compatibility issues. In fact, it just prevents the app from raising a "*NoSuchMethodError*". For example, when Google removes the API *Resources.getColor(int)*, it suggests developers invoke another API *Resources.getColor(int, Theme)* instead [6]. If an app just performs a check with *SDK_INT* before invoking *Resources.getColor(int)* without providing a replacement API on those incompatible API levels, the app would incur an inconsistency bug. To better deal with the compatibility issue in this example, developers should invoke another API (such as the recommended *Resources.getColor(int, Theme)* method) on those incompatible API levels. Based on the above observation, this paper distinguishes two kinds of checks against evolution-induced compatibility issues.

(1) The first kind of check just aims to prevent the app from crash by simply avoiding the invocations of APIs that are not supported by the underlying platform. This kind of check is named as SigChk for short (explained later in §2.2).
(2) The second kind of check not only guarantees that APIs with compatibility issues are invoked when supported, but also provides similar functionalities to these APIs when not supported. We call this kind of check as RplChk for short (explained later in §2.2).

To provide better app compatibility against different API levels, developers are expected to implement consistent app behavior across all API levels they claim to keep compatible with.

Previous works [16, 24] only manually analyzed several cases of how evolution-induced compatibility issues are checked. However, their study only covers a small number of cases. There is no large-scale study to measure the current practice of developers in handling evolution-induced incompatible API invocations, e.g. do developers tend to replace them or just simply check them, how different compatibility-related APIs are replaced, what typical difficulties do developers encounter in finding ways to replace them, can novice developers be benefited from learning the practice of experienced developers.

In light of this, this paper aims to perform a large-scale study on real-world Android market apps (about 300,000 apps) to shed light on the practice of developers in dealing with evolution-induced API compatibility issues. To the best of our knowledge, this paper is the first to perform such a study with a large volume of real-world apps. The results reported by our study can facilitate an in-depth understanding of the current status in handling app compatibility issues, and enable the community to conduct more effective efforts to help developers fix compatibility issues. Except for evolution-induced API compatibility issues, Android apps are also vulnerable to other kinds of compatibility issues [46], such as device-specific issues [26]. In this paper, we mainly consider evolution-induced API compatibility issues, because of their prevalence [16], severe effects, and the feasibility of systematically modeling these issues [24]. In the following, we shortly name evolution-induced API compatibility issues as compatibility issues when not specifically mentioned.

To facilitate the large-scale study, we need an automated tool which can classify compatibility checks into SigChk and RplChk. However, building such a tool entails challenging program analysis. Since there are a lot of compatibility-related APIs (more than 10,000 APIs as reported in §2.1) and different APIs may be replaced in specific ways, we cannot directly search all possible alternative implementations for every compatibility-related API to provide appropriate classification. As demonstrated in the large-scale study (see §4), developers have various ways to implement alternative functionalities to the compatibility-related APIs.

To this end, we propose a tool, named `RAPID` (**R**eplacement **API** **D**etection) to automatically recognize RplChk. The intuition of `RAPID` originates from the definition of RplChk, i.e. a compatibility check which provides a replacement operation when the API is not supported. Thus, if there is a similar operation to the API in the other branch, it is RplChk. Otherwise, it is SigChk. However, it is quite difficult to judge whether there is a similar operation to an API in the other branch, because each API provides different functionalities and exhibit diverse behaviors. By manually analyzing about one hundred real-world SigChk and RplChk cases, we have some interesting findings:

- We find that the alternative operation in RplChk shares similar semantics with the API, which is often reflected in the semantic similarity of the corresponding natural language artifacts (e.g. API names). For example, the recommended alternative API *Resources.getColor(int, Theme)* by Google is quite similar to the original API *Resources.getColor(int)* in terms of the API names.
- We find that the arguments that an alternative operation takes are similar to the arguments of the original API, and the return value of an alternative operation may flow to similar points as the return value of the original API.

Based on these observations, we propose a learning-based approach to automatically determine if a compatibility check is RplChk or not. Specifically, we extract 19 features from code blocks at each compatibility check point and encode them into numeric vectors. We manually label 293 compatibility checks and utilize machine learning to train a classifier. The model reports a precision of 96.76% and a recall of 93.72% in the training stage. Besides, we also randomly select 414 compatibility checks from the study to manually validate the effectiveness of our trained classifier. The result shows that it achieves a precision of 93.41% and a recall of 90.58%, which we believe is good enough for a large-scale study.

We collect about 300k apps from 5 popular Android app markets to study the practice of handling compatibility issues in the real world. Our study yields many important and interesting findings that have never been reported before, e.g.: 1) we find that only 38.4% of compatibility checks actually provide replacement implementations on those incompatible API levels; 2) developers are likely to handle incompatible API invocations with alternative operations when Google provides recommendations, although Google only gives recommendations for very few APIs; 3) developers do not always follow the recommended way when providing replacement implementations, especially when the recommended API needs more arguments to invoke and hence is not the easiest way to handle the issue; 4) developers can find their own way to provide replacement implementations for those incompatible APIs when Google does not give recommendations; 5) not all developers succeed in providing replacement implementations and the current status of handling compatibility issues can be significantly improved by simply learning from experienced developers.

In all, this paper makes the following contributions:

- **Large-scale study to understand the current practice of compatibility issue handling.** This paper is the first to distinguish two kinds of compatibility issue checks: SigChk and RplChk, and performs the first such large-scale study to measure the real-world practice of handling compatibility issues. Our study reveals many important findings that are expected to stimulate more relevant works to improve the compatibility issue fixing.
- **The RAPID tool that can automatically classify two kinds of compatibility checks.** It is quite challenging to automatically classify SigChk and RplChk. This paper designs a learning-based approach to automatically determining if a compatibility issue check is a RplChk or not, based on a comprehensive set of features from texts, data dependencies, etc. Despite from facilitating a large-scale study, our tool is also very useful to help developers to check whether they have properly performed compatibility checks.

Note that our goal is to distinguish two kinds of compatibility issue checks and study the current fix practice on a large scale, while not aiming to automatically fix/repair incompatible APIs.

The rest of this paper is organized as follows. § 2 introduces our definition of two kinds of compatibility checks and presents the motivation of this paper as well as the challenges we met. § 3 presents the design of our RAPID tool which is built to automatically classify SigChk and RplChk. § 4 evaluates the performance of RAPID and reports the results drawn from our large-scale study. § 5 discusses the limitations of our work. § 6 summarizes the related work and § 7 concludes the paper.

2 PROBLEM STATEMENT

2.1 Evolution-induced Compatibility Issues

Android platform is frequently updated to meet various emerging requirements and to optimize the user experience. For example, there have been two or more major platform updates almost every year in the past decade. The evolution of the Android operating system brings significant burdens for developers to keep their apps compatible with different system versions. To help developers manage app compatibility, Google designates an *API level* for each major system version, which defines all the APIs that can be accessed on this version. If an app invokes an API that is not defined by the API level, it will trigger a "*NoSuchMethodError*" and crash.

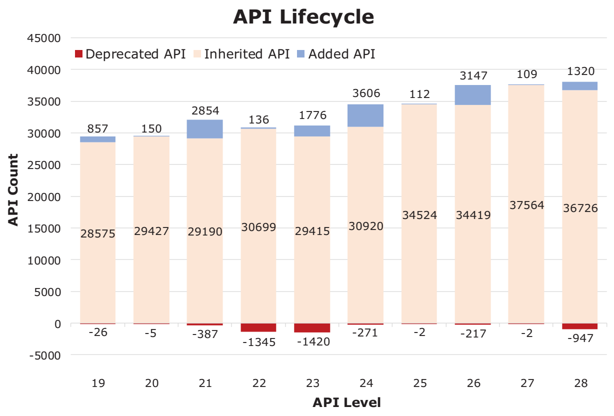


Figure 1: API Lifecycle ranging from API Level 19 (Android 4.4) to 28 (Android P). This figure shows the numbers of APIs that are deprecated/added at this API level and those inherited from the previous API level.

While it is unnecessary for an app to be compatible with all API levels, app developers usually declare the API levels that their app should be compatible with in the app manifest file (aka. *AndroidManifest.xml* [9]). There are three attributes in the manifest file that can be used to declare the compatibility level: 1) *minSDKVersion* defines the minimum API level that the app is compatible with, and Android ensures that the app cannot be installed on devices below this API level; 2) *targetSDKVersion* defines the most appropriate API level that the app is designed to run on; 3) *maxSDKVersion* is previously used to define the maximum level, but this attribute is deprecated since Android 2.1. Among the three attributes, *minSDKVersion* is the most important because it determines the range of API levels that an app should ensure compatibility. For example, if an app sets *minSDKVersion* to 21, it should ensure compatibility with the API levels from 21 to 28 (the latest API level of Android).

API Lifecycle Database. During API evolution, some APIs are deprecated and new APIs are introduced. Android maintains the available API level range for each occurred API in api-versions.xml [11] of Android SDK. By analyzing this file, we can build a database to figure out the APIs that can be invoked at each API level. Figure 1 shows the number of APIs that each API level defines. From this figure, we can find that a lot of APIs are added/removed during each API level update.

2.2 Two Kinds of Compatibility Checks

To handle evolution-induced API compatibility issues, developers need to perform API level checks of the running platform before invoking a compatibility-related API. Specifically, for those newly-introduced APIs, app developers should guarantee that these APIs

```
if (Build.VERSION.SDK_INT >= 22) {
    ((AccountManager)v0).removeAccountExplicitly(v1);
}
```

(a) Compatibility Issue Check with if-then CFG Structure

```
if (Build.VERSION.SDK_INT >= 22) {
    ((AccountManager)v0).removeAccountExplicitly(v1);
} else {
    ((AccountManager)v0).removeAccount(v1, null, null);
}
```

(b) Compatibility Issue Check with if-then-else CFG Structure

```
if (Build.VERSION.SDK_INT >= 22) {
    ((AccountManager)v0).removeAccountExplicitly(v1);
} else {
    logger.warn("Just a log");
}
```

(c) Compatibility Issue Check with if-then-else CFG Structure

Figure 2: Different cases of checking compatibility issues. In this figure, *AcccountManager.removeAccountExplicitly(X1)* is added since API level 22 and *AcccountManager.removeAccount(X1, X2, X3)* is deprecated since API level 22 [5].

are not invoked on the previous API levels; for those deprecated APIs, app developers should ensure that these APIs are not invoked on the subsequent API levels. In practice, different checks can be performed for a compatibility-related API invocation. Figure 2 shows an example of several cases of checking compatibility issues. Based on the aims of different compatibility issue checks, we differentiate two kinds of compatibility issue checks:

(1) SigChk: Single Compatibility Issue Check. A simple way to handle compatibility issues is to perform a check against *SDK_INT* (which returns the API level of the running platform) before invoking any compatibility-related API. Figure 2(a) gives an example, where *AcccountManager.removeAccountExplicitly(X1)* is added since API level 22, and hence the code snippet tests whether *SDK_INT* is larger than or equal to 22. By placing such API level checks before invoking compatibility-related APIs, developers can effectively prevent "*NoSuchMethodError*". From Figure 2(a), we observe that this kind of check does nothing but just checking the API level of the underlying platform. We call such practice to handle compatibility issues as SigChk for short.

(2) RplChk: Replaced Compatibility Issue Check. Although SigChk can prevent apps from crashes, it is not the best effort to handle compatibility issues. A better way to handle evolution-induced compatibility issues is to provide an alternative implementation/function that has similar functionalities when an API is not compatible. Take Figure 2(a) as an example, the remove account operation is only performed when the API level is larger than 22. It would lead to inconsistent behaviors on devices whose API level is below 22. In contrast, Figure 2(b) gives a better way to handle the compatibility issue by invoking another API with similar functionalities on API levels lower than 22. We call such a way to handle compatibility issues as RplChk for short.

2.3 Motivation

Obviously, RplChk is more desirable than SigChk. When some APIs are deprecated, Google may recommend developers to use other APIs to replace the incompatible ones. For example, when Google removes the API *Resources.getColor(int)*, it suggests developers to invoke another API *Resources.getColor(int, Theme)* as an alternative [6]. However, not all deprecated APIs have suggested replacement APIs. Meanwhile, for APIs that are newly introduced, there are no recommendations on earlier API levels. When there are no official recommendations to handle a compatibility issue, developers often resort to their own efforts to implement similar functionalities. Sometimes it is almost impossible to fix a compatibility issue in this way, because the feature provided by the compatibility-related API may only be implemented by the platform. For example, *WifiManager.startScan()* is deprecated since API level 28, which requests a scan for Wifi access points [7]. Except for this API, an app cannot implement similar functionality because a normal app does not have the privileges to control Wifi devices for access point scanning.

Based on the above observations, we can find that it is quite challenging for developers to fix compatibility issues with replacement implementations. Previous work leverages manual analysis of open source apps to gain insights into how compatibility issues are fixed [46]. However, their study only covers a small number of apps, and thus cannot give a representative and quantitative measurement about the status of compatibility issue handling. In light of this, our paper aims to perform a large-scale study to measure the current practice of developers in dealing with compatibility issues, such as: how many compatibility issues are addressed by providing alternative functionalities, do Google's recommendations help developers in handling these issues. Besides facilitating an in-depth understanding of compatibility issues and their fixes, the insights acquired through our study may have substantial ramifications on future research about automatically addressing such issues.

2.4 Challenges

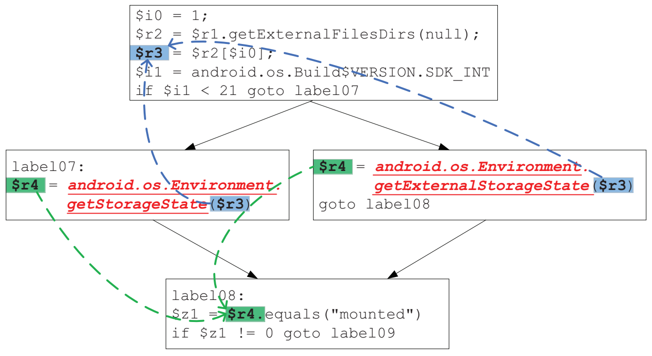
To facilitate such a large-scale study, we first need a tool which can automatically and reliably classify SigChk and RplChk. However, it is challenging to achieve this goal, because there are a large number of compatibility-related APIs (more than 10,000 APIs as reported in §2.1), and different APIs have different functionalities such that their invocations can only be replaced in specific ways. Thus, we cannot simply crawl all possible alternative implementations for every compatibility-related API to conduct classification. Specifically, there are two challenges to differentiate RplChk from SigChk:

Challenge-I: Recognizing the control structure of compatibility checks does not reliably distinguish RplChk from SigChk. One may observe that a SigChk usually has only a single true branch after the check of *SDK_INT*, while a RplChk always has both true branch and false branches. However, as depicted in Figure 2(c), not all compatibility checks with both branches are RplChk. The case in Figure 2(c) is actually a SigChk.

Challenge-II: Measuring the similarity between the true and false branches of compatibility check does not accurately differ RplChk and SigChk. To differ Figure 2(b) from Figure 2(c), one may propose to use code block similarity, since the two branches of the *SDK_INT* check in Figure 2(b) are quite similar while the two branches in Figure 2(c) are dissimilar. However, this rule is not reliable. Figure 3 gives an example in which the two branches are quite similar. However, by examining this case, we can

```
if (Build.VERSION.SDK_INT >= 16 ) {
   Intent i = new Intent(this, CameraGallery.class);
   i.putExtra("from", "hoarding");
   ActivityOptions op = ActivityOptions.makeCustomAnimation
                        (this, R.anim.anim1, R.anim.anim2);
   this.startActivity(i, op.toBundle());
} else{
   Intent i = new Intent(this, CameraGallery.class);
   i.putExtra("from", "hoarding");
   this.startActivity(i);
}
```

Figure 3: A SigChk case where there is no alternative operation in the else branch for *ActivityOptions.makeCustomAnimation()*. This case demonstrates that the identification of RplChk requires fine-grained analysis.



```
$i0 = 1;
$r2 = $r1.getExternalFilesDirs(null);
$r3 = $r2[$i0];
$i1 = android.os.Build$VERSION.SDK_INT
if $i1 < 21 goto label07
```

```
label07:
$r4 = android.os.Environment.
getStorageState($r3)
```

```
$r4 = android.os.Environment.
getExternalStorageState($r3)
goto label08
```

```
label08:
$z1 = $r4.equals("mounted")
if $z1 != 0 goto label09
```

Figure 4: Example to demonstrate the features that can be used to locate replacement API.

find that the functionality provided by the compatibility-related API *ActivityOptions.makeCustomAnimation()* is not provided in the other branch. To reliably differ RplChk from SigChk, we need to analyze the fine-grained code semantics.

To address these challenges, this paper proposes a learning-based approach, RAPID (**R**eplacement **API** **D**etection), which can automatically classify RplChk and SigChk by leveraging static analysis and machine learning. In the following, we will elaborate on the technique.

3 RAPID APPROACH

3.1 Overview

By manually analyzing about one hundred real-world SigChk and RplChk cases, we gain some insights about determining whether a compatibility check is SigChk or RplChk. Basically, we follow two steps.

Firstly, we examine the control structure of the check on *SDK_INT*. If the check is a *"if-then"* check, it is directly flagged as SigChk, since there is no way for the app to provide similar functionalities to the compatibility-related API in this scenario (see Figure 2(a) for an example). Note that we focus on checks on *SDK_INT* because according to recent works [16, 24] that detect missing checks of incompatible API invocations, developers mostly use the attribute *SDK_INT* to check API compatibility.

Secondly, if the check is a *"if-then-else"* check, we cannot simply flag it as RplChk. Our core idea to classify this kind of check is inspired by the definition of RplChk, i.e., whether the app provides similar functionalities to the problematic API on unsupported API levels.

Since there is a large search space for the possible replacements of an incompatible API invocation, we usually cannot give a definitive answer. Instead, we disclose the characteristics that RplChk exhibits from the experience in manual inspection of a number of real-world SigChk and RplChk cases, and design a learning-based approach to facilitate such classification. Specifically, we find RplChk cases share many common features from two perspectives that SigChk cases do not have: *behavior semantics* and *input/output dependencies.*

Behavior Semantics. The names of Android API classes, methods and fields contain rich semantic. In RplChk cases, we observe that the alternative operations to an incompatible API often share very similar semantics in their natural language artifacts (e.g. names). Figure 4 gives an example, where *Environment.getExternalStorageState()* is used to replace *Environment.getStorageState()* on those incompatible API levels. We can observe that the two APIs are from the same class. Besides, from their names, it is not hard to infer that they provide very similar functionalities. In contrast, in the SigChk case shown in Figure 2(c), it is easy to recognize that *Logger.warn()* and *AccountManager.removeAccountExplicitly()* are likely to be semantically different. Note that only names of API classes, methods and fields are considered here, because they belong to the public interface of Android system that should not be obfuscated [12, 14, 45]. In contrast, app-specific artifacts (e.g., functions defined locally by an application) are precluded in our analysis.

Input/Output Dependencies. Besides behavior semantics, we also observe that if an API can be used to replace another one. They are expected to accept very similar inputs and the outputs generated may be used in a similar way. The RplChk case in Figure 4 can be used as an example to illustrate such a similarity. We can find that the arguments of *Environment.getExternalStorageState()* and *Environment.getStorageState()* are the same (i.e. $r3), and the return value of them are also the same (i.e. $r4). Note that not all the APIs that can replace each other have the same inputs and outputs. Meanwhile, for the SigChk cases in Figure 2(c), there is no operation in the false branch which shares similar inputs and outputs with *AccountManager.removeAccountExplicitly().*

RAPID Workflow. We present RAPID which leverages the aforementioned insights to automatically classify RplChk and SigChk. Figure 5 depicts the overall workflow of RAPID. First, RAPID identifies all compatibility checks using static analysis, and also extracts the branches of each check. To identify compatibility checks, RAPID leverages the API Lifecycle Database constructed in §2.1. Second, RAPID extracts features from the branches and encodes them into vectors. At last, we choose Support Vector Machine (SVM) (for its over-fitting resistant feature) to train a classifier for RplChk identification. In the following, we will detail the design of the two modules in RAPID.

3.2 Static Analysis

3.2.1 Compatibility Check Identification. In order to recognize RplChk, RAPID needs to first identify compatibility checks. Actually,

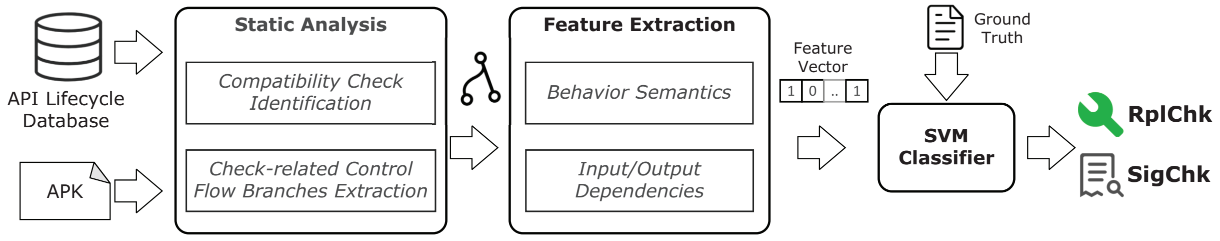


Figure 5: Workflow of RAPID. It first identifies all compatibility checks and check-related control flow branches with static analysis, and then extracts features from the branches to encode them into vectors. Finally, SVM is trained to give a classification result.

previous works [16, 24, 46] on detecting missing compatibility checks have explored many ways for compatibility check identification. This part of RAPID is inspired by these works, and we do not claim contributions. The key contributions of RAPID lie in the classification of SigChk and RplChk which will be elaborated.

Nevertheless, we give an overview of this part for the sake of completeness. Specifically, RAPID follows three steps to identify compatibility checks. First, we leverage FlowDroid [1] to build a precise inter-procedural control flow graph (ICFG) for each app. During ICFG construction, FlowDroid generates a dummy main method to connect all app components and callback functions. Second, by traversing the ICFG, RAPID locates all compatibility-related API invocations in the app. In order to find all compatibility-related APIs for an app, RAPID looks up the API Lifecycle Database which is constructed in §2.1 with the *minSdkVersion* attribute in the *AndroidManifest.xml* [9] of the app. At last, we traverse backward to find the API level check against *SDK_INT* for each located API.

3.2.2 Check-related Control Flow Branches Extraction. After identifying compatibility checks, we further extract the control flow graphs of these checks for the following RplChk classification. If a check has a *"if-then"* control structure, we can quickly recognize it as a SigChk. Thus, only the compatibility checks with two branches (i.e. *"if-then-else"* control structure') proceed to this step. To differ the two branches, we call the branch that invokes the API the *api-branch*, and call the other branch *candidate-branch*. The goal of RAPID is to find an alternative operation in the candidate-branch which provides similar functionalities to the incompatible API, if it exists. As indicated in [16], the most common fix practice is to invoke an alternative API (from a different API level). Besides, combined with our experience in manual inspection, we focus on two kinds of operations in the candidate-branch to reduce the search space: 1) *method-invocation operation* which invokes an API; 2) *field-access operation* which gets/sets a field of a system class. Specifically, we name each operation as *UnitOp*, including the compatibility-related API invocation in the api-branch.

Our compatibility check branch extraction works by transforming the statements in the candidate-block into a list of *UnitOp*s. During the transformation, we also inline the app methods that are invoked by the candidate-branch since an alternative operation may be inside an app method. Besides, we recognize Java Reflection API calls in the candidate-branch and transform them into normal Java statements with the help of constant propagation. Note that we do not employ a dedicated tool (such as DroidRA [23]) to tame the reflection calls here because our scenario is not an adversarial setting and the cases we meet are quite standard to handle. For each *UnitOp*, we also perform intra-block data flow analysis to track the flows to the arguments of an *UnitOp* and the propagation of its return value.

3.3 Feature Extraction

We further extract features for each identified *UnitOp* to train a classifier to identify RplChk. If a *UnitOp* in the candidate-branch provides similar functionalities to the incompatible API invocation *UnitOp*, the check is recognized as RplChk. Specifically, the feature extraction module extracts features from two perspectives as mentioned before: behavior semantics and input/output dependencies.

Overall, we extract 19 features for every *UnitOp* pair (i.e., one is the alternative of the other) and each feature is represented as a floating point value. Before we detail the extraction for these 19 features, we introduce 3 similarity functions as follows:

- *Jaccard [48] similarity* models the similarity between two sets based on how many common items they share.
- *Word similarity* utilizes the word2vec [49] model to calculate the semantic similarity between two words.
- *Word set similarity* leverages the word2vec model to calculate the similarity between two word sets. Specifically, we adopt an algorithm of calculating sentence similarity [38] which works by calculating the average word similarity between all words in the two sets.

3.3.1 Feature Extraction for Behavior Semantics. We notice that names and comments of code elements in Java code usually covey a lot of semantic information. Therefore, we leverage the text similarity between two *UnitOp*s to represent their closeness in behavior semantics. Specifically, for every *UnitOp* pair, we extract 8 features from their package names, class names, method/field names and comments, as described below.

Common Package Prefix Length: For example, the common package prefix length of "android.app" and "android.support.v4.app" is 1.

Package Hierarchy Distance: For example, it takes 4 steps to reach "android.support.v4.app" from "android.app" in the package hierarchy, so the package hierarchy distance is 4.

Package Name Similarity: We split package names into separated words by the dot symbol, and use the Jaccard similarity to calculate the package name similarity. For example, we can split "android.app" and "android.support.v4.app" into two word sets "android, app" and "android, support, v4, app". With Jaccard similarity, we can calclate the package name similairty is 0.5.

Class Name Similarity: Because Java uses camel nomenclature, we also split class names into word sets, and use the word set similarity to calculate their similarity.

Is Both Static: We test whether the method invoked or the field accessed in the two *UnitOp*s is both static or non-static.

Operation Action Similarity: The operation action is the action (a verb) performed by a *UnitOp*. For method-invocation operations, we can extract the verbs from their method name. To tag Part-Of-Speech for different words, we use Stanford Log-linear Part-Of-Speech Tagger [40]. For example, "get" is the operation type for "getStorageState()". For field-access operations, we use "get" or "set" to represent their actions.

Operation Target Similarity: The target of an operation can be represented by the nouns in the method name for a method-invocation operation and the nouns in the field name for a field-access operation. We also use [40] to tag the Part-Of-Speech for different words. We use word set similarity to calculate the similarity between the two noun sets.

Comment Similarity: Android provides sufficient comments to describe each method and field. Note that we do not require source code of apps. For each *UnitOp* pair, the methods invoked or the fields accessed all belong to Android system classes, which have comments. These comments also help to gauge the distance of two methods or fields. By parsing the comments and extracting the nouns, we also use word set similarity of two noun sets to calculate comment similarity.

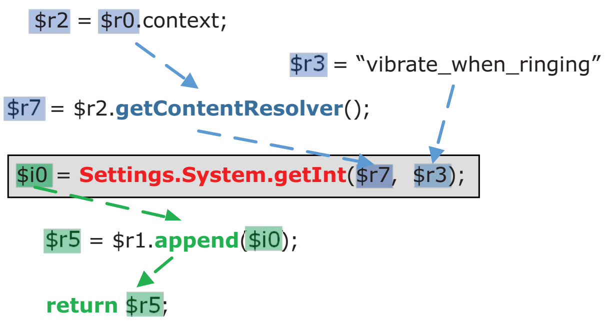
3.3.2 Features Extraction for Input/Output Dependencies. The behavior semantic features mainly consider the characteristics of *UnitOp* pairs, without taking their dependencies with other statements into account. Therefore, we also extract input/output dependency features for each *UnitOp* as a complement.

Specifically, during the static analysis phase, we perform intra-branch data flow analysis to track data flow to the arguments of an *UnitOp* and the propagation of its return value. Figure 6 gives an example to demonstrate this kind of data flow. In this example, *$i0 = Settings.System.getInt($r7, $r3)* is the target *UnitOp*. By tracking the data flow to its arguments, we find that $r3 and $r0 are two input-related variables ($7 is transitively dependent on $r0), and the *getContentResolver()* API is an input-related operation (because $r7 is acquired by this API). Besides, we track the usage of return value $i0, and find that $r5 is an output-related variable (because it depends on $i0) and the *append()* API is an output-related operation (because it takes $i0 as argument). Based on input/output dependencies, we extract the following 11 features.

Argument Number: We extract the number of arguments that the subject API takes.

Arguments Type Similarity: We extract all argument types for a method-invocation operation, and the declared type of a field for a field-set operation. Based on the texts of these types, we split them into words and calculate word set similarity between them. For field-get operations, they have no argument, so the word set for them is empty.

Constant Similarity: Sometimes constants are used as arguments for method-invocation operations or assigned to fields



```
$r2 = $r0.context;
                                    $r3 = "vibrate_when_ringing"
$r7 = $r2.getContentResolver();

$i0 = Settings.System.getInt($r7, $r3);

  $r5 = $r1.append($i0);

  return $r5;
```

Figure 6: Example to demonstrate feature extraction for input/output dependencies for Settings.System.getInt().

for field-set operations. Typically, these constants are static final variables which are declared in Android system classes. The names of the constant variables contain a lot of semantic information useful for similarity testing. For example, for a method-invocation operation *AlarmManager.set(0, ..., ...)*, if we search the constant 0 in the AlarmManager class we can find its name is *RTC_WAKEUP*. By recovering the constant names, we can get more semantic information about the operation. To recover the names for constants, we build a name database from Android API documents which records all possible names for a given constant value. By querying the database we may get several names for a constant value, and we will use the one declared in the same class with the method/field of the *UnitOp*. After splitting the constant names into words, we use word set similarity between two word sets to represent constant similarity.

Has Return Value: We test whether the compatibility-related API has return value.

Return Type Similarity: We extract the return type for a method-invocation operation, the declared type of the field for a field-get operation. Based on the texts of these types, we split them into words and calculate word set similarity. A field-set operation does not have return value. Hence its word set is empty.

Input-/Output-related Variable Set Size: We extract the number of input-/output-related variables that depend on the arguments/return value of the subject API.

Input-/Output-related Variable Set Similarity: We use Jaccard similarity to calculate the similarity between two input-/output-related variable sets. For example, the similarity for "($r0, $r3)" and "($r3)" is 0.5.

Input-/Output-related Operation Set Similarity: We collect two input-/output-related operation sets for each *UnitOp* pair, and split the method names into words. We use the word set similarity to calculate the similarity between the two word sets.

4 RESULTS

The static analysis module of RAPID is implemented on FlowDroid [1]. It leverages the ICFG (Inter-procedure Control Flow Graph) of FlowDroid to identify compatibility checks. To support feature extraction for input/output dependencies, it implements an intra-branch data flow analysis for the compatibility check related branches. Overall, the static analysis module contains 3,817 LOC Java code, and the feature extraction module has 2,649

Table 1: The App Dataset for the Large-scale Study.

Market	Date	Count
Google Play	2018.03-2018.10	93,681
Huawei	2019.02	25,638
Qihoo	2019.02	22,537
Tencent	2019.02	58,504
Baidu	2019.02	123,388
All (unique)	–	309,967

LOC Java code. The feature extraction module utilizes the pre-trained GoogleNews word2vec model [33] and Deeplearning4j framework [3] to calculate word similarity. Our classifier is built with the SMO classifier [36] in Weka [50], a kind of support vector machine (SVM), with a poly kernel (exponent as 2).

Dataset. To perform a large-scale study, we crawled apps from Google Play and third-party app markets, including their top-ranked apps. As shown in Table 1, we collect 309,967 unique apps. We run `RAPID` in parallel to analyze these apps and set the timeout for each app to 5 minutes. In all, `RAPID` successfully analyzed 296,133 apps with 12,470 apps timeout. Besides, there are 1,364 apps which failed to parse by FlowDroid. Based on the results of 296,133 apps, our study is conducted from the following perspectives:

- *Effectiveness & Efficiency of* `RAPID` (see §4.1);
- *Landscape of Compatibility Issue Handling* (see §4.2);
- *The Help of Google's Recommendations* (see §4.3);
- *SigChk Cases that Can Be Improved* (see §4.4).

4.1 Effectiveness & Efficiency of RAPID

To construct ground truth for the classifier, we manually analyze 293 compatibility checks, covering 123 distinct APIs. For each compatibility check, we first reverse the bytecode and label whether it is SigChk or RplChk following the definition in §2.2. For RplChk, we further locate the alternative *UnitOp* to the incompatible API and label it as a positive case in our ground truth. For other *UnitOp*s in the candidate-branch, we label them as negative cases. Here each case consists of two *UnitOp*s: one being the incompatible API invocation from the api-branch and the other a *UnitOp* from the candidate-branch. In all, our ground truth consists of 1,053 *UnitOp* pairs (including 191 positives and 862 negatives). As we will show later, such a ground truth set is sufficient for achieving high classification accuracy. In all, 4 student authors participate the ground truth labeling and every participant has at least 2 years of Android app reverse engineering experience. Besides, at least two participants are involved in every *UnitOp* pair.

Q1: How effective is **`RAPID`** ***in differentiating SigChk and RplChk?*** First, we use ten-fold cross validation to evaluate the performance of the trained SVM classifier. As Table 2 shows, our model achieves high accuracy during the training phase (F1-score: 95.21%). Second, we randomly select 414 compatibility checks from the whole dataset as the validation set to further validate the performance of the classifier for the large-scale study. Note that there is no overlap between the training set (293 checks) and the validation set (414 checks), and the validation set is constructed randomly before the classifier is trained. For the selected 414 checks, `RAPID` reports 210 RplChk cases and 204 SigChk cases. We manually label these checks and every check is verified by at least two student authors. The results show that, our trained model still achieves quite good performance (F1-score: 91.96%) in this set.

Table 2: The Effectiveness of `RAPID`.

Dataset	Precision	Recall	Accuracy	F1-score
Training Set	96.76%	93.72%	98.29%	95.21%
Validation Set	93.41%	90.58%	97.43%	91.96%

Q2: How do the selected features help **`RAPID`*****?*** We also split the whole feature set into two parts: feature set from behavior semantics and from input/output dependencies, and train two respective models. Table 3 shows the performance of the two models. Observe that the trained model with the feature set from either behavior semantics or input/output dependencies alone exhibits inferior performance. It shows the necessity for `RAPID` to consider the two feature sets together.

Table 3: The Effectiveness of `RAPID` with Different Feature Sets (in training set).

	RAPID	Behavior Semantic	Input/Output Dependency
F1-score	95.21%	86.81%	76.54%

Q3: How efficient is **`RAPID`*****?*** We collect the time cost of `RAPID` in performing the analysis and classification on the whole dataset. Specifically, in the static analysis phase, the average time for `RAPID` to analyze an app is 56.23 seconds. In the feature extraction and RplChk predication phase, the average time for processing an app is 3.74 seconds, depending on how many compatibility checks are detected in the app.

Summary: `RAPID` is both effective and efficient in classifying SigChk and RplChk, by achieving a F1-score of 95.21% and 91.96% in the training set and randomly-selected validation set respectively. Besides, the two feature sets both contribute significantly to its overall performance.

4.2 Landscape of Compatibility Issue Handling

Q4: How prevalent is evolution-induced API compatibility issues in real world? By checking the analysis results of the 296,133 apps that have been successfully analyzed (as presented in Table 4), we find 49,137 apps do not invoke any API that is incompatible on the API levels they declared to have compatibility with, thus they are unaffected by evolution-induced API compatibility issues. For the remaining 246,996 apps (i.e. 83.41% of the apps), at least one incompatible API is used by each app. This result shows that compatibility issues are indeed prevalent in Android apps.

Q5: How many checks belong to SigChk and RplChk respectively? As Table 5 shows, among the affected 246,996 apps, developers perform 1,540,480 checks against compatibility-related APIs. For these checks, `RAPID` reports that 592,089 checks provide

Table 4: Apps with Compatibility Issues in the Dataset.

All Apps	Unaffected Apps	Affected Apps
296,133	49,137	246,996

replacement implementations for the compatibility-related APIs on incompatible API levels, while the remaining compatibility checks just prevent the app from crashing. We can find that the ratio of RplChk (38.4%) is quite low in the real-world, which calls for improvement. Besides, by examining the control structure of these checks, 1,008,445 checks are found to have both true and false branches. This result clearly shows that simply utilizing the control flow structure to differ RplChk from SigChk is not reliable, rendering the necessity to design a tool (such as RAPID) for RplChk classification.

Table 5: SigChk and RplChk Distribution in the Dataset.

All Checks	SigChk / RplChk Cases	If-then / If-then-else Checks
1,540,480	948,391 / 592,089	532,035 / 1,008,445

Summary: Evolution-induced API compatibility is a prominent challenge, while developers do not tend to deal with incompatible API invocations by providing replacement functionalities. Besides, SigChk and RplChk can not be classified by simply considering the control structure of the compatibility check.

4.3 The Help of Google's Recommendations

In our study, there are 2,850 compatibility-related APIs that have been checked by developers, constituting 1,540,480 compatibility checks. Since the overall ratio of RplChk (38.4%) is quite low, we want to dive further into the results to find the factors that affect developers' choice on handling compatibility issues. Specifically, we focus on the following research questions.

Q6: How many recommendations does Google give to developers to deal with API-related compatibility issues? Considering Google sometimes provides recommendations for compatibility-related APIs in the API document [4], we first collect all the recommendations that Google provides to developers. Since Google's recommendations are given as comments of the compatibility-related APIs without a specialized format, we choose to manually compile such recommendations. By carefully reading the documents for these 2,850 APIs, we label the recommendations that are given by Google as follows. Each API is labeled by at least two analysts, and if they have different opinions, another analyst will participate. In all, 5 analysts took part in labeling, and it costs about 25 people hours. Based on our investigation, Google only gives replacement recommendations for 130 APIs among the total 2,850 APIs and all the 130 APIs belong to deprecated APIs. For the remaining 2,720 APIs, 259 APIs are newly-introduced APIs, and 2,461 APIs belong to deprecated APIs.

Q7: Do Google's recommendations affect developers' decision on SigChk and RplChk? For APIs that have recommendations and those that do not have recommendations, we calculate the ratio of RplChk for them separately. Table 6 presents the results. From this table, we find that 80.40% of the compatibility checks for APIs with Google's recommendations are RplChk, while the RplChk ratio for APIs without Google's recommendations is only 31.02%. The significant gap in the RplChk ratio between the two API sets (*p-value* = 4.313×10^{-9}) indicates that whether Google giving replacement API recommendations significantly impacts developers' decision on dealing with the compatibility issue using SigChk or RplChk. We can also infer that providing developers with replacement recommendations is an effective way to help them handle compatibility issues.

Table 6: The ratio of RplChk for compatibility checks of the APIs that (do not) have recommendations.

	RplChk Cases	SigChk Cases	RplChk Ratio
With Recommendations	186,005	45,332	80.40%
No Recommendations	406,084	903,059	31.02%

Q8: Do developers follow Google's recommendations when they replace incompatible API invocations? Since we have manually extracted Google's recommendations for 130 APIs, we want to further check whether developers follow Google's recommendations to provide replacement functionalities for these APIs. First, we collect all the RplChk cases for the 130 APIs (i.e. 186,005 RplChk cases). Second, we search the compatibility check branches for these cases. If a recommended API appears in a candidate-branch, we considered this case is handled by following Google's recommendation. Table 7 presents the ratio of the RplChk cases that do not follow Google's recommendations. The results show that although Google gives recommendations, developers handle a large part (i.e. 19.31%) of compatibility issues in their own way.

Table 7: The ratio of RplChk cases that do not follow Google's recommendations.

RplChk Cases that have Google's Recommendations	RplChk Cases that do not follow Recommendations	Not Followed Ratio
186,005	35,909	19.31%

Q9: Why do developers not follow Google's recommendations? For those incompatible API invocations that are not replaced in the recommended way by Google, we further investigate the underlying causes. First, we count the RplChk cases that do not adopt Google's recommended API. Second, we select the top 5 APIs and manually analyze the RplChk cases for these APIs to infer why developers do not want to follow recommendations. Overall, we observe two main reasons that cause developers to handle compatibility issues in a different way (from the recommended way). The analysis result is presented in Table 8.

- **Reason-A: The recommended API needs extra parameters.** The replacement APIs that are recommended by Google sometimes need more arguments than the original

Table 8: Top 5 APIs that are not replaced in the recommended way.

API	Google Recommended API	Reason	Actual Replacement
Resources.getColor(I)I	Resources.getColor(I,Theme)I	A	Context.getColor(I)I
Resources.getDrawable(I)Drawable	Resources.getDrawable(I,Theme)Drawable	A	Context.getDrawable(I)Drawable
Resources.getColorStateList(I)ColorStateList	Resources.getColorStateList(I,Theme)ColorStateList	A	Context.getColorStateList(I)ColorStateList
Display.getWidth()I	Display.getSize(Point)V	B	DisplayMetrics.widthPixels
Display.getHeight()I	Display.getSize(Point)V	B	DisplayMetrics.heightPixels

compatibility-related API. For example, as depicted in Table 8, Google recommends using *Resources.getDrawable(I,Theme)* when *Resources.getDrawable(I)* is not available. However, to invoke *Resources.getDrawable(I,Theme)*, developers need to prepare a Theme instance as the second argument which is not so easy. As a result, we observe that in real-world apps developers tend to use another API *Context.getDrawable(I)* to fix the compatibility issue, because it can be invoked with the same argument as the incompatible API *Resources.getDrawable(I)*.

- **Reason-B: A quick replacement exists.** Some of the compatibility-related APIs are getter functions of some fields. For example, *Display.getWidth()* returns the current width of display. Although Google recommends using *Display.getSize(Point)* instead when this API is deprecated (since API level 15), developers are more willing to deal with this issue by directly accessing the field *DisplayMetrics.widthPixels* to get the current width of display. It shows that a quick and easy replacement is more preferable.

Summary: On one hand, the RplChk ratio for compatibility-related APIs that Google give recommendations is significantly higher than those without recommendations, but Google only gives recommendations for 130 APIs out of 2,850 compatibility-related APIs seen in our study. On the other hand, though Google gives recommendations, developers do not always follow, because an easier way to fix the compatibility issue than the recommended one exists. Therefore, providing easily-adoptable recommendations seems a promising way to help developers fix compatibility issues.

4.4 SigChk Cases that Can Be Improved

For those SigChk cases, we want to measure how many of them actually have replacement implementations while developers do not actively explore. First of all, we need to know which compatibility-related APIs can be replaced, since we observe that some APIs can not be alternated if the platform does not support the functionalities. For example, *WifiManager.startScan()* [7] is deprecated since API level 28 and the platform does not provide similar interfaces for apps to use. Thus, no app in our dataset is found to provide a replacement implementation for this API.

Q10: How many compatibility-related APIs have replaced implementation? By checking all the RplChk cases, we can find out all the compatibility-related APIs that have alternative implementations. Since some apps replace these APIs, it is appropriate to expect that other apps are able to replace these APIs too. In our dataset, we find that developers successfully provide alternative implementations for 1,086 compatibility-related APIs. Among the 1,086 APIs, 201 APIs belong to deprecated APIs and 885 APIs belong to newly-introduced APIs. Considering that Google only gives recommendations for 130 deprecated APIs, we find that developers can also work out their own solutions to provide replacement functionalities for newly-introduced APIs. It is worth noting that this knowledge acquired from experienced developers is quite valuable to facilitate compatibility issue handling.

Q11: How many SigChk cases can actually be handled by RplChk? Based on the compatibility-related APIs that can be replaced, we count all the SigChk cases on these APIs. As Table 9 shows, in our study, 690,736 checks can be handled with RplChk while developers do not do. If all these issued are checked with replacements, the ratio of RplChk in our dataset can significantly increase from 38.4% to 83.3%. This gap shows that there is a great room to improve the current practice of handling compatibility issues. However, generating a legitimate alternative implementation automatically entails addressing new challenges (e.g., parameter preparation), which are beyond the scope of the paper.

Table 9: SigChk Cases that Can be Handled by RplChk.

All Checks	RplChk Cases	SigChk Cases	SigChk Cases that Can be Handled by RplChk
1,540,480	592,089	948,391	690,736

Summary: Developers find more ways to replace incompatible APIs than Google's recommended ones, while not all developers know these ways. It is quite meaningful to mine such knowledge and share it with the whole community, especially with novice developers.

5 LIMITATIONS

Limitations of RAPID. This paper aims to conduct a large-scale study on real-world Android apps to shed light on the current practice of handling compatibility issues, without trying to extract common fix patterns from them. With the aim to facilitate a measurement study, RAPID is not designed to be a repair tool. It is quite different from the heavy-weight static/dynamic analysis techniques used in program repair [13, 41, 43] and patch generation [29, 31, 32, 47], which need to reason about repair correctness. The main contribution of RAPID is that it features a deeper analysis than existing works by classifying RplChk and

SigChk. RAPID extracts 19 features for classifier training. Actually, more features can be considered, such as features extracted from API implementations. Nevertheless, our trained model achieves good accuracy on both training and random-selected data sets. Our static analyzer is built upon existing tools [1, 22, 44], thus it inherits limitations of these tools. Besides, developers may handle API compatibility issues by checking other conditions instead of *SDK_INT*. For example, Java reflection may be used to find out if an API is supported by the underlying platform. Currently, RAPID cannot handle this kind of cases. However, *SDK_INT* is the most convenient way for developers to check compatibility issue and the most prevalent way as reported in [16]. To handle the compatibility checks against Java reflection calls, RAPID can adopt existing techniques [20, 25, 39].

Limitations of Study. The inherent limitations of the underlying analysis infrastructures may post external threats to the validity of the reported results. Several experiments in our study depend on Google's fix recommendations. Since Google only provides the fix recommendations in API documents, we have to manually collect these recommendations. To reduce the threat of manual mistakes or biases, we follow the widely-adopted cross-validation method to ensure the correctness of our results. Besides, to guarantee the representativeness of our study, we have collected about 300k apps to conduct the study. To the best of our knowledge, this is the largest study in the scope of Android app compatibility issues. The ground truth data set for training the classifier contains 293 compatibility checks with 123 distinct APIs. Note that substantial manual efforts are involved for each check as we have to label each pair-wise combination of the incompatible API and each statement in the other branch. Although our results show that the trained classifier achieves good accuracy, it is possible that the samples in our ground truth data set do not provide comprehensive coverage of the features for classification. To mitigate the threat, we further validate the results on a set of randomly-selected samples.

6 RELATED WORK

Compatibility Issues in Android Apps. Wei et al. [46] perform the first systematical study on the compatibility issues of Android apps by manually analyzing 191 real-world issues in open source apps. Their study categorizes compatibility issues into device-specific and non-device-specific. According to this classification method, evolution-induced compatibility issue belongs to non-device-specific issues. Since Android API evolution is well documented, it is quite easy for systematical modeling [24]. Based on Android API lifecycle, CiD [24] and IctApiFinder [16] leverage program analysis techniques to detect missing checks of evolution-induced compatibility issues. Huang et al. [17] discover another kind of non-device-specific compatibility issue, called callback compatibility issue, which is caused by the callback protocol evolution. Wei et al. [26] recently present PIVOT which can help to find device-specific compatibility issues by ranking API-device correlations. ACRyL [37] detects compatibility issues by learning the app changes in response to API changes. To understand the observed incompatibilities in Android apps, Haipeng et al. [2] conduct a study on installation-time and run-time app incompatibilities. Ziyi et al. [53] give a study on the intentions of developers on app compatibility. Existing works also measure the consequences of compatibility issues: affecting application performance [28], disappointing users [15, 21], and even leading to security risks [54]. In comparison, our work is the first to distinguish two kinds of compatibility checks: SigChk and RplChk, and to perform a study of a much larger scale on real-world practice in handling evolution-induced API compatibility issues.

Program Repair/Fix. Monperrus [34] presents a systematic survey on automatic software repair. To guide automated program repair, Liu et al. [27] analyze bug fix commits in open source Java projects to extract repair patterns. [30] presents some manual analysis results on bugs in Defects4J to shed light on how to automatically repair these bugs. [51] further proposes a way to automatically repair Defects4J bugs based on API documents. More advanced program analysis techniques are also incorporated into this line of research. Koyuncu et al. [19] automatically mine repair code pattern by differing the AST trees between fixed and unfixed code, and utilize this pattern to guide program repair. Xuan et al. [52] propose a way to repair buggy conditional statements in Java programs. [42] seeks to establish common root causes for Android apps crashes, and propose generic transformation operators to facilitate the remedy of these crashes, such as replacement code and null pointer check. Compared to these works, this paper performs a large-scale study on the developers' practice in fixing compatibility issues and the findings gained through the study could help to investigate automatic repair techniques for these issues.

7 CONCLUSION

Previous works on evolution-induced API compatibility issues mainly focus on detecting missing checks for incompatible APIs. However, performing compatibility checks does not necessarily mean the compatibility issues are well-handled. This paper conducts the first large-scale study on the practice of handling evolution-induced compatibility issues with about 300,000 Android apps. To perform such a large-scale study, this paper presents RAPID which adopts a learning-based approach to determining whether developers simply check an incompatible API invocation or provide a replacement implementation on those incompatible API levels. Through the study, many interesting findings are derived to help us gain an in-depth understanding about the current status of compatibility issue handling, and indicate several directions to help developers to handle these issues.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099, 61602123, 61602121), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). The authors from Purdue University were supported in part by NSF 1748764, 1901242 and 1910300. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Edinburgh, United Kingdom). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[2] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-scale Study of Application Incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 216–227. https://doi.org/10.1145/3293882.3330564

[3] Deeplearning4j. 2019. Deeplearning4j: Open-source, distributed deep learning for the JVM. http://deeplearning4j.org/.

[4] Android Documentation. 2019. Android API Document. https://developer.android.com/reference/packages.

[5] Android Documentation. 2019. Android API Document for AccountManager. https://developer.android.com/reference/android/accounts/AccountManager.html.

[6] Android Documentation. 2019. Android API Document for android.content.res.Resources.getColor(int). https://developer.android.com/reference/android/content/res/Resources#getColor(int).

[7] Android Documentation. 2019. Android API Document for WifiManager.startScan(). https://developer.android.com/reference/android/net/wifi/WifiManager.html#startScan().

[8] Android Documentation. 2019. Android API Level. https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels.

[9] Android Documentation. 2019. Android Manifest File. https://developer.android.com/guide/topics/manifest/manifest-intro.

[10] Android Documentation. 2019. Android SDK Platform release notes. https://developer.android.com/studio/releases/platforms.

[11] Android Documentation. 2019. api-versions.xml in Android SDK. https://android.googlesource.com/platform/development/+/refs/heads/master/sdk/api-versions.xml.

[12] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Security and Privacy in Communication Networks: SecureComm 2018 International Workshops*. Springer.

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[14] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A Large-scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-malware Products. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 421–431. https://doi.org/10.1145/3180155.3180228

[15] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *19th Working Conference on Reverse Engineering (WCRE)*. IEEE, 83–92.

[16] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)* (Montpellier, France). ACM, New York, NY, USA, 167–177. https://doi.org/10.1145/3238147.3238185

[17] 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 (ASE)* (Montpellier, France). ACM, New York, NY, USA, 532–542. https://doi.org/10.1145/3238147.3238181

[18] M. E. Joorabchi, A. Mesbah, and P. Kruchten. 2013. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/ESEM.2013.9

[19] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[20] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for static analysis of Java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 507–518.

[21] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Felix Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. 2015. Characterizing smartphone usage patterns from millions of android users. In *Proceedings of the 2015 Internet Measurement Conference (IMC)*. ACM, 459–472.

[22] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 280–291.

[23] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In *The International Symposium on Software Testing and Analysis (ISSTA)*.

[24] 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 (ISSTA)*. ACM, 153–163. https://doi.org/10.1145/3213846.3213857

[25] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *International Static Analysis Symposium (SAS)*. Springer.

[26] Wei Lili, Liu Yepang, and Cheung Shing-Chi. 2019. PIVOT: Learning API-Device Correlations to Facilitate Android Compatibility Issue Detection. In *Proceedings of 41st ACM/IEEE International Conference on Software Engineering (ICSE)* (Montréal, QC, Canada).

[27] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 275–286.

[28] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 1013–1024.

[29] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[30] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[31] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Transactions on Software Engineering and Methodology* 27 (10 2018), 1–37. https://doi.org/10.1145/3241980

[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[33] mmihaltz. 2019. Word2vec Google News model. https://github.com/mmihaltz/word2vec-GoogleNews-vectors/.

[34] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 17.

[35] J. Park, Y. B. Park, and H. K. Ham. 2013. Fragmentation Problem in Android. In *2013 International Conference on Information Science and Applications (ICISA)*. 1–2. https://doi.org/10.1109/ICISA.2013.6579465

[36] John Platt. 1998. Sequential minimal optimization: A fast algorithm for training support vector machines. (1998).

[37] 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 *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, Piscataway, NJ, USA, 288–298. https://doi.org/10.1109/MSR.2019.00055

[38] Sentence-similarity. 2019. Comparing Sentence Similarity Methods. http://nlp.town/blog/sentence-similarity/.

[39] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer, 485–503.

[40] Stanford. 2019. Stanford Log-linear Part-Of-Speech Tagger. https://nlp.stanford.edu/software/tagger.html.

[41] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 187–198. https://doi.org/10.1145/3180155.3180243

[42] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 187–198.

[43] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 471–482. http://dl.acm.org/citation.cfm?id=2818754.2818813

[44] 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*. IBM Corp., 214–224.

[45] Yan Wang and Atanas Rountev. 2017. Who changed you?: obfuscator identification for Android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 154–164.

[46] 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 (ASE)* (Singapore, Singapore). ACM, New York, NY, USA, 226–237. https://doi.org/10.1145/2970276.2970312

[47] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[48] Wikipedia. 2019. Jaccard Index. https://en.wikipedia.org/wiki/Jaccard_index.

[49] Wikipedia. 2019. Word2vec. https://en.wikipedia.org/wiki/Word2vec.

[50] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[51] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[52] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering (TSE)* 43, 1 (2017), 34–55.

[53] Ziyi Zhang and Haipeng Cai. 2019. A Look into Developer Intentions for App Compatibility in Android. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems* (Montreal, Quebec, Canada) *(MOBILESoft '19)*. IEEE Press, Piscataway, NJ, USA, 40–44. http://dl.acm.org/citation.cfm?id=3340730.3340741

[54] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 409–423.