View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UNL | Libraries

University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

4-2019

GAINDroid: General Automated Incompatibility Notifier for Android Applications

Bruno Vieira Resende e Silva
University of Nebraska-Lincoln, bvrsilva1@gmail.com

Follow this and additional works at: https://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Vieira Resende e Silva, Bruno, "GAINDroid: General Automated Incompatibility Notifier for Android Applications" (2019). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 167.
https://digitalcommons.unl.edu/computerscidiss/167

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

GAINDroid: GENERAL AUTOMATED INCOMPATIBILITY NOTIFIER FOR ANDROID APPLICATIONS

by

Bruno Vieira Resende e Silva

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Hamid Bagheri

Lincoln, Nebraska

April, 2019

GAINDroid: GENERAL AUTOMATED INCOMPATIBILITY NOTIFIER FOR ANDROID APPLICATIONS

Bruno Vieira Resende e Silva, M.S.

University of Nebraska, 2019

Adviser: Hamid Bagheri

With the ever-increasing popularity of mobile devices over the last decade, mobile apps and the frameworks upon which they are built frequently change. This rapid evolution leads to a confusing jumble of devices and applications utilizing differing features even within the same framework. For Android apps and devices, representing over 80% of the market share, mismatches between the version of the Android operating system installed on a device and the version of the app installed, can lead to several run-time crashes, providing a poor user experience.

This thesis presents GAINDroid, an analysis approach, backed with a class-loader based program analyzer, that automatically detects three types of mismatches to which an app may be vulnerable across versions of the Android API it supports. Unlike all prior techniques that focus on identifying a particular problem, such as callback APIs issues, GAINDroid has the potential to greatly increase the scope of the analysis by automatically and effectively analyzing various sources of incompatibilities that may lead an app to crash at run-time. We applied GAINDroid to 3,590 real-world apps and compared the results of our analysis against state-of-the-art tools. The experimental results demonstrate its ability to outperform the existing analysis techniques in terms of both the number and type of mismatches correctly identified as well as run-time performance of the analysis.

Table of Contents

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

2 Background **5**

2.1 Android API Background 5

2.2 API Compatibility Issues 6

2.2.1 API invocation mismatch 7

2.2.2 API callback mismatch 8

2.3 Permission-induced Compatibility Issues 9

3 Approach **12**

3.1 API Usage Extraction 13

3.2 Database Construction 14

3.3 Permissions Mapping 15

3.4 Mismatch Detection 15

4 Empirical Evaluation **22**

4.1 Objects of Analysis 23

4.2 Variables and Measures 25

4.2.1 Independent Variables 25
4.2.2 Dependent Variables 26
4.2.3 Study Operation 26
4.3 Threats to Validity 27

5 Results **28**
5.1 RQ1: Accuracy 30
5.2 RQ2: Real-World Applicability 34
5.3 RQ3: Performance 36

6 Discussion **41**

7 Related Work **43**

8 Conclusion and Future Work **48**

Bibliography **51**

List of Figures

2.1 Mismatch between app and device API level 7

3.1 Architectural Overview of GAINDroid 13

5.1 Scatter plot representing analysis time for compatibility checking of Android apps using GAINDroid 38

5.2 Number of classes loaded by GAINDroid and CiD when analyzing real-world Android apps. 39

5.3 Amount of memory used by GAINDroid and CiD when analyzing real-world Android apps. 40

List of Tables

2.1 Types of mismatch 5

4.1 Basic Characteristics of Objects of Analysis 24

5.1 Effectiveness and Efficiency of GAINDroid 29

5.2 Time comparison between GAINDroid 37

7.1 Comparing GAINDroid to the state-of-the-art of compatibility detection techniques. 45

Chapter 1

Introduction

Android is the leading mobile operating system representing over 80% of the market share [4]. The rapid rise of Android is largely due to its vibrant app market [12], which currently contains nearly three million apps, with thousands, are added and updated on a daily basis. Android apps are developed using an application development framework (ADF) that ensures apps devised by a wide variety of suppliers can interoperate and coexist in the same environment. An ADF exposes well-defined application programming interfaces (APIs) that encapsulate a series of commands that directly access resources from the Android operating system and hardware. In this fashion, developers can devote their effort only to develop apps using the provided APIs, instead of deeply understanding the operating system or hardware.

The Android ADF evolves frequently, with hundreds of releases from multiple device vendors since the birth of Android in 2008 [27]. Such a rapid evolution witnessed in the Android ADF can lead to various incompatibilities in the Android apps that target a specific version of the Android operating system but run on older or newer versions of it. As a result, defects and vulnerabilities, especially after ADF updates, have continued to plague the dependability and security of the Android devices and their apps [54, 58]. A recent study shows that 23% of

Android apps behave differently after a framework update, and around 50% of the Android updates have caused previously working apps to fail or rendered systems unstable [45]. This type of dependability concern has been referred to as "death on update" [33, 39, 36, 64, 47, 69].

In addition, the launch of Android ADF version 6 (API level 23) introduced a dynamic permission system, which may lead to a new class of incompatibility issues. In the prior versions, the permission system was entirely static, where the user should grant permissions requested by an app during the app's installation, otherwise the installation would be canceled. The new permission system instead allows users to give permissions dynamically, allowing only the access to resources he/she thinks is safe, at run-time [19]. There are two common ways that compatibility issues can arise; (i) when an app targets an API level of 23 or higher and it does not appropriately implement mechanisms to support run-time dangerous permissions; and (ii) when any app using dangerous permissions is targeting API-level of 22 or below and is installed on devices with the API-level of 23 or higher.

Recent efforts in the literature have studied compatibility issues [46, 67, 68]. However, existing compatibility detection techniques target only certain types of APIs. For example, work by Huang et al. [46] only targets callback APIs related lifecycles; generically identifying them requires significant manual labor [46] as well as thorough inspection of incomplete documentations [68]. Approaches presented in [51, 44] also target misuse of the Android API. However, they specifically target mismatched regarding API invocation. In other words, when an API method is invoked within the code but the current Android platform does not support the referred method. Problems related to the permission mechanism have also been targeted [61, 60]. However, those techniques are specific for the interaction

between wearable apps with the same mobile app, only verifying if both apps have shared the same set of permissions. Furthermore, none of the state-of-the-art techniques consider incompatibilities due to the dynamic permission system, introduced in Android level 23. Meanwhile, the state-of-the-art compatibility detection techniques suffer from acknowledged frequent 'false alarms', because of the coarse granularity at which they capture API information. With the lack of proper support for detecting compatibility issues, it may take a long time to isolate and repair such issues. It has been shown that a large portion of faults reported after an Android ADF release was not addressed for more than six months [65].

In this thesis, we present GAINDroid (**G**eneral, **A**utomated **I**ncompatibility **N**otifier for And**DROID**), that automatically detects mismatches regarding the use of Android APIs and the permission-induced compatibility issues. Unlike all prior techniques that focus on identifying a particular problem, such as callback APIs issues, our approach has the potential to greatly increase the scope of the analysis by automatically and effectively analyzing three sources of incompatibilities that may lead an app to crash at run-time.

We present results from experiments on 3,590 real-world apps (including benchmark apps used to validate state-of-the-art tools), validating GAINDroid's ability in (1) effective perform compatibility analysis of Android apps, that many of those reported issues cannot be even detected by the state-of-the-art analysis techniques; and (2) outperforming other compatibility detection tools also in terms of scalability. The rest of this thesis is organized as follows. Chapter 2 presents essential background information about run-time errors in Android. Chapter 3 describes our approach to identify potential mismatches. Chapter 4 describes our empirical evaluation. Chapter 5 answers the three research questions about our approach. Chapter 6 provides an additional discussion of our results. Chapter 7

describes related work. Finally, chapter 8 discusses future work and concludes this thesis.

Chapter 2

Background

In this section, we provide background information related to the Android API and motivating examples that demonstrate the need for building a technique that can effectively identify general API compatibility issues. We also describe three types of API and permission-related compatibility issues, summarized in Table 2.1, with illustrative examples.

Table 2.1: Types of mismatch

Mismatch type	Abbr	Compatibility	App level	Device level	Mismatch if
API invocation (App $\rightarrow$ Android)	API	Backward Forward	$\geq \alpha$ $< \alpha$	$< \alpha$ $\geq \alpha$	app invokes API method introduced/updated in α
API callback (Android $\rightarrow$ App)	APC	Backward Forward	$\geq \alpha$ $< \alpha$	$< \alpha$ $\geq \alpha$	app overrides API callback introduced/updated in α
Permissions	PRM	N/A Forward	≥ 23 < 23	≥ 23 ≥ 23	app misuses runtime permission checking

2.1 Android API Background

As of November 2018, there are 16 releases of the Android API and the most recent is Android Pie (supporting API level 28) [38]. Each version contains new and updated methods to improve performance and security, and to make new features and resources available for developers so they can enhance the user experience of

the app. In this work, we mainly refer to each release of the Android API by its API level (e.g., 26) rather than the associated name (Oreo) or Android version number (8.0) [6]. Developers use the provided APIs to access the core functionalities of the platform in order to create an application. Google strongly recommends that developers specify the range of the API levels that it can support in the manifest file or Gradle file by specifying a value for three specific attributes:

- `minSdkVersion`: The earliest level of the API supported by the app. If this attribute is not set, the app can be installed on a device running any API level.
- `targetSdkVersion`: The targeted level of the API, against which the app was developed.
- `maxSdkVersion`: The most recent supported level of the API. According to the Google documentation, declaring this attribute is not recommended [25] but installing an older app on a newer device may still lead to unexpected behavior [58].

2.2 API Compatibility Issues

As previously mentioned, incompatible API levels can cause run-time crashes in Android apps installed on a device running a different level of the API than that targeted by the app. Changes to the API are generally additive, so most such crashes stem from a lack of *b*ackward-compatibility, where an app is targeting a higher API level but, installed on a device running a lower one [25]. For example, in Figure 2.1, in the spectrum of device SDK version, the app would work flawlessly on devices with Android framework as the same astargetSdkVersion but, when

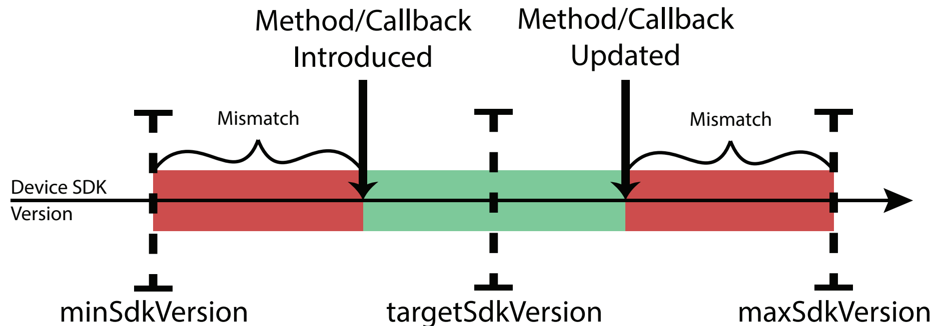

Figure 2.1: Mismatch between app and device API level

installed on devices with older versions of the system (in the first red area), apps would crash. However, despite Google's assurances, there may also be issues with *f*orward-compatibility when an app is installed on devices with a higher API level than the app's target (the second red area in Figure 2.1). If the app invokes a method or overrides a callback introduced in a newer level of the API than that supported by the device or removed in a newer level of the API than targeted by the app, there would be a mismatch, which could potentially crash the app.

We divide these API incompatibilities into two types, as shown in Table 2.1: *i*nvocation mismatches, where an app attempts to directly invoke an API method that is not supported by the device; and *c*allback mismatches, where an app implements a callback method missing from the API level installed on the device, which the Android system on the device will never invoke.

2.2.1 API invocation mismatch

Mismatches in an API method invocation are the simplest type of the API incompatibility. An app developed against a higher version of the API attempts to call a method introduced somewhere between its target version and that installed on the device, and the app crashes when the system cannot find the desired method.

```java
@Override
protected void onCreate(Bundle b){

  super.onCreate(b);
  setContentView(R.layout.activity_main);

  TextView text = findViewById(R.id.text);
  // if (Build.VERSION.SDK_INT >= 23) {
  text.setTextColor(resources.getColorStateList(
    R.color.colorAccent, context.getTheme()));
  // } else { ... }
}
```

Listing 2.1: API Invocation Mismatch

```java
public class CustomPreferenceFragment
  extends PreferenceFragment {

  @Override
  public void onAttach(Context context) {
    reinitializeData(context);
    super.onAttach(context);
  }
}
```

Listing 2.2: API Callback Mismatch

It also happens when an app developed against a lower version of the API may crash on a device running a higher version if a method has been removed.

An illustrative example is shown in Listing 2.1. The app targets Android API level 28, but its `minSdkVersion` is set to 21. Assuming that the app was installed on a device with the specified `minSdkVersion`, it will crash on the invocation of getColorStateList (lines 9-10), which was introduced in API level 23. One common way to protect against this mismatch is to check the device's API level at run-time, as shown in the comment on line 8. This prevents the app from executing the call on versions where it might be missing. However, developers could easily forget to add or modify the check when updating an app, leaving the app vulnerable to a mismatch.

2.2.2 API callback mismatch

The second type of mismatch works in the opposite direction, with calls initiating in the Android system invoking callback methods overridden in the app. Listing 2.2

```
1  @Override
2  protected void onCreate(Bundle b){
3
4    super.onCreate(b);
5    setContentView(R.layout.activity_main);
6
7    // if (Build.VERSION.SDK_INT >= 23) {
8    //   ActivityCompat.requestPermissions(...);
9    // } else {
10   Intent intent = new Intent(
11     MediaStore.ACTION_IMAGE_CAPTURE);
12   startActivity(intent);
13   // }
14 }
15
16 // @Override
17 // public void onRequestPermissionsResult(...)
18 //   { ... }
```

Listing 2.3: Permissions Mismatch for app targeting Android API $\geq$ 23

shows a snippet adapted from the *Simple Solitaire* [22] app, where the API callback onAttach(Context), introduced in API level 23, is overridden. However, the app is also specified to run on devices with API level lower than 23, which would not call that method. Thus, any critical actions (e.g., initialization of an object) performed by the app in that method would be omitted, possibly leading to run-time crashes.

2.3 Permission-induced Compatibility Issues

With the release of Android 6 (API level 23), the Android permission system is completely redesigned. If a device is running Android 5.1.1 (API level 22) or below, or the app's targetSdkVersion is 22 or lower, the system grants all permissions at installation time [19]. On the other hand, for devices running Android 6.0 (API level 23) or higher, or when the app's targetSdkVersion is 23 or higher, the app must ask the user to grant dangerous permissions at run-time. According to the Android documentation [5], the new run-time permission system encourages developers to help users understand why an application requires the requested dangerous permission.

Permission-induced incompatibility can also be divided into two general types of mismatch: permission request mismatches, where an app targeting API level 23 or higher does not implement the new run-time permission checking; and permission revocation mismatches, when an app targeting API 22 or earlier runs on a device with API 23 or later and the user revokes the use of a dangerous permission used by the app at run-time.

In the first scenario, an app is targeting an API level that is 23 or higher. However, it does not implement the new mechanism to handle requests for dangerous permissions at run-time, which can cause the app to crash. Listing 2.3 illustrates this scenario. The app may crash on line 12 where it attempts to use dangerous permission it did not request. To prevent the mismatch, the app would need to check the API version and request permissions at run-time (shown as comments on lines 7-9) and implement onRequestPermissionsResult (line 16). More detailed examples of the new run-time permissions system can be seen in the Android documentation [5].

The second scenario is when an app targets API levels 22 or below, but is installed on a device with Android API 23 or above–a $f$orward-compatibility issue. In this scenario, the user must accept all dangerous permissions requested by the app at install time, or the app will not be installed. However, in Android 6.0 (API level 23) or higher, the user has the ability to revoke those permissions at any time. If the user revokes any dangerous permission in the older app's setting after installation, the app would crash while trying to use that permission. This behavior has been recurrently reported in real-world apps. *AdAway* [1], for example, tries to access to external storage (such as an SD card) at run-time. If that permission is revoked, the app crashes when it tries to load data from the storage mechanism. In the next section, we outline our approach to detecting all three types of

mismatch–the two stemming from API invocations or callbacks and those arising from the new run-time permissions system.

Chapter 3

Approach

In this thesis, we present a method for identifying the four types of API and permission-based mismatches described in Table 2.1. Our approach is realized in GAINDroid (General Automated Incompatibility Notifier for Android Applications), a static analysis tool capable of detecting all three types of incompatibility.

As depicted in Figure 3.1, our approach comprises four main components, generating a list of mismatches from an app APK and a set of Android framework versions. The Android frameworks passed as input range from API level 2 through API level 28, collected using *sdkmanager*, provided in the Android SDK Tools to view, install, update, and uninstall packages for the Android SDK [21].

The three intermediate steps each produce a distinct artifact used by the fourth:

- *API Usage Extraction* uses static analysis techniques to identify calls to API methods, generating a usage report containing the call and data flow graphs for the app.
- *Database Construction* builds an API database that includes all the API methods of the relevant API levels.
- *Permission Mapping* outputs a map from each dangerous permissions to the API methods known to use that permission.

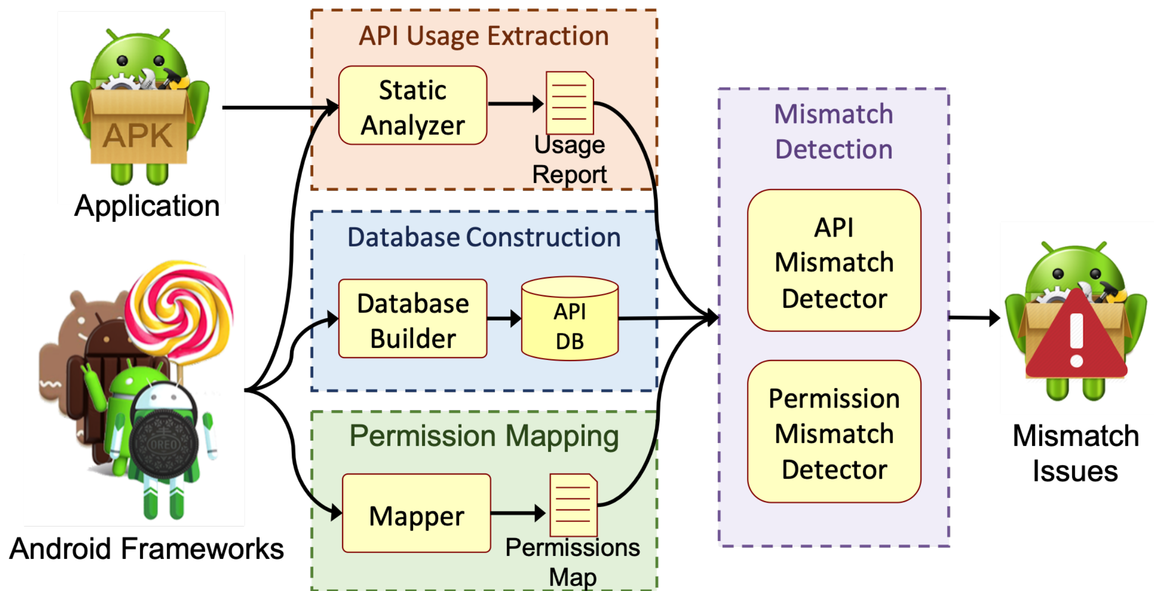

Figure 3.1: Architectural Overview of GAINDroid

- *Mismatch Detection* consumes each of those three artifacts and produces a list of all API call or permissions mismatches detected in the app.

3.1 API Usage Extraction

The API Usage Extraction module generates call graphs with embedded instruction graphs (for data flow analysis purpose) on each of its nodes a given decompiled APK file. This call and data-flow graph provide information about all Android API methods and callbacks invoked or overridden in the app, as well as any guard conditions that might enclose such calls. These graphs are going to be further used by the Mismatch Detector module.

In GAINDroid, this module is built upon Jitana [66], a high-performance hybrid analysis tool for Android. Even though Jitana works directly on Dalvik executable (dex) files such as those contained in an APK, we still need to perform APK decompilation to extract information such as needed permissions and

supported SDK versions from the decrypted manifest file. We modified Jitana to work based on .dex files from the Android level 23, a choice that was made based on the fact that this specific version was the first to introduce instructions to handle run-time permissions. We also had to modify the static analysis framework to perform inter-procedural analysis.

3.2 Database Construction

In Database Construction, we built an API database based on the Android APIs from levels 2 to 28, available via the Android command line tool *sdkmanager*. This tool enables us to download any available version of Android ADF. One of the downloaded files is *android.jar*, which contains all the Android APIs from the downloaded version. It can be opened and saved as a text document, which will be parsed following the *package.class_name api_name(parameters)return_type*. This output represents all the Android APIs for a given Android version, in a format that GAINDroid can read it, and then build the database. The database is based on Hash Map, where the Android level is the key and its APIs are the values. It is important to mention that the only intervention needed to update GAINDroid to analyze a newer version of API levels, is to download its new version via *sdkmanager*, parse it and paste the file on GAINDroid's SDK folder. The main purpose of building this module is to identify all the APIs that exist within each level, and a key step in our approach is to verify if a given API method is presented within the app's *minSdkVersion* and *targetSdkVersion* range. Note that while Google provides the list of APIs for each level, it does not include hidden APIs that may be called internally by the APIs within the framework. It is also important to have as many API levels involved because applications may

use different platforms as their *minSdkVersion* and *targetSdkVersion*. GAINDroid parses the provided Android framework versions and stores them in a format that can be statically analyzed by the *API Usage Extraction* component to generate the list of APIs in each level and a method call graph for each API method.

3.3 Permissions Mapping

The Permissions Mapping component produces a map of API methods to the dangerous permissions used by the Android framework during the execution of that API method. To achieve this, we used PScout [34], an Android permission mapping tool. We extended the latest official release of PScout to include new mappings that would reflect the more up to date Android API levels. Pscout only maps until Android level 22. Similar to the Android API database, permission maps are constructed once and reused in the subsequent analyses. It is important to mention that PScout mapping (regarding APIs) is already in the format that GAINDroid can decode (the same as in Database Construction). Some manual work will be necessary to update the mapping to newer Android versions. It will consist of verifying Android's official documentation, identifying new APIs which can access sensitive data, and updating the current mapping adding the new information, which should be in the same format used by PScout.

3.4 Mismatch Detection

The *Mismatch Detection* component analyzes the artifacts produced by *API Usage Extraction*, *Database Construction* and *Permission Mapping* to identify both API-related mismatches (*API Mismatch Detector*) and permissions-related mismatches (*Permissions Mismatch Detector*). The *Mismatch Detection* component first checks for

API incompatibility issues (described in Section 2.2) using the following process to detect both API invocation and callback mismatches:

Invocation mismatch: The detector uses Algorithm 1 to detect API invocation mismatches in each data flow graph in a node in the call graph generated by the *API Usage Extraction* module. If the current block represents an if statement with a guard condition (line 2), the range of supported API levels is filtered by extracting the minimum and maximum range from the guard and updating the minimum and maximum supported levels (line 3). If the current block is a call to an Android API method (line 4), we query the API database at each supported level to determine whether the method called in the current block is defined in the database (line 5-6). In case it is not defined, we add the current block to the set of mismatches (line 7). In the case when the app calls a method which does not belong to the Android API (line 8), we load and analyze this callee method (line 9) to check if there are Android API invocations that may lead to run-time issues. Finally, we reset the minimum and maximum supported API levels to those defined in the app's manifest at the end of each the conditional with a guard condition (lines 10-11). It is important to notice that *app.maxSdk* represents the most recent version of the Android API level in our database (currently API level 28).

GAINDroid can reliably detect Invocation mismatches because the *API Usage Extraction* component performs path-sensitive, context-aware, and inter-procedural data-flow analysis, which enables accounting for guard conditions on the supported versions across methods, missing in the other state-of-the-art techniques, such as Lint and CiD.

To better understand how our approach identifies API invocation mismatches, Listing 3.1 presents a code where GAINDroid performs path-sensitivity, context-

sensitivity, and inter-procedural analysis. It is crucial for our tool to identify when the app forks, especially when it is conditioned to guard condition, like in line 14, indicating that the code of block between line 14 and 16 will only be triggered under specific API levels. After decoding an instruction, we verify if it contains a method call, a variable declaration, a condition, etc. This functionality is given by the static analysis framework Jitana. Knowing that the decoded instruction is an if statement (line 14), we parse it to extract its condition. GAINDroid understand its condition as a guard condition because $Build.VERSION.SDK_INT$ was decoded. We continue to parse the condition and extract $>=$ and 23. At this point, GAINDroid knows that the next instructions to be decoded are only executed in Android version 23 or higher, and keeps this conditional information as context. The next decoded instruction is line 15. GAINDroid understands it as a method called by the app because its package is different from the ones which are part of Android APIs (e.g., android/content/res/Resources). Because our approach is also class-loader based and implements inter-procedural analysis, we can efficiently load *setText*() to analyze it. After loading *setText*(), GAINDroid starts analyzing it, but still preserving contexts (e.g., state in the caller method, current variables and conditions). The decoding of line 2 returns *setTextColor*(), *getColorStateList*(), $colorAccent$ and *getTheme*(). GAINDroid understands that *setTextColor*(), *getColorStateList*() and *getTheme*() are Android methods (they are part of android package) and verifies if they are present in the database for Android versions from 23 to 28, because we preserve the context that they are executed (only if $Build.VERSION.SDK_INT >= 23$). In this example, all the APIs have been found in the API database. Otherwise, GAINDroid would generate a warning, indicating that an API is not available within the searchable range of APIS (e.g., from 23 to 28). Once the analysis of *setText*() is over, GAINDroid

Algorithm 1 Finding API mismatches

```
1: procedure FindApiMismatches(block, app)
   ▷ Input: Block from data flow graph, decompiled APK
2:    if IsGuardStart(block) then
3:        (minLvl,maxLvl) ← GetGuard(block,minLvl,maxLvl)
4:    else if IsApiCall(block) then
5:        for each lvl in (minLvl..maxLvl) do
6:            if ¬apidb.Contains(block,lvl) then
7:                mismatches ← mismatches ∪ {block}
8:    else if IsMethod(block) then
9:        mismatches ← mismatches ∪ FindApiIn(block, minLvl, maxLvl)
10:   else if IsGuardEnd(block) then
11:       (minLvl,maxLvl) ← (app.minSdk,app.maxSdk)
12:   return mismatches
```

goes back to its original context (line 15), concluding the inter-procedural analysis. When we decode line 16, which contains a special flag, indicating the end of the path (end of the if statement), finishing the analysis of the first identified path. Line 17 represents another path in the program, an else statement. Because it does not have any conditions, GAINDroid negates the previous condition, transforming $Build.VERSION.SDK_INT >= 23$ to $Build.VERSION.SDK_INT < 23$. All the API calls inside the else statement are going to be verified if they exist from app's *minSdkVersion* to API level 22. The analysis of this new path works as the same as described for the if statement, but with a different condition. GAINDroid can also identify conditions for an else if statement.

Callback mismatch: The detector uses Algorithm 2 to detect API callback mismatches in each method within the call graph generated in *API Usage Extraction* module. If the method in the call graph overrides an API callback (line 2), we iterate over the API levels that the app declares to support and query the API

```
1 private void setText(TextView text){
2     text.setTextColor(resources.getColorStateList(
3         R.color.colorAccent , context.getTheme()));
4 }
5
6 @Override
7 protected void onCreate(Bundle b){
8
9   super.onCreate(b);
10  setContentView(R.layout.activity_main);
11
12  TextView text = findViewById(R.id.text);
13
14  if (Build.VERSION.SDK_INT >= 23) {
15     setText(text);
16  }
17  else { ... }
18 }
```

Listing 3.1: Code exploring context-sensitive, path-sensitive and inter-procedural analysis of our approach

database to determine whether the callback is defined within the entire range of supported API levels (lines 4-5). This is the major difference from our approach to prior research, such as Cider [68]. We consider all API callbacks in Android, without any manual modeling of callbacks. Our approach is also easier to scale, in terms of ability to detect APIs. We only need minimal manual work to support a new Android API, as described in Section 3.2

It is important to mention that for this detector, Jitana, our static analyzer framework, already gives a special flag to API callbacks methods. Therefore, GAINDroid loops through those nodes in the graph, performing decoding and API database search from *minSdkVersion* to *maxSdkVersion*, which is 28.

The second part of the *Mismatch Detection* component detects incompatibilities

Algorithm 2 Finding APC mismatches

```
1: procedure IsApcMismatch(method, app)
   ▷ Input: Method from call graph, decompiled APK
2:     if IsApiOverride(method) then
3:         for each lvl in (app.minSdk..app.maxSdk) do
4:             if ¬apidb.Contains(method, lvl) then
5:                 mismatches ← mismatches ∪ {method}
6:     return mismatches
```

Algorithm 3 Finding PRM mismatches

```
 1: procedure DetectPermissionMismatch(app, graph, permMap)
      ▷ Input:   Decompiled APK, call/data flow graph, permission map
      ▷ Output: List of detected mismatches
 2:   dangerousPerms ← GetDangerousPermsFromManifest(app)
 3:   if dangerousPerms = ∅ then
 4:       return ∅
 5:   callGraph ← BuildCallGraph(app)
 6:   if app.targetSdkVersion ≥ 23 then
 7:       for each method in callGraph do
 8:           if OverridesOnRequestPermissionsResult(method) then
 9:               return ∅
10:       return App does not implement runtime permission
11:   for each method in callGraph do
12:       dataFlowGraph ← GetDataFlowGraph(graph, method)
13:       for each block in dataFlowGraph do
14:           for each perm in dangerousPerms do
15:               if permMap.IsUsingPermission(perm, block) then
16:                   mismatches ← mismatches ∪ {perm}
17:   return mismatches
```

related to previous and the new run-time permissions system introduced in API level 23. A unique capability to our approach. The logic of algorithm 3 that checks permission-induced compatibility issues is as follows: First, extract dangerous permissions from the app's manifest (line 2). If there are no dangerous permissions there is no risk of permission mismatches, as normal permissions are automatically granted (lines 3-4). In case the app requests dangerous permissions, we get the call graph from the *API Usage Extraction* component (line 5). If the app targets a more recent version of Android (23 or higher), we check if the call graph contains a method overriding onRequestPermissionsResult (lines 6-8). In case the app does implement the new run-time permission system, there is no risk of mismatch (line 9). If after looping through all methods from the app, it did not implement the new run-time system, we return an error message, indicating that developers should

implement this functionality to avoid run-time crashes. If app targets an API level earlier than 23, the revocation of a dangerous permission could result in a run-time crash. To detect dangerous permission usages, GAINDroid iterates through each method in the call graph (line 11), retrieve the data flow graph for the method (line 12) and check whether each block in the data flow graph uses any of the dangerous permissions (lines 13-15). In case any dangerous permission is used, add it to the set of mismatches (line 16). While checking if a block in the data flow uses a dangerous permission, GAINDroid decodes the instruction, analyzes if it is an Android API invocation, and checks in the mapping produced in *Permission Mapping* (Section 3.3) if the API is used by any dangerous permission.

Chapter 4

Empirical Evaluation

This section presents the evaluation of GAINDroid. We have implemented GAINDroid's static analysis capability on top of the Jitana framework[66]. Jitana is a high-performance hybrid analysis tool for Android. It works directly on Dalvik executable (dex) files contained in each APK. We also use APKTool [28] to decompile apks, thus accessing its manifest file and binaries As a result, our approach implementation does not require the source code from apps, only their executables (apk). GAINDroid, can be used not only by developers but also by end-users as well as third-party reviewers to assess the compatibility of their mobile apps.

We further modified Jitana to decode dex files using Android version 6.0.0, which is the version in which the new runtime permissions system is introduced. We also extended Jitana to perform inter-procedural dataflow analysis, which enabled us to detect more API related issues within different methods of an Android app.

To evaluate GAINDroid, we conducted experiments to answer the following research questions.

RQ1. *Accuracy:* What is the overall accuracy of GAINDroid in detecting compatibility issues compared to the other state-of-the-art techniques?

RQ2. *Applicability:* How well does GAINDroid perform in practice? Can it

find compatibility issues in real-world applications?

RQ3. *Performance:* What is the performance of GAINDroid's analysis to identify sources of compatibility issues?

4.1 Objects of Analysis

In order to evaluate the effectiveness and efficiency of GAINDroid, we compared GAINDroid against other state-of-the-art applications which utilize different approaches to detecting API-related compatibility issues. The most closely related works to ours are by Li et at. [51] (CiD) and by Huang et al. [46] (Cider). To make the comparison as direct as possible, we used obtained the 7 benchmarks apps (CiD-Bench) from [51] and the 20 apps (Cider-Bench) used in [46]. One of our baseline system, Lint, requires building the apps from source code to perform the compatibility analysis. Out of the 27 benchmark apps, eight apps cannot be built; therefore, they are excluded from the analysis, leaving the total of 19 apps used in our comparative study. Table 4.1 shows all benchmark apps, where apps from 1 to 12 belongs to Cider-Bench and the remaining apps are from CiD-Bench. The collection includes apps of varying sizes ranging from 10,400 to 294,400 lines of Dex code and up to tens of thousands of methods. The benchmark apps both support and target a variety of API levels, with minimum levels ranging from 10 to 21 and targets ranging from level 23 to 27.

We used the Android Studio version 3.1.3 to load and build all the benchmark apps. However, we were unable to build eight apps from CiD-Bench. The initial error was related to the Gradle plugin. The applications required to update to a more recent version of Gradle. After updating the plugin, the apps generated another error related to toolchains. We had a machine set up with all version of

Table 4.1: Basic Characteristics of Objects of Analysis

No.	App Name	Number of Downloads	Rating	KLoC	Number of Methods	minSdkVersion	targetSdkVersion
1	AFWall [2]	500K+	4.3	21.8	46184	15	23
2	DuckDuckGo [8]	1M+	4.4	10.4	45049	21	27
3	FOSS Browser [10]	5K+	4.2	18.0	10946	21	27
4	Kolab notes [13]	1K+	4.1	73.4	34969	16	26
5	MaterialFBook [14]	10K+	4.3	68.0	34526	17	27
6	Network monitor [16]	50K+	4.3	20.8	43942	14	27
7	NyaaPantsu [17]	-	-	14.1	66249	21	27
8	Padland [18]	-	-	58.9	8713	14	26
9	PassAndroid [37]	1M+	4.2	85.0	52696	14	25
10	Simple Solitaire [22]	10K+	4.5	294.4	27782	11	25
11	SurvivalManual [23]	1M+	4.7	49.4	35832	10	25
12	Uber ride [24]	100M+	4.2	12.7	25610	14	26
13	Basic [46]	-	-	10.1	5099	10	25
14	Forward [46]	-	-	10.2	6201	10	19
15	GenericType [46]	-	-	12.4	5564	10	25
16	Inheritance [46]	-	-	11.2	6039	10	25
17	Protection [46]	-	-	10.9	5198	10	25
18	Protection2 [46]	-	-	10.8	5988	10	25
19	Varargs [46]	-	-	12.7	5876	10	25

Android but the error persisted. We then installed the development environment in two laptops with a different operating system, but the error still persisted. Since we are also comparing our approach to Lint, which requires building apps, those eight apps were excluded from our analysis. We provide the basic characteristics of the remaining 19 apps in Table 4.1. As shown, the collection includes apps of varying sizes ranging from 10,400 lines of Dex code to 294,400 lines of Dex code and tens of thousands of methods. The apps both support and target a variety of API levels, with minimum levels ranging from 10 to 21 and targets ranging from level 23 up to 27.

To further evaluate the applicability of our tool in practice, we collected a set of real-world Android apps from two repositories of FDroid [9] and AndroZoo [32]. FDroid is a software repository that contains free and open source Android apps. Our collection of subject systems includes all 1.391 apps available from the FDroid repository. We also include 2,300 apps from AndroZoo, a growing repository of Android apps collected from various sources, including the official Google Play store [32]. We were unable to build 120 of the apps from AndroZoo so we excluded

them from our analysis, leaving 3.571 apps in total.

4.2 Variables and Measures

4.2.1 Independent Variables

Our analysis evaluates GAINDroid against other approaches which also perform analysis of compatibility issues.

CiD represents a state-of-the-art in detecting Android compatibility issues. It has been publicly released, and we are able to obtain the tool and compile it in our experimental environment. We use it as the baseline system to answer RQ1 and RQ3.

Cider is another state-of-the-art approach developed to analyze API compatibility issues. Unfortunately, it is not available in either source or binary forms at the time of writing this article. As such, we rely on their results as reported in [46] to answer RQ1 and RQ3.

Lint is a static analysis technique, shipped with the Android Development Tools (ADT), to examine code bases for potential bugs, including incompatible API usages. Lint performs the compatibility analysis as part of building apps, and thus requires the app source code to conduct the analysis. We use Lint to answer RQ1 and RQ3.

We also considered IctApiFinder [44] as a possible baseline technique. IctApiFinder was introduced at about the same time as Cider. Unfortunately, the tool is not publicly available and our attempts to contact the authors to request access were unsuccessful. Therefore, we did not use it in our study.

4.2.2 Dependent Variables

As dependent variables, we chose metrics allowing us to answer each of our three research questions.

To measure accuracy, we compare the number of detected compatibility issues with known issues as reported by prior work [46, 51]. For each analysis technique, we report true and false positives and false negatives thereof in detecting compatibility issues of the apps under analysis. Lastly, we report precision, recall, and F-measure for each technique.

To measure applicability, we report the number of detected compatibility issues in real-world apps. Finally, to measure performance, we report the analysis time and the amount of memory used by each of the analysis techniques, i.e., GAINDroid, CiD, and Lint.

4.2.3 Study Operation

To address RQ1 and RQ2, we executed GAINDroid, CiD, Cider and Lint once to identify sources of API incompatibility issues, and verified how many API calls were identified as problematic. To address RQ3, we performed this experiment three times and measured the amount of time and memory needed to perform the analysis of each app.

To perform this study we used a MacBook Pro running OS High Sierra version 10.13.3, with an 8GB memory and a 2.5GHz Intel Core i5. The performance times we obtained were all recorded within this environment.

4.3 Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. In this work, we have studied a smaller set of benchmark programs developed and released by prior research work [51, 46] so that we can directly compare our results with their previously reported results. However, we also extend our evaluation to employ over 3,590 complex real-world apps from other repositories, which in turn enabled us to assess our system in real-world scenarios, representative of those that engineers and analysts are facing.

The primary threat to internal validity involves potential errors in the implementations of GAINDroid and the infrastructure used to run CiD and GAINDroid. To limit these, we extensively validated all of our tool components and scripts to ensure correctness. By using the same objects as our baseline systems we can also compare the results produced by our approach with those previously reported to help with ensuring correctness.

The primary threat to construct validity relates to the fact that we study efficiency measures relative to applications of GAINDroid, but do not yet assess whether the approach helps software engineers or analysts address dependability and security concerns more quickly than current approaches.

Chapter 5

Results

The results of our analysis regarding mismatch detection are summarized in Table 5.1. For each of the 19 manually-inspected apps, we report the number of true and false positives (represented as ☑ and ☒, respectively) reported by the approach for that app for each category of mismatch (using the abbreviations introduced in Table 2.1) as well as the number of false negatives (☐), if an approach missed a mismatch detected by another approach and verified by manual inspection. The bottom of the table lists the precision, recall, and F-measure for each technique to summarize the overall effectiveness. According to [42], precision expresses the proportion of the data points that our model says was relevant actually were relevant, for example, APIs classified as mismatch that are actually a mismatch. Recall expresses the ability to find all relevant instances in a dataset, it means that, the capability of finding every instance that is a mismatch. Finally, F-measure is the harmonic mean of precision and recall, in other words, it is the optimal blend of precision and recall. As reported by [41], it is important to have more robust metrics such as the ones mentioned above because they are more suitable for dealing with results such as true positives, false positives and false negatives.

Table 5.1: Effectiveness and Efficiency of GAINDroid

	App	GAINDroid			CiD			Cider			Lint		
		API	APC	PRM	API	APC	PRM	API	APC	PRM	API	APC	PRM
Cider-Bench	AFWall+	☑(9)	☑(7)		☐(9)	-		-	☑☐(6)	-	☑(1)☐(8)	☐(7)	
	DuckDuckGo		☐		☒(3)	-		-	☑	-		☐	
	FOSS Browser		☑(7)	☑	☒(4)	-	☐	-	☐(7)	-		☒(3)☐(7)	☐
	Kolab notes	☑(3)☒(9)		☑	☑(3)☒(13)	-	☐	-	☒	-	☐(3)		☐
	MaterialFBook	☑(11)☒☐(3)			☑(14)☒(17)	-		-		-	☐(14)		
	NetworkMonitor		☑(5)			-		-	☐(5)	-		☐(5)	
	NyaaPantsu		☑(12)			-		-	☐(12)	-		☐(12)	
	Padland		☐		☒(4)	-		-	☑	-		☒(2)☐	
	PassAndroid	☑(9)	☑(3)		☐(9)	-		-	☐(3)	-	☐(9)	☐(3)	
	SimpleSolitaire	☑☒	☑(2)		☑☒(10)	-		-	☑☐	-	☐	☒(2)☐(2)	
	SurvivalManual				☒(19)	-		-		-			
	Uber ride		☑(4)		☒(2)	-		-	☑(4)	-		☒(1)☐(4)	
CiD-Bench	Basic	☑			☑	-		-	-	-	☐		
	Forward	☑			☑	-		-	-	-	☐		
	GenericType					-		-	-	-			
	Inheritance	☑(2)			☑(2)	-		-	-	-	☐(2)		
	Protection	☑			☑	-		-	-	-	☐		
	Protection2	☑			☑	-		-	-	-	☐		
	Varargs	☑(2)			☑(2)	-		-	-	-	☐(2)		
Precision:		79%	100%	100%	27%	-	0%	-	89%	-	100%	0%	0%
Recall:		93%	95%	100%	59%	-	0%	-	19%	-	2%	0%	0%
F-Measure:		85%	98%	100%	42%	-	0%	-	31%	-	4%	0%	0%

In order to demonstrate the efficiency of each tool, we reported the analysis time for each approach against all the 19 benchmark apps. Table 5.2 shows those results, along with the maximum, minimum and average time to evaluate an app. It is important to notice that all the unit of measurement is in second.

5.1 RQ1: Accuracy

For RQ1, we evaluated the ability of our tool to find API invocation and callback mismatches by comparing against CiD [51] ,Cider [46] and Lint [3]. We divide the results into two parts, API invocation and callback mismatches, respectively.

API Invocation: We first start utilizing the 12 apps that we were able to built from [46] and applying GAINDroid, CiD and Lint to them.

As shown in Table 5.1, our approach reports less false positives and more true positives than CiD and Lint.

Our approach and Lint were able to analyze all the 12 apps, while CiD did not halt for 4 apps (*AFWall*, *NetworkMonitor*, *NyaaPantsu* and *PassAndroid*).

Comparing against CiD, GAINDroid shows much higher precision and comparable recall, contributing for than double of F-Measure. CiD has a slightly higher recall because the aforementioned tool finds less false negatives than GAINDroid. However, GAINDroid identifies less false positives than CiD. Thus, contributing to higher precision. The tool Lint has higher precision than GAINDroid because it has found no false positives (but only one true positive). However, it has found many false negatives, corroborating for lower recall. Its F-Measure is 19 times lower than GAINDroid. After this analysis and based on a higher F-Measure compared to CiD and Lint, it is possible to conclude that GAINDroid indeed better in finding API invocation mismatches, with a convincing number of true

positives and lower false positives and false negatives. Below we describe the reasons that our approach found false positives and false negatives.

The apps *Kolab notes*, *MaterialFbook* and *SimpleSolitaire* are examples of false positives and false negative, respectively. In the first app, the false positive happens because of the invocation of *getFileName()* in method *onBindViewHolder(ViewHolder, int)* from class *AttachmentRecyclerViewAdapter*. It is a call to a method which belongs to an external library. Jitana understands this method as being part of the API, searches for it in the database and does not fin it. Therefore, generating a false positive. We would have to identify in the static analysis framework, why this error happens while decoding certain instructions. In the second app, The false positive happens because a correctly guarded call to an Android API (line 27 and 29) in a private method in the *MainActivity* class is made from inside an anonymous class (line 8), as shown in Listing 5.1. Finally, the false positive in *SimpleSolitaire* is due to an error while decoding the instruction by the static analysis framework. Jitana incorrectly identified a call to *Checkable.setChecked*, which has been in the stack of Android frameworks since version 1, as a call to *TwoStatePreference.setChecked*, which was introduced in API level 14. The false negative happens because of the invocation of *getUrl()* in method *onCreate* from class *MainActivity* of app *MaterialFBook*. The API is triggered by a method inside an anonymous class. The decoding of anonymous class is a challenge that will definitely improve our results once implemented. However, due to time limitation, the identification of anonymous classes will be left as future work. Furthermore, the reason that Lint yield many false negatives is because it does not analyze methods with the annotation *@TargetApi* or *SuppressLint*. For example, app *PassAndroid* utilizes the mentioned annotation in the method *doPrint* in class *PrintHelper*. The method uses *android.print.PrintManager* which was introduced only in API level 19.

```
1     @Override
2     public boolean onCreateOptionsMenu(Menu menu) {
3         ...
4         notif.setOnClickListener(new View.OnClickListener() {
5             @Override
6             public void onClick(View v) {
7                 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP)
8                     circleReveal(v);
9                 mWebView.loadUrl(baseURL + "notifications.php");
10                setTitle(R.string.nav_notifications);
11                Helpers.uncheckRadioMenu(mNavigationView.getMenu());
12                NotificationsJIS.ClearbyId(MainActivity.this, 1);
13            }
14        });
15        ...
16        return true;
17    }
18
19    @SuppressWarnings("NewApi")
20    private void circleReveal(int viewID, final boolean show) {
21        final View v = findViewById(viewID);
22
23        int cy = v.getHeight() / 2;
24
25        Animator anim;
26        if (show)
27            anim = ViewAnimationUtils.createCircularReveal(v, v.getWidth(), cy, 0,
                  v.getWidth());
28        else
29            anim = ViewAnimationUtils.createCircularReveal(v, v.getWidth(), cy, v.
                 getWidth(), 0);
30
31        ...
32
33        anim.start();
34    }
```

Listing 5.1: Perfectly guarded API invocation but identified as false positive

However, the app *minSdkVersion* is 14, then the aforementioned method can fail in Android versions below 19.

GAINDroid also identifies more true positives than CiD and Lint in some scenarios. For instance, in the app *Simple Solitaire*, while CiD only reports only one true positive and Lint none. Our approach has found two (method *setChecked* from class *TwoStatePreference* in addition to *onAttach* from class *Fragment*). We have noticed that CiD presents more false positives than our approach. During manual verification, it was possible to identify that CiD reports API invocation issues even if they are bounded by conditions, in other words, using a guard. For example in

app *SurvivalManual*, where *setTextIsSelectable(boolean)* is only executed if the app runs on a device with Android level greater or equal than 11.

The second app set-up consists of utilizing the 7 apps from [51]. As demonstrated in Table 5.1, GAINDroid was able to find all the errors reported by CiD. Once again, Lint has only reported false negatives and the reason is all the API methods causing errors are in methods annotated with *@TargetApi* or *SuppressLint*. Therefore Lint will not check those methods.

With respect to API callback issues, as presented in Table 5.1, our approach finds more true positives and less false positive errors compared to Cider and Lint. This is expected because our approach is automatic and does not require manually constructed callback control flow graph as being done with Cider. Cider also only detects 4 families of API callbacks (Activity, Fragment, Service and WebView). Instead, GAINDroid relies on a database of Android methods derived from each API level. Therefore, it can support a larger number of callback APIs than the 24 within the 4 families of API callbacks supported by Cider. For example, Cider finds 1 true positive for the app *AFWall* while GAINDroid finds 7 true positives. The reported result shows that callbacks such as *PreferenceActivity*, *ContentObserver* and *onAuthenticationError* from classes that Cider does not support, have been found as problematic. However, there are some callbacks that Jitana cannot decode into its call-graph, generating false negative. For example, in apps DuckDuckGo and Padland, both of these mismatches occur in anonymous classes which override callbacks. As seen in Listing 5.2, app Padland, a callback that can be invoked by the system but, it only exists in Android level 23 and further versions. The app may fail when installed on devices below Android 23.

```java
1 private WebView _makeWebView() {
2
3     ...
4
5         String[] url_whitelist = getServerWhiteList();
6         webView.setWebViewClient(new PadLandSaferWebViewClient(url_whitelist) {
7
8             ...
9
10            @Override
11            public void onReceivedError(WebView view, WebResourceRequest request,
                  WebResourceError error) {
12                super.onReceivedError(view, request, error);
13                --webview_http_connections[0];
14                _hideProgressWheel();
15                Log.e(TAG, "WebView␣Error␣" + error.toString() +",␣Request:␣"+
                      request.toString());
16            }
17        }
18 }
```

Listing 5.2: Callback being invoked inside an anonymous class

5.2 RQ2: Real-World Applicability

To evaluate the implications of our tool in practice, we applied GAINDroid to real-world apps collected from [9] and [32]. GAINDroid detected 68,268 potential API invocation mismatches, with 41.19% of the apps harboring at least one potential mismatch. It also identified 2,115 potential API callback mismatches occurring in 20.05% of the apps under analysis. To perform the permission-induced mismatch analysis, we divided the apps into two groups based on the target SDK version: (i) 1,815 apps target Android API levels greater than or equal to 23 and (ii) 1,756 apps target Android API levels below 23. We identified a total of 1,430 apps across both groups with at least one permissions-induced compatibility issue. 224 apps (12.34%) in the group (i) attempt to use dangerous permissions without implementing the runtime permissions request system, and 1,206 apps (68.68%) in the group (ii) are vulnerable to permissions revocation mismatches.

We then manually investigated the GAINDroid's results to appraise its utility in practice. In the following, we report some of our findings. To avoid revealing

previously unknown compatibility issues, we only disclose a subset of those that we have had the opportunity to bring to the app developers' attention.

API invocation mismatch. In the Offline Calendar app [30], the invocation of the *getFragmentManager()* API method in *PreferencesActivity.onCreate* causes an API invocation mismatch. The *getFragmentManager()* method was added to the *Activity* class in API level 11. Also, Offline Calendar sets its *minSdkVersion* to API level 8. Therefore, as soon as the *PreferencesActivity* is activated, the *Offline Calendar* app will crash if running on API levels 8 to 11. The mismatch could be resolved by wrapping the call to *getFragmentManager()* in a guard condition to only execute it if the device's API level is equal or greater than 11, or by setting the *minSdkVersion* to 11.

API callback mismatch. FOSDEM [29] is a conference companion app. It exhibits an API callback mismatch in its *ForegroundLinearLayout* class, which overrides the *View.drawableHotspotChanged* callback method, introduced in API level 21. However, its *minSdkVersion* is set to API level 15, which would not support the aforementioned callback method, and in turn may not properly propagate the new hotspot location to the *Drawable* stored as a member of the layout class. This could lead to crashes or other instability in the app's interface. Setting the *minSdkVersion* to 21 would resolve the mismatch.

Permission request mismatch. Kolab Notes [13] is a note-taking app that can synchronize notes with other apps. It exhibits a permission request mismatch. The app targets API 26 and uses the WRITE_EXTERNAL_STORAGE permission, but does not implement the necessary methods to request the permission at runtime. If the permission is not already granted when the user attempts to save or load data to/from an SD card, the action will fail. To resolve the mismatch, the developers should update the app to implement the new runtime permissions request system,

particularly the *onRequestPermissionsResult* callback.

Permission revocation mismatch. AdAway [1] is an ad blocking app that suffers from a permission revocation mismatch. The app targets API level 22 and uses the WRITE_EXTERNAL_STORAGE permission, which could be revoked by the user when installed on a device running API 23 or greater. If the user revokes the permission and tries to export a file, the app will crash. The developers could resolve the issue by updating the app to use runtime permissions and setting the *minSdkVersion* to 23.

5.3 RQ3: Performance

In this last research question, we evaluated the efficiency of GAINDroid and compared its running time against CiD and Lint. Note that we could not compare against the performance of Cider because the authors of that work neither reported the analysis time of their technique nor made the tool available. Since Cider authors have not replied to our inquiries, we were unable to measure its analysis time.

Table 5.2 shows the analysis time (in seconds) of GAINDroid and the two state-of-the-art tools mentioned above. Dashes indicate that a system fails to produce analysis results after 600 seconds. As shown, the analysis time taken by GAINDroid is significantly lower than those of CiD and Lint for almost all the apps. Also, note that CiD fails to completely analyze four apps after 600 seconds have passed. The average analysis time taken by GAINDroid, CiD, and Lint per app is 5.7, 22.9 and 17.4 seconds respectively, corroborating that GAINDroid can efficiently vet Android apps for compatibility issues in a fraction of time taken by the other state-of-the-art tools.

Table 5.2: Time comparison between GAINDroid and state-of-the-art tools

	App	GAINDroid Time (s)	CID Time (s)	Lint Time (s)
Cider-Bench	AFWall	8.2	–	41.3
	DuckDuckGo	7.7	60.3	35.1
	FOSS Browser	3.6	17.2	30.3
	Kolab notes	7.2	16.5	22.8
	MaterialFBook	6.2	19.6	12.3
	NetworkMonitor	8.2	–	40.1
	NyaaPantsu	11.3	–	27.4
	Padland	2.3	13.3	11.1
	PassAndroid	9.9	–	32.5
	SimpleSolitaire	6.3	13.2	20.6
	SurvivalManual	7.2	60.1	10.5
	Uber ride	4.7	15.8	25.8
CiD-Bench	Basic	3.9	21.1	2.5
	Forward	1.8	6.2	2.5
	GenericType	4.1	18.7	2.6
	Inheritance	3.8	19.2	3.1
	Protection	3.9	17.1	3.5
	Protection2	3.9	21.2	3.1
	Varargs	3.8	23.5	3.8
Max:		**11.3**	**60.3**	**41.3**
Min:		**1.8**	**6.2**	**2.5**
Average:		**5.7**	**22.9**	**17.4**

Figure 5.1 presents the time taken by GAINDroid to perform compatibility analysis on real-world apps. The scatter plot depicts both the analysis time and the app size. As shown in the figure, our approach analyzes 98% of the apps in less than 20 seconds. The experimental results show that the average analysis time taken by GAINDroid, CiD, and Lint per app on real-world data sets are 6.2 seconds (ranging from 1.6 to 37.8 seconds), 29.5 seconds (ranging from 4.1 to 78.4 seconds), and 24.7 seconds (ranging from 4.7 to 75.6 seconds), respectively. We

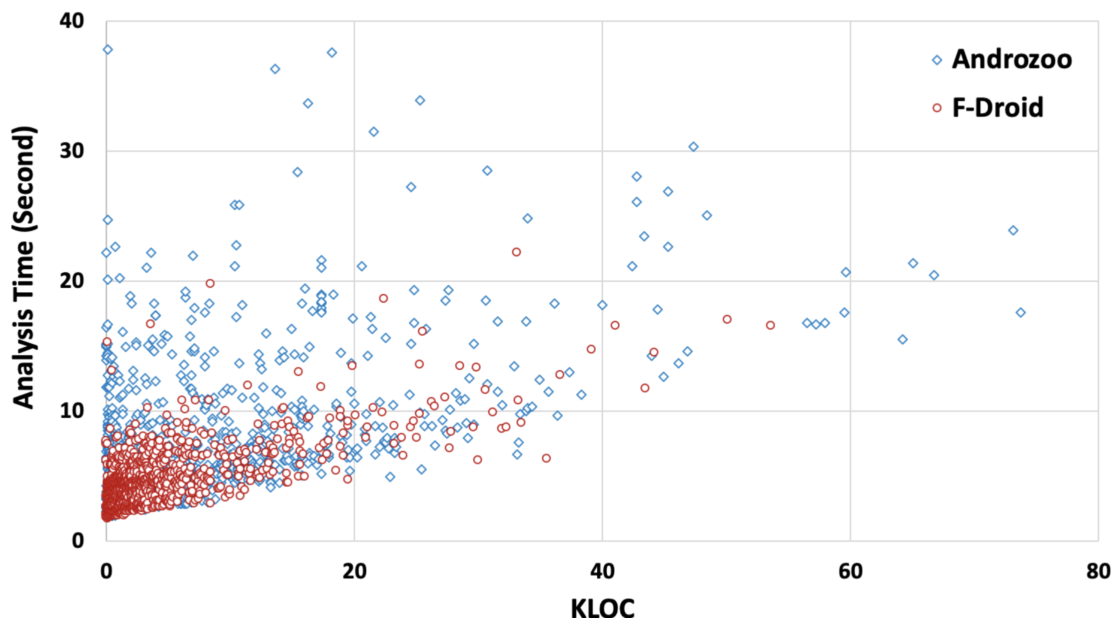

Figure 5.1: Scatter plot representing analysis time for compatibility checking of Android apps using GAINDroid

have found outliers during the analysis. For example, the app in the top left corner in Figure 5.1 is a game application which extensively uses third-party libraries, which took a considerable amount of time for our analysis framework to compute the data structures for analysis, despite its small KLOC. On the other hand, the app in the right side of the diagram, closer to 80 KLOC, loads three times fewer library classes than the aforementioned app, implicating in less complex graphs to analyze. Overall, the timing results show that on average GAINDroid is able to complete analysis of real-world apps in just a few seconds (on an ordinary laptop), confirming that the presented technology is indeed feasible in practice for real-world usage.

To better understand why GAINDroid performs more efficient than the state-of-the-art approaches, we conducted a further performance evaluation, comparing the number of resources and analysis efforts required by each approach. Since our

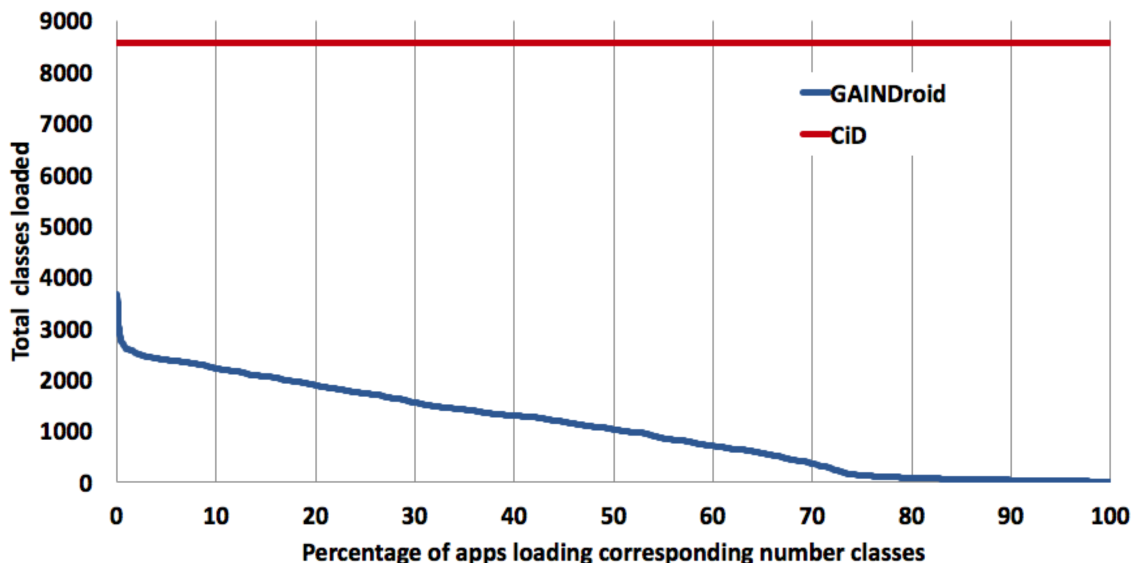

Figure 5.2: Number of classes loaded by GAINDroid and CiD when analyzing real-world Android apps.

approach extends a class-loader based program analysis framework rather than a compiler based program analyzer, we expect the efficiency gains in GAINDroid is due to the effective loading of classes during the analysis. In this set of experiments, we attempted to corroborate our intuition and obtain empirical evidence of this relationship.

We first monitored the number of analyzed classes in each approach. Figure 5.2 depicts the number of classes loaded by GAINDroid and CiD when analyzing real-world apps. The red line in Figure 5.2 shows that CiD loads all Android classes from the latest available Android framework [26]. As of January 2019, there are 8552 classes in the Android framework. On the contrary, GAINDroid only loads the classes that the app actually uses. According to the diagram, GAINDroid, shown by the blue line in Figure 5.2, at most loads 3,600 classes, and that only occurs for a very small number of apps. Indeed, for over 60% of the analyzed apps, GAINDroid loads less than 1,000 classes, which is eight times more efficient

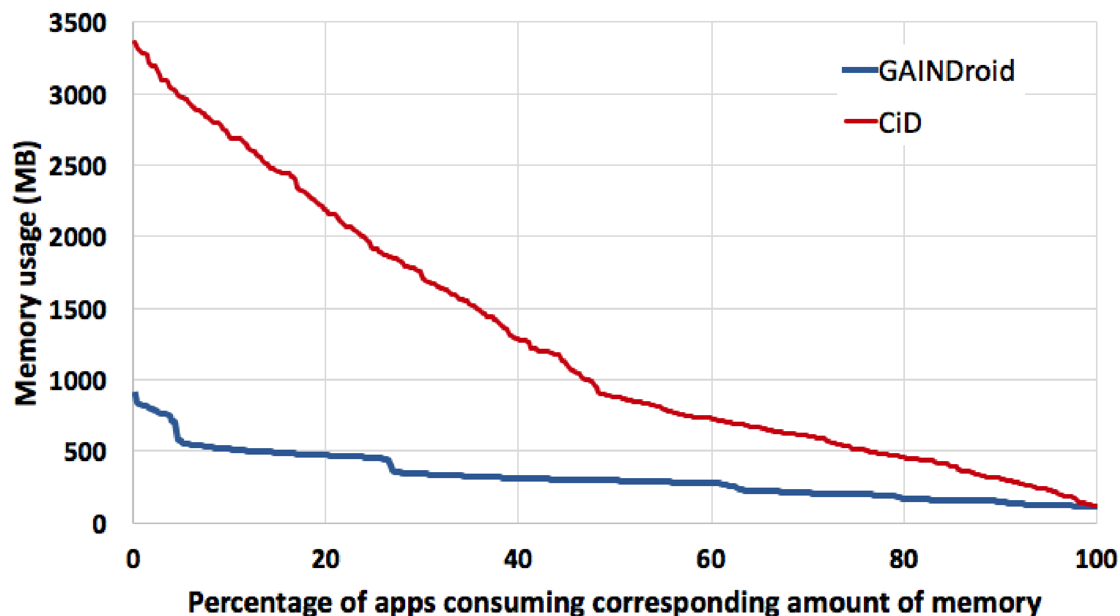

Figure 5.3: Amount of memory used by GAINDroid and CiD when analyzing real-world Android apps.

compared to CiD.

Loading fewer classes also allows GAINDroid to require less memory to perform its analysis. To investigate this matter, we also monitored the memory footprint required by each approach for performing analysis. Figure 5.3 shows a comparison of how much memory GAINDroid and CiD are using during the analysis of real-world apps. According to the results, GAINDroid on average requires 329 MB (ranging from 119MB to 898MB) of memory to perform the compatibility analysis. On the other hand, CiD on average uses 1.3 GB (or four times more memory) to perform the same analysis. We interpret this data as corroborating the effectiveness of our technique based on a class-loader based approach for compatibility analysis.

Chapter 6

Discussion

We were expecting that apps supporting a wider range of API levels would yield more compatibility issues. However, based on the experimental results presented in Table 5.1, this behavior is not being observed. For example, The two apps with the most detected incompatibility support 10 and 8 API levels. Those apps are *MaterialFBook* and *AFWall*, respectively. On the other hand, the two apps that support the widest ranges of API level (15 for *SurvivalManual* and 14 for *Simple Solitaire*), only show 0 and 1 incompatibility issues, respectively.

The Android framework evolves fast, with at least one release per year. Therefore, it is important for GAINDroid includes new API methods to its database easily. For this end, our tool only needs a text file with all API methods from a given Android level. Adding new API levels will only introduce a negligible overhead while loading the database for analysis.

It is known that Android is trying to decrease API and permission incompatibilities by enforcing new rules to developers. According to [15], Google Play will require that new apps target at least Android API level 26 from August 1, 2018, and that updated apps target Android API level 26 from November 1, 2018. Another announcement [20] states that Android 9 (API level 28) introduce new restriction regarding the use of hidden APIs, which is the use of Android APIs that are no

longer public. However, it will still be possible to download apps from repositories other than Google Play, such as [9] and [7], which can present runtime failures. Because of this restriction, a new module able to detect the use of inaccessible APIs is desirable and we target it as future work.

Chapter 7

Related Work

Android incompatibility issues have received a lot of attention recently. Here, we provide a discussion of the related efforts in light of our research.

API evolution. A large body of existing research focuses on the evolving nature of APIs, which is an an important aspect of software maintenance [57], [35], [52], [48], [53], [40], [56], [62]. These research efforts explore the problems that are introduced by API changes. Among others, McDonnell et al. [57] studied Android's fast API evolution (115 API updates/month), and noticed developers' hesitation in embracing the fast-evolving APIs. The results of this study suggest that API updates are more defect-prone than other types of changes, which might cause application instability and potential vulnerabilities. Mutchler et al. [62] explored the consequences of running applications targeted to older Android versions on devices employing recent Android versions, and how it can introduce serious security issues. Li et al. [52] investigated the frequency with which deprecated APIs are used in the development of Android apps, considering the deprecated APIs' annotations, documentations, and removal consequences along with developers' reactions to APIs deprecations. Bavota et al. [35] showed that applications with higher user ratings use APIs that are less change- and fault-prone compared to the applications with lower ratings. Linares et al. [53] studied

the impact of using evolving APIs, concluding that it can significantly hurt an application's success.

These prior research efforts clearly motivate the need to address issues that can arise from API evolution. However, their approaches do not provide detailed technical solutions or methods to systematically detect the root causes of these problems. GAINDroid, on the other hand, is designed to be effective at detecting API related issued that can cause runtime crashes.

Android fragmentation. The other relevant thrust of research has focused on investigating the Android ecosystem by running different custom Android distributions on different hardware to identify potential application instability and uncovering the causes [43], [49], [63], [55], [70], [67], [31]. Aafer et al. [31] investigated how modifying the operating system can introduce security problems within the mobile OS. Han et al. [43] studied the bug reports related to HTC and Motorola devices in the Android issue tracking system, and discovered that The android ecosystem was fragmented, meaning that applications might behave differently when installed on phones from different vendors. Liu et al. [55] observed that a noticeable percentage of Android performance bugs occur only on specific devices and platforms. Moran et al. [59] presented a systematic input generation is driven by both static and dynamic analyses to trigger app crashes. Given such automatically generated inputs, it produces a a crash report containing screenshots, detailed crash reproduction steps, the captured exception stack trace, and a fully replayable script that automatically reproduces the crash on a target device.

These research efforts primarily focused on behavioral differences when an app is installed on different operating systems and/or hardware platforms. They mainly rely on hardware specifications and changes in the Android documentation

Table 7.1: Comparing GAINDroid to the state-of-the-art of compatibility detection techniques.

	API	APC	PRM
CiD [51]	*V*	*X*	*X*
Cider [46]	*X*	*V*	*X*
IctApiFinder [44]	*V*	*X*	*X*
Lint [3]	*V*	*X*	*X*
GAINDroid	*V*	*V*	*V*

to uncover potential compatibility or behavioral issues. Therefore, these approaches are not useful when such platform related information is incomplete, inconsistent, or unavailable. Furthermore, applying these approaches to test an application on the entire vast hardware ecosystem of Android devices may not be feasible due to exponentially large system configurations. Our work, on the other hand, focuses on a more tractable and important problem due to API evolution and how it can affect the apps and their performance regardless of the operating system distribution or the hardware the applications are running on.

API incompatibility. A number of techniques have been recently developed for detecting Android API incompatibilities. In Table 7.1, we compare the detection capabilities of GAINDroid against the current state-of-the-art approaches. It is important to stress that GAINDroid is the only solution that provides the capability of automatically detecting various types of Android compatibility issues, i.e., API invocation compatibility issues (API), API callback compatibility issues (APC), and permission-induced compatibility issues (PRM).

Wu et al. [68] investigated side effects that may cause runtime crashes even within an app's supported API ranges, inspiring subsequent work. Huang et al. [46] aimed to understand callback evolution and developed Cider, a tool capable of identifying API callback compatibility issues. However, Cider's analysis relies on manually built PI-GRAPHS, which are models of common compatibility

callbacks of four classes: Activity, Fragment, Service, and WebView. CIDER thus does not deal with APIs that are not related to these classes or permission induced mismatches. Moreover, by only focusing on callback classes, their reported result is a subset of ours. In addition, CIDER's API analysis is based on the Android documentation, which is known to be incomplete [68]. Our work, on the other hand, automatically analyzes each API level in its entirety to identify all existing APIs. This allows our approach to be more accurate in detecting actual changes in API levels, as there are frequent platform updates and bug fixes. As a result, and as confirmed by the evaluation results, our approach features much higher precision and recall in detecting compatibility issues.

Lint [3] is a static analysis tool introduced in ADT (Android Development Tools) version 16. One of the benefits of Lint is that the plugin is integrated with the Android Studio IDE, which is the default editor for Android development. The tool checks the source code to identify potential bugs such as layout performance issues and accessing API calls that are not supported by the target API version. However, the tool generates false positives when verifying unsupported API calls (e.g., when an API call happens within a function triggered by a conditional statement). Another disadvantage is that it requires the availability of the original source code, and it does not analyze Android application packages, i.e., apk files. In addition, LINT requires the project to be first built in the Android Studio IDE before conducting the analysis. Unlike LINT, GAINDROID operates directly on Dex code. While LINT claims to be able to detect API incompatibility issues, our experimental results as well as the results obtained by Huang et al. indicate that LINT is not as effective as GAINDROID or CIDER.

Li et al. [51] provided an overview of the Android API evolution to identify cases where compatibility issues may arise in Android apps. They also presented

CiD, an approach for identifying compatibility issues for Android apps. This tool models the API lifecycle uses static analysis to detect APIs within the app's code, and then extracts API methods from the Android framework to detect backward incompatibilities. CiD supports compatibility analysis up to the API level 25. In comparison, GAINDroid offers automated extraction of the API database, and thereby supports up to the most recent Android platform (API level 28). Moreover, in contrast to GAINDroid, CiD did not consider incompatibilities regarding the runtime permission system.

Wei et al. [67] conducted a study to characterize the symptoms and root causes of compatibility problems, concluding that the API evolution and problematic hardware implementations are major causes of compatibility issues. They also propose a static analysis tool to detect issues when invoking Android APIs on different devices. Their tool, however, needs manual work to build API/context pairs, of which they only define 25. Similar to our prior discussion of work by Huang et al., the major difference between our work and this work is that our approach can focus on all API methods that exist at an API level. Again, the result reported by their approach would be a subset of our detected issues.

Chapter 8

Conclusion and Future Work

We have presented GAINDroid, a tool that operates on the Dexcode level to identify misuse of the Android API and the permission system, which may lead to runtime crashes. We have compared our results with those of CiD, Cider and Lint and we can conclude that GAINDroid is more effective than the other three state-of-the-art approaches. It can detect more sources of potential mismatches while yielding less false positives. Metrics such as precision, recall, and F-Measure supports our claim that our approach performs better than the other three state-of-the-art tools. GAINDroid can also perform analysis at least 3 times faster than CiD and Lint.

We further conducted a case study using 3,571 real-world Android apps downloaded from Androzoo [32] and F-Droid [9]. Again, we used GAINDroid to detect potential API invocation and callback mismatches, along with permission request and revocation mismatch. Our analysis results report that 41,19% of the downloaded apps can have at least one potential API invocation mismatch, they also show that 20,05% of the analyzed apps can crash due to API callback mismatch. With respect to permission mismatches, 12,34% suffer from permission request mismatch while 68,68% of the downloaded apps can suffer from crashes due to permission revocation mismatch. The obtained results demonstrate that

such problems are still present in modern Android apps. The study also showed that GAINDroid can be used in real-world Android applications to perform a large-scale analysis.

With respect to future work, we have a few items that we plan to address. Our immediate plan is to contact Huang et al. [46] and He et al. [44] and request access to Cider and IctApiFinder so that we can conduct more extensive experiments to compare their effectiveness and efficiency with our work. We also would like to evaluate if they can handle misuse of the Android permission mechanism. We also plan to contact the authors of apps that showed mismatches problems. The Android framework evolves fast and developers tend to not engage in keeping their apps up to date to be compatible with new features and APIs in the most recent versions of the framework.

In terms of implementation of additional features to increase true positives in GAINDroid, we plan to analyze anonymous classes, inaccessible APIs and identify behavioral changes in Android APIs.

As described in the Result section, GAINDroid is unable to identify API methods within anonymous classes in apps such as *MaterialFBook*, *DuckDuckGo* and *Padland*. Therefore we plan to implement a module that is able to identify anonymous classes and analyze its content. Thus, recognizing API invocations and callbacks that could not be analyzed previously, culminating in an increase of true positives for the aforementioned API related mismatches.

It is known that is possible to access inaccessible APIs, such as internal and hidden APIs [50]. Such APIs are often classified as not stable enough to be promoted or are still subject to invasive changes in future releases. However, developers may find more convenient to implement their desired functionalities with hidden APIs and consequently, to use them along with public APIs. As stated

by Google [20], more restrictions regarding the use of inaccessible APIs will be implemented, starting in Android level 28. Therefore, the scenario where an app using a hidden API works flawlessly but, after a system update, the app may crash because the method is not available anymore due to a system restriction, can happen. In order to warn developers about this upcoming issue, we want to include the capability to identify internal and hidden methods within an Android app.

We are also interested in identifying behavioral changes in API methods across different levels of Android APIs. The reason for this module is because methods can still have the same signature but, different implementations in order to improve performance and security. We want to create a repository containing all changes of a method and whenever a developer uses an API, GAINDroid would send an alert to the user, mentioning that the API has changes on its behavior. Maybe this change in behavior can lead to a runtime crash. For example, API methods *WifiInfo.getMacAddress()* and *BluetoothAdapter.getAddress()* return now return the default constant value of *02:00:00:00:00:00* starting from Android level 23. Such behavior change has initiated many discussions online [11].

In addition, another idea is to provide guidance to users to replace the use of possibly outdated or deprecated APIs with more updated ones. For example, GAINDroid would recommend a developer to replace *Apache HTTP Client* class by the class *HttpURLConnection*.

Bibliography

[1] Adaway. https://github.com/AdAway/AdAway/releases/tag/v3.0.2". 2.3, 5.2

[2] Afwall. https://github.com/ukanth/afwall/tree/71e6c66". 4.1

[3] Android lint. http://tools.android.com/tips/lint. 5.1, 7.1, 7

[4] Android market share. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. 1

[5] Android runtime permissions. https://source.android.com/devices/tech/config/runtime_perms. 2.3, 2.3

[6] Android versions. https://en.wikipedia.org/wiki/Android_version_history. 2.1

[7] Bazaar repository. https://cafebazaar.ir. 6

[8] Duck duck go. https://github.com/duckduckgo/Android/tree/2d7d379". 4.1

[9] F-droid repository. https://f-droid.org/. 4.1, 5.2, 6, 8

[10] Foss browser. https://github.com/scoute-dich/browser/commit/e08f5b6". 4.1

[11] Getting mac address in android 23. http://stackoverflow.com/questions/33159224/getting-mac-address-in-android-6-0. 8

[12] Google play apps repository. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. 1

[13] Kolab notes. https://github.com/konradrenner/kolabnotes-android/commit/14ba3c3". 4.1, 5.2

[14] Materialfbook. https://github.com/ZeeRooo/MaterialFBook/tree/2cb3c61". 4.1

[15] Meet google play's target api level requirement. https://developer.android.com/distribute/best-practices/develop/target-sdk". 6

[16] Network monitor. https://github.com/caarmen/network-monitor/tree/0e17b95". 4.1

[17] Nyaa pantsu. https://github.com/NyaaPantsu/NyaaPantsu-android-app/tree/53ad9a8". 4.1

[18] Padland. https://github.com/mikifus/padland/commit/38f7e66". 4.1

[19] Permissions in android. https://developer.android.com/guide/topics/permissions/overview#permission-groups. 1, 2.3

[20] Restrictions on non-sdk interfaces. https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces. 6, 8

[21] sdkmanager tool. https://developer.android.com/studio/command-line/sdkmanager. 3

[22] Simple solitaire. https://github.com/TobiasBielefeld/Simple-Solitaire/commit/1483ee". 2.2.2, 4.1

[23] Survival manual. https://github.com/ligi/SurvivalManual/tree/13b1f43". 4.1

[24] Uber rides. https://github.com/uber/rides-android-sdk/tree/4d77c38". 4.1

[25] Using sdk in android apps. https://developer.android.com/guide/topics/manifest/uses-sdk-element. 2.1, 2.2

[26] Android - aosp. https://github.com/aosp-mirror/, 2018. 5.3

[27] Android platform frameworks base. *https://github.com/aosp-mirror/platform_frameworks_base/releases*, August 2018. 1

[28] Apktool. https://ibotpeaches.github.io/Apktool/, 2018. 4

[29] FOSDEM Companion. https://github.com/cbeyls/fosdem-companion-android/releases/tag/1.5.0, 2018. 5.2

[30] Offline Calendar. https://github.com/PrivacyApps/offline-calendar/releases/tag/v1.8, 2018. 5.2

[31] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting inconsistent security configurations in custom android roms via differential analysis. In *USENIX Security Symposium*, pages 1153–1168, 2016. 7

[32] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community.

In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016. 4.1, 5.2, 8

[33] AndroidCentral. Phone Died During System Update, 2013. http://forums.androidcentral.com/htc-desire-c/265098-phone-died-during-system-update.html. 1

[34] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012. 3.3

[35] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015. 7

[36] A. Bera. How To Fix Apps Crashing After 4.4 Kit-Kat Update Problem On Nexus 7, 2016. http://www.technobezz.com/fix-apps-crashing-4-4-kitkat-update-problem-nexus-7/. 1

[37] Marcus Bueschleb. PassAndroid. `https://github.com/ligi/PassAndroid/tree/fcd9ba"`, 2018. 4.1

[38] Dave Burke. Introducing Android 9 pie. https://android-developers.googleblog.com/2018/08/introducing-android-9-pie.html, 2018. 2.1

[39] Zach Epstein. Did Apps Just Start Crashing Constantly on Your Android Phone?, 2015. http://bgr.com/2015/04/28/android-tips-tricks-fix-crashing-apps/. 1

[40] Mattia Fazzini and Alessandro Orso. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 308–318, Piscataway, NJ, USA, 2017. IEEE Press. 7

[41] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017. 5

[42] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European Conference on Information Retrieval*, pages 345–359. Springer, 2005. 5

[43] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92. IEEE, 2012. 7

[44] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 167–177. ACM, 2018. 1, 4.2.1, 7.1, 8

[45] Ville-Veikko Helppi. What Every App Developer Should Know About Android. http://www.smashingmagazine.com/2014/10/02/what-every-app-developer-should-know-about-android/, October 2014. 1

[46] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. 2018. 1, 4.1, 4.1, 4.2.1, 4.2.2, 4.3, 5.1, 7.1, 7, 8

[47] Apple breaks new iphones with terrible software update. http://www.slate.com/blogs/future_tense/2014/09/24/apple_ios_8_0_1_software_update_major_bugs_hit_iphone_6_6_plus.html, 2014. 1

[48] Maxime Lamothe and Weiyi Shang. Exploring the use of automated api migrating techniques in practice: An experience report on android. 2018. 7

[49] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Felix Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. Characterizing smartphone usage patterns from millions of android users. In *Proceedings of the 2015 Internet Measurement Conference*, pages 459–472. ACM, 2015. 7

[50] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 411–422. IEEE, 2016. 8

[51] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 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 2018, pages 153–163, New York, NY, USA, 2018. ACM. 1, 4.1, 4.2.2, 4.3, 5.1, 5.1, 7.1, 7

[52] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264. ACM, 2018. 7

[53] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013. 7

[54] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94. ACM, 2014. 1

[55] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014. 7

[56] Mehran Mahmoudi and Sarah Nadi. The android update problem: an empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 220–230. ACM, 2018. 7

[57] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013. 7

[58] Michael Kassner. Beware of danger lurking in Android phone updates. http://www.techrepublic.com/article/

beware-of-danger-lurking-in-android-phone-updates/", April 2014. 1, 2.1

[59] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 15–18. IEEE, 2017. 7

[60] Suhaib Mujahid. Detecting wearable app permission mismatches: a case study on android wear. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 1065–1067. ACM, 2017. 1

[61] Suhaib Mujahid and Emad Shihab Rabe Abdalkareem. Studying permission related issues in android wearable apps. 1

[62] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John Mitchell. Target fragmentation in android apps. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 204–213. IEEE, 2016. 7

[63] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2011. 7

[64] Mehul Rajput. Tips For Solving Your Android App Crashing Issues, 2015. http://tech.co/tips-solving-android-app-crashing-issues-2015-10. 1

[65] Mallisa Tolentino. Will These Bugs be Fixed in Android 5.1.1 Update. http://siliconangle.com/blog/2015/04/24/will-these-bugs-be-fixed-in-android-5-1-1-update/, April 2015. 1

[66] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-An, Gregg Rothermel, and Jackson Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the 39th International Conference on Software Engineering*, pages 324–334. IEEE Press, 2017. 3.1, 4

[67] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237. ACM, 2016. 1, 7, 7

[68] Daoyuan Wu, Ximing Liu, Jiayun Xu, David Lo, and Debin Gao. Measuring the declared sdk versions and their consistency with api calls in android apps. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 678–690. Springer, 2017. 1, 3.4, 7

[69] YouTube API change: some older devices can't update to new app. http://hexus.net/ce/news/audio-visual/82570-youtube-api-change-older-devices-update-new-app/, 2014. 1

[70] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 409–423. IEEE, 2014. 7