Scalable, Automated Incompatibility Detection for Android

ABSTRACT

With the ever-increasing popularity of mobile devices over the last decade, mobile applications and the frameworks upon which they are built frequently change, leading to a confusing jumble of devices and applications utilizing differing features even within the same framework. For Android apps and devices—the largest such framework and marketplace—mismatches between the version of the app API installed on a device and the version targeted by the developers of an app running on that device can lead to runtime crashes, providing a poor user experience. In this paper, we present SAINTDroid, a new analysis approach that automatically detects all types of mismatches to which an app may be vulnerable across versions of the Android API it declaratively supports. We applied SAINTDroid to 3,590 real-world apps and compared the results of our analysis against the state-of-the-art techniques, which corroborates that SAINTDroid is up to 76% more successful in detecting compatibility issues, while issuing significantly less (11-52%) false alarms. We also show that our approach, which gradually loads and analyzes classes as needed, remarkably outperforms the existing techniques in terms of scalability.

KEYWORDS

Android compatibility, program analysis, software evolution

ACM Reference Format:
. 2020. Scalable, Automated Incompatibility Detection for Android. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Android is the leading mobile operating system representing over 80% of the market share [4]. The meteoric rise of Android is largely due to its vibrant app market [12], which currently provisions nearly three million apps, with thousands added or 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 as long as they comply with the rules and constraints imposed by the framework. An ADF exposes well-defined application programming interfaces (APIs) that manifest the set of extension points for building the application-specific logic, setting it apart from traditional software systems often realized as a monolithic, independent piece of code.

The Android ADF frequently evolves, with hundreds of releases from multiple device vendors since 2010 [5]. Such rapid evolution leads to incompatibilities in Android apps targeted to older versions of the framework. As a result, defects and vulnerabilities, especially following ADF updates, continue to plague the dependability and security of Android devices and apps [37, 40]. A recent study shows that 23% of Android apps behave differently after a framework update, and around 50% of the Android updates have caused instability in previously working apps and systems [30]. This has been referred to as "death on update" [21, 24, 26, 32, 42, 48]. As part of Android ADF version 6.0 (API-level 23), Google introduced a dynamic permission system. Previously, the permission system was entirely static, with the user granting all requested dangerous permissions at install time. The new permission system instead allows users to grant or revoke permissions at run-time [2]. This new system creates a new class of permissions-related incompatibility issues.

Recent research efforts have studied compatibility issues [31, 46, 47], but existing detection techniques target only certain types of APIs. For example, Huang et al. [31] only targets API callback related to lifecycles; identifying them requires significant manual labor [31] and thorough inspection of incomplete documentation [47]. Furthermore, none of the state-of-the-art techniques consider incompatibilities due to the dynamic permission system. The state-of-the-art compatibility detection techniques also suffer from acknowledged frequent "false alarms" because of the coarse granularity at which they capture API information. The lack of proper support for detecting compatibility issues can increase the time needed to address such issues, often longer than six months [43]. Finally, these techniques [29, 31] have been shown facing difficulties in handling large scale libraries, due to direct loading of the entire code base for analysis purposes.

In this paper, we present a **s**calable, **a**utomated **i**ncompatibility **n**o**t**ifier for Android, dubbed SAINTDroid, which automatically detects all types of API- and permission-induced mismatches. Existing state-of-the-art compatibility detection techniques require to either analyze the entire ADF codebase or manually model common compatibility callbacks of ADF classes, prior to detecting incompatibilities [29, 31, 34]; as such they face serious scalability issues that limit their abilities to detect complex types of incompatibilities. Different from all these techniques, SAINTDroid overcomes such scalability issues by gradually loading and analyzing classes, wherein a reachability analysis is leveraged to stumble on all pertinent classes.

SAINTDroid has several advantages over existing work. First, unlike prior techniques that focus on specific types of APIs, our approach has the potential to greatly increase the scope of analysis by automatically and effectively analyzing all of the APIs in an ADF version. Second, SAINTDroid also analyzes platform code to uncover compatibility issues that exist within the framework; these issues cannot be detected by analyzing only the application code—an approach commonly taken by the state-of-the-art incompatibility detectors [15, 31, 34]. TY: (We need to revise the above sentence — ICSE reviewer 1 says CiD does not analyze only the application code.) Third, incrementally loading and analyzing classes allows

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

our technique to be remarkably faster and more scalable than the state-of-the-art in compatibility detection.

Our evaluation of SAINTDroid against the state-of-the-art analysis techniques using thousands of real-world Android apps indicates that SAINTDroid is up to 76% more successful in detecting compatibility issues, while issuing significantly less (11-52%) false alarms. It also successfully detects permission-induced mismatches that cannot be even detected by state-of-the-art techniques. In addition, SAINTDroid is up to 8.3 times (four times on average) faster than the state-of-the art techniques.

To summarize, this paper makes the following contributions:

- *General API and permission-induced incompatibility detection algorithms*: We introduce novel algorithms that automatically detect all types of API incompatibilities and misuses of runtime permission APIs to which an app may be vulnerable across ADF versions.
- *Scalable incompatibility detection approach*: We introduce a scalable analysis approach that can incrementally load and analyze classes to handle large scale libraries in detecting incompatibility issues.
- *Publicly available tool implementation*: We develop a fully automated technology, SAINTDroid, that effectively realizes our compatibility detection approach. We make SAINTDroid publicly available to the research and education community [8].
- *Experiments*: We present results from experiments run on 3,590 real-world apps and benchmark apps, corroborating SAINTDroid's ability in (1) effective compatibility analysis of Android apps, reporting many issues undetected by the state-of-the-art analysis techniques; and (2) outperforming other tools in terms of scalability.

The rest of this paper is organized as follows. Section 2 illustrates various examples of Android compatibility issues. Section 3 provides an overview of SAINTDroid to effectively detect compatibility issues. Section 4 describes our empirical study, and Section 5 reports the results. Finally, the paper concludes with a discussion of current limitations, and an outline of the related research and future work.

2 API AND PERMISSION-INDUCED COMPATIBILITY ISSUES

To motivate the research and demonstrate the need for mechanisms for incompatibility detection, this section describes three types of Android compatibility issues, which can lead to runtime app crashes. Table 1 summarizes API- and permission-induced compatibility issues. We will later show how SAINTDroid helps effectively identify these incompatibility scenarios.

Table 1: Three Types of Compatibility Issues in Android.

Mismatch type	Abbr.	Compatibility	App level	Device level	Results in mismatch if...
API invocation *(App ⟶ Android)*	API	Backward Forward	$\geq \alpha$ $< \alpha$	$< \alpha$ $\geq \alpha$	app invokes API method introduced/updated in α
API callback *(Android ⟶ App)*	APC	Backward Forward	$\geq \alpha$ $< \alpha$	$< \alpha$ $\geq \alpha$	app overrides API callback introduced/updated in α
Permission-induced	PRM	Forward	≥ 23 < 23	≥ 23 ≥ 23	app misuses runtime permission checking

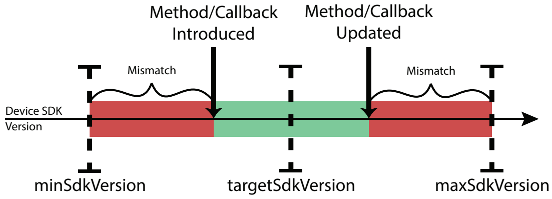

Figure 1: Mismatch between app and device API levels

2.1 Android API Background

As of August 2018, there are 26 releases of the Android API, most recently API level 28 [25]. Each version contains new and updated methods to help developers improve app performance, security, and user-experience. 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) [1]. Google strongly recommends that developers specify the range of the API levels their apps can support in the manifest or Gradle file by setting three specific attributes: (1) `minSdkVersion`, which specifies the lowest level of the API supported by the app; (2) `targetSdkVersion`, which specifies the level of the API used during app development; and (3) `maxSdkVersion`, which specifies the highest level of the API supported by the app.[1]

2.2 API Compatibility Issues

Incompatible API levels can cause runtime 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 *backward-compatibility*, where an app targeting a higher API level is installed on a device running a lower one [19]. However, despite Google's assurances, there may also be issues with *forward-compatibility* when an app is run on a device with a higher API level than the app's target. 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, a mismatch arises (the two red regions as shown in Figure 1), which could potentially crash the app. TY: (Need to clarify the overriding a callback will cause incompatibility, raised by ICSE reviewer1.)

We divide these API incompatibilities into two types (Table 1): *invocation mismatches*, where an app attempts to invoke an API method not supported by the device; and *callback mismatches*, where an app implements a callback method missing from the API level installed on the device, which will never be invoked.

2.2.1 API invocation mismatch. Mismatches in API method invocation occur when an app developed against a higher version of the API attempts to call a method introduced between its target version and that installed on the device; the app crashes when the system cannot find the desired method. Similarly, 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. The former is an instance of a backward-compatibility issue, while the latter touches forward-compatibility, as referenced in Table 1.

[1] According to the Google documentation, declaring this attribute is not recommended [19] but installing an older app on a newer device may still lead to unexpected behavior [40].

Listing 1 provides an illustrative example, where the app targets Android API level 28, but its `minSdkVersion` is set to 21. In case the app is installed on a device with the Android API level 21—the API level supported by the app according to its specified `minSdkVersion`—it will crash on the invocation of getColorStateList (lines 9-10), which was introduced in API level 23. One way to safeguard against this mismatch is to check the device's API level at runtime, as shown in the comment on line 8. This prevents the app from executing the call on versions where it might be missing, but it is not fool-proof; developers could easily forget to add or modify the check when updating an app, leaving the code vulnerable to a mismatch.

2.2.2 API callback mismatch. API callback compatibility issues initiate in the Android system when its invokes callback methods overridden in the app. Listing 2 shows a snippet adapted from the *Simple Solitaire* [18] app, where the API callback onAttach(Context), which is introduced in API level 23, is overridden. 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 runtime crashes. In the case where a callback is added to the API, this mismatch is a backward-compatibility issue; if a callback is removed, it is a problem with forward-compatibility.

2.3 Permission-induced Compatibility Issues

With the release of Android API level 23 (Android 6), 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 [2]. 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 runtime. In total, Android classifies 26 permissions as dangerous [28]. The goal of the new runtime permission system is to encourage developers to help users understand why an application requires the requested dangerous permission [6].

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

Listing 1: API Invocation Mismatch

```
1  public class CustomPreferenceFragment
2    extends PreferenceFragment {
3
4    @Override
5    public void onAttach(Context context) {
6      reinitializeData(context);
7      super.onAttach(context);
8    }
9  }
```

Listing 2: API Callback Mismatch

Permission-induced incompatibility can also be divided into two types of mismatch: *permission request mismatches*, where an app targeting API level 23 or higher does not implement the new runtime 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 runtime.

Listing 3 illustrates a permission request mismatch; the app may crash on line 12 where it attempts to use a dangerous permission it did not request. To prevent the mismatch, the app would need to check the API version and request permissions at runtime (shown as comments on lines 7-9) and implement onRequestPermissionsResult (line 16). More detailed examples of the new runtime permissions system can be seen in the Android documentation [6].

If a user installs an app targeting APIs lower than 23 on a device running API 23 or above, the user must accept all dangerous permissions requested by the app at install time, or the app will not be installed. However, API 23, i.e., Android version 6.0, allows the user 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–a permissions revocation mismatch. This behavior has been recurrently reported in real-world apps. *AdAway* [3], for example, tries to access external storage (such as an SD card) at runtime. If that permission is revoked, the app crashes when its tries to load data from the storage mechanism.

2.4 Limitations of Existing Work

For existing incompatibility detection techniques to be able to identify API and APC issues, they need to be able to analyze both the application code and ADF code. Because the ADF code can be very large, analyzing it would require a significant amount of memory resource that may not be feasible. Thus, a state-of-the-art technique called Cider TY: (need citiation) conserves memory by creating models from the underlying ADF to represent API invocations and their callback counterparts [31]. As previously reported, creating

```
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 3: Permissions Mismatch (tgt ≥ 23)

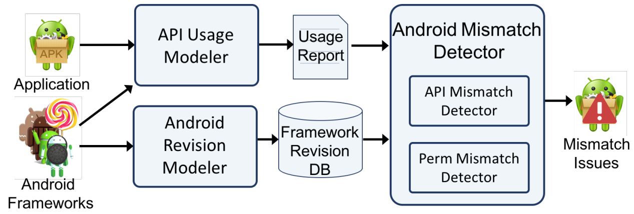

Figure 2: Architecture of SAINTDroid

models can be a daunting task that may not be able to keep up with rapid releases of ADFs [45]. As will be shown later, Cider can miss detecting compatibility issues that exist in different ADFs. Moreover, it is only capable of detecting compatibility types that have been modeled.

Another state-of-the-art incompatibility detector is CiD. TY: (add citation). To conserve memory, CiD extracts specific API code from the ADF, and then appends it to the call graph of the application code. However, to reduce memory usage, it only analyzes the initial API call and does not analyze subsequent calls within the ADF [34]. As will be shown later, this approach misses incompatibility issues that exist deeper into the ADF code.

Compared to the existing work, our approach, SAINTDroid, can achieve higher effectiveness through a more effective memory conservation. Specifically, SAINTDroid uses reachability analysis to incrementally identify pertinent code and then analyze it, along with the application code, to detect compatibility issues. Thus, our approach focuses its analysis effort on relevant portions of the ADF and not the entire ADF. For example, if an application makes an API call, SAINTdroid identifies the class to which the invoked API belongs, then loads just that class, and continues its analysis into that class. SAINTDroid analysis can seamlessly move between the application code and the ADF code during the compatibility analysis. In the next section, we describe the details of SAINTDroid.

3 APPROACH

This section overviews our approach to automatically detect all three types of API- and permission-induced mismatches described in Table 1. As depicted in Figure 2, SAINTDroid takes as input an app APK[2] along with a set of Android framework versions, and produces a list of mismatches for the given Android app. SAINTDroid comprises three main components: (1) The *API usage modeler (AUM)* that utilizes different static analysis techniques, i.e., control flow and data flow analyses, to identify the API call sites and the concomitant conditions thereof; (2) The *Android revision modeler (ARM)* that extracts essential information about the framework APIs' lifetime and mappings between Android API calls and the permissions required to perform those calls from the Android framework revision history; (3) The *Android mismatch detector (AMD)* that leverages the artifacts produced by AUM and ARM components to effectively detect API- and permission-related mismatches in the app under analysis.

In the rest of this section, we describe the details of each component in turn.

3.1 AUM: API Usage Modeler

The AUM module performs path sensitive, inter-procedural data flow analysis on call and data flow graphs of a given decompiled APK file to determine references to API methods or callbacks. More precisely, AUM derives an inter-procedural control-flow graph, augmented to account for implicit invocations (e.g., callbacks). The produced inter-procedural control-flow graph is further annotated with permissions required to enact Android API calls. Finally, a reachability analysis is conducted over the augmented graph to identify the guards that encompass the execution paths reaching the annotated API calls or permission-required functionalities.

SAINTDroid's compatibility analysis has three major advantages over the state-of-the-art approaches. First, to render our analysis more *effective, efficient, and scalable*, SAINTDroid's AUM module employs an approach that incrementally discovers pertinent classes via reachability analysis [44]. It then analyzes only these classes. This, in turn, allows our technique to be significantly faster and more space efficient than the state-of-the-art in compatibility analysis, without sacrificing its capability in detecting compatibility issues, as evidenced by the experimental results (cf. Section 4). Some existing state-of-the-art techniques [29, 31], on the other hand, directly load the entire code base into memory, and thereby facing difficulties in handling large scale libraries such as ADF.

Second, the AUM module analyzes actual ADF code to detect *more instances and types of compatibility issues*. Prior work, on the other hand, focuses on creating models of the ADF to only identify API callback compatibility issues [31]. Third, while prior work by and large focuses on the first level framework API calls, i.e., the first call to the framework from an app [34], the AUM module analyzes method calls beyond that initial level, which, once again, empowers SAINTDroid to detect *more instances and types of incompatibility issues.*

Another key feature in Android that can affect the accuracy of the compatibility analysis is late binding. Indeed, apps may dynamically load code that is not included in the main dex file[3] of the original application package, initially loaded at the installation time. This mechanism enables an app to be extended with new desirable features at run-time. However, in spite of its virtue, it poses challenges to analysis techniques for assessing compatibility of Android apps.

To avoid missing any potential compatibility-related issues that may result in crashes at run-time, SAINTDroid takes a conservative approach and considers all possible bindings that can be statically discovered. More precisely, SAINTDroid's AUM component examines not only the main app code loaded at the installation time, but also any other code accessible from the app package that can be dynamically loaded at run-time. AUM incrementally augments the control-flow and data-flow graphs by recursively identifying and examining such to-be dynamically loaded classes to ensure that every method in every such classes is analyzed. Note that such to-be dynamically loaded code may not always be statically analyzable, especially when it is not bundled within installed packages, and rather externally loaded from remote servers.

[2] APK is an app bytecode package used to distribute and install an Android application.

[3] A dex file is an executable file that incorporates compiled code written for Android.

Algorithm 1 Detecting 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
```

3.2 ARM: Android Revision Modeler

The ARM module derives both the API lifecycle and the permission mapping models through mining of the Android framework revision history. It first constructs an API database containing all public APIs defined in Android API levels 2 through 28[4], allowing SAINTDroid to determine which methods and callbacks exist in each level within the app's supported range. SAINTDroid automatically mines Android framework versions and stores the captured API information in a format that can be effectively queried by both the AMD module to generate the list of APIs in each level and a method call graph for each API method. Note that the API database is constructed once for a given framework, i.e., an Android API level, as a reusable model upon which the compatibility analysis of all apps relies. The Android revision modeling component to derive the lifetime of each framework's API is realized in an entirely automated fashion, which in turn facilitates supporting the upcoming versions of the framework. Note that ARM differs from the API Lifecyle Modeling (ALM) component of CiD in the following ways: TY: (We should explicitly list the differences if there are any.)

SAINTDroid next extends the database with mappings between Android API methods and the permissions required by the Android framework during the execution of those methods. To achieve this, ARM relies in part on PScout [22], one of the most comprehensive permission maps available for the Android framework. The issue in using PScout was that its latest mapping was provided for the Android API level 22. ARM, therefore, extended the latest official release of PScout to include new mappings that would reflect the more up to date Android API levels. Similar to the Android API database, permission maps are constructed once and reused in the subsequent analyses.

3.3 AMD: Android Mismatch Detector

The AMD module analyzes the artifacts produced by the other modules shown in Figure 2 to identify both API-related mismatches (*API Mismatch Detector*) and permissions-related mismatches (*Permissions Mismatch Detector*). The *Mismatch Detection* component first checks for API compatibility issues (cf. Section 2.2), using the following process to spot both API invocation and callback mismatches:

Algorithm 2 Detecting 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
```

Invocation mismatch: The detector uses Algorithm 1 to detect API invocation mismatches in each block of each method from the data flow graph generated by static analysis of the app. If the current block represents 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 API method (line 4), query the API database at each supported level to determine whether the method called in the current block is defined (line 5-6). In case that it is not defined, add the current block to the set of mismatches (line 7). In the case that an Android API is invoked inside a method call, our algorithm (line 8) also checks if there is an invocation to a user's defined method (i.e., not an invocation to an Android API). If this is the case, our algorithm also analyzes the callee method to look for Android API invocations (line 9). Finally, we reset the minimum and maximum supported API levels to those defined in the app's manifest at the end of each guard condition (lines 10-11). SAINTDroid 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.

Callback mismatch: The detector uses Algorithm 2 to detect API callback mismatches in each method within the call graph derived from the app under analysis. If the method overrides an API callback (line 2), iterate over the API levels that the app declares to support and query the API database—automatically generated by the Database Construction component—to determine whether the callback is defined within the entire range of supported API levels (lines 4-5). This sets our approach apart from prior research, such as Cider [47], through *automatically* detecting incompatible API callbacks without requiring any manual effort of compiling a list of candidate callbacks beforehand, thereby making it widely applicable and practical for use by many.

The second part of the *Mismatch Detection* component detects incompatibilities surrounding the new runtime permissions system introduced in API level 23, a capability unique 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, retrieve the call graph from the *API Usage Extraction* component (line 5), and check whether each method of the app that

[4] The Android frameworks range from API level 2 through API level 28, collected using *sdkmanager*, shipped with the Android SDK Tools to manage packages for the Android SDK [17].

Algorithm 3 Detecting 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:     mismatches ← ∅
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
```

targets API level 23 or newer overrides onRequestPermissionsResult (lines 6-8). In case the app does implement the new runtime permission system, there is again no risk of mismatch (line 9). If the app either does *not* implement the new runtime system *or* targets an API level earlier than 23, each usage of a dangerous permissions could result in a mismatch and crash. To detect dangerous permission usages, iterate 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).

4 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of SAINTDroid. We have implemented SAINTDroid's static analysis capability on top of the Jitana framework [44], which is hybrid analysis tool for Android. We also used APKTool [7] for extracting apps' manifest files. As a result, our approach implementation only requires the availability of Android executable files, and not the original source code. SAINTDroid, thus, 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. SAINTDroid's tool and the experimental data are available at the project website [8].

We used the SAINTDroid apparatus for carrying out the experiments. In our evaluation, we address the following research questions:

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

RQ2. *Applicability:* How well does SAINTDroid perform in practice? Can it find compatibility issues in real-world applications?

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

4.1 Objects of Analysis

Our experimental objects are a set of Android apps drawn from different sources.

To evaluate the accuracy of our analysis technique and compare it against the other compatibility analysis tools, we used two suites of benchmark apps, CiD-Bench [34] and Cider-Bench [31], developed independently by other research groups. CiD-Bench contains seven benchmark apps and Cider-Bench contains 20 apps, for which all compatibility issues are known in advance–establishing a ground truth. TY: (The ICSE reviewer 1 noted that Cider did not evaluate recall, so he/she does not think Cider-Bench has ground truths. We should clarify.) 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. One of our baseline system, i.e. Lint, requires building the apps to perform the compatibility analysis. Out of the 27 benchmark apps, eight apps cannot be built[5]; therefore, they are excluded from the analysis, leaving the total of 19 apps used in our comparative study. Using the same benchmark apps as prior research allows us to compare our results against them and help eradicate internal threats to the validity of our results.

To evaluate the implications of our tool in practice, we collected over 3,000 real-world Android apps from the following two sources: (1) FDroid [9] 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. (2) We also include 2,300 apps from AndroZoo [20], a growing repository of Android apps collected from various sources, including the official Google Play store. 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

Independent Variables. Our independent variables involve baseline techniques used in our study to perform the analysis of compatibility issues. These techniques include CiD [34], Cider [31], and Lint [15].

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 [31] 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 [29] as a possible baseline technique. IctApiFinder was introduced at about the same time as Cider. Unfortunately, the tool is not publicly available and our

[5] The benchmark apps were built using Gradle [13], which dropped support of some Android SDK tool chains. Even with the appropriate SDKs in place on two different systems, Gradle were unable to build the apps.

Table 2: Comparison between SAINTDroid, CiD, Cider, and Lint. TP, FP and FN are represented by symbols ☑, ☒, □, respectively. X(#) indicates the number # of detected instances for the corresponding symbol X.

		SAINTDroid		CiD+Cider		Lint	
	App	API	APC	API	APC	API	APC
Cider-Bench	AFWall+	(☑9)	(☑7)	(□9)	☑(□6)	☑(□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)		☒(□4)
CiD-Bench	Basic	☑		☑		□	
	Forward	☑		☑		□	
	GenericType	☑		☑		□	
	Inheritance	(☑2)		(☑2)		(□2)	
	Protection						
	Protection2					□	
	Varargs	(☑2)		(☑2)		(□2)	
Precision: **Recall:** **F-Measure:**		**79%** **93%** **85%**	**100%** **95%** **98%**	**27%** **59%** **42%**	**89%** **19%** **31%**	**100%** **2%** **4%**	**0%** **0%** **0%**

attempts to contact the authors to request access were unsuccessful. Therefore, we did not use it in our study.

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 [31, 34]. 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., SAINTDroid, CiD, and Lint.

4.3 Study Operation

We conduct our experiments on a MacBook Pro running High Sierra 10.13.3 with an Intel Core i5 2.5 GHz CPU processor and 8 GB of main memory. To answer RQ1 and RQ2, we run each analysis once since the techniques are based on static analysis. To handle uncontrollable factors in our experiments addressing RQ3 (performance), we repeat the experiments three times and measure the amount of time required to perform the analysis of each app using the analysis techniques, each averaged over three attempts. Further, since Lint needs to build the app before it can perform the analysis, we perform four consecutive analysis attempts with Lint, and report the average analysis time of the last three analyses.

4.4 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 [31, 34] 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 SAINTDroid and the infrastructure used to run CiD and SAINTDroid. 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 SAINTDroid, but do not yet assess whether the approach helps software engineers or analysts addresses dependability and security concerns more quickly than current approaches.

5 RESULTS AND ANALYSIS

This section reports the data we collected to empirically address the three research questions, their interpretations, and our results.

5.1 RQ1: Accuracy

Table 2 summarizes the results of our experiments for evaluating the accuracy of SAINTDroid in detecting compatibility issues compared to the other state-of-the-art and state-of-the-practice techniques. For each app under analysis, we report the number of true (☑) and false (☒) positives and false negatives (□) according to the three categories of compatibility issues:

API Invocation Compatibility Issues (API). SAINTDroid succeeds in detecting all 8 known API compatibility issues in CiD-Bench suite, and 33 API compatibility issues out of 36 in Cider-Bench suite. It also correctly ignores 32 cases in `FOSS Browser`, `Padland`, `DuckDuckGo`, `SurvivalManual` and `Uber ride` apps, where there are no API compatibility issues yet wrongly reported by CiD. The only missed issues are the ones that occur in the `MaterialFBook` app's anonymous classes, not handled by our model extractor, discussed in more detail in Section 6. CiD detects fewer (26 out of 44) API invocation compatibility issues, and it has a high rate of false positives, the majority of which arise because CiD's analysis is not context-sensitive and does not track guard conditions across function calls. Lint does even worse and only identified one of the verified mismatches. According to the results, Cider is unable to examine Android apps for API invocation compatibility issues. The results show that SAINTDroid outperforms the other three techniques in terms of both precision and recall.

API Callback Compatibility Issues (APC). SAINTDroid successfully detects 40 callback compatibility issues out of 42 in the objects of analysis, with no false positives. Cider misses most of the issues identified by SAINTDroid mainly because it only considers the classes that were manually modeled, i.e., Activity, Fragment, Service, and WebView. Whereas, SAINTDroid automatically identifies potential callback mismatches across all classes in the Android API. As expected, CiD is unable to examine Android apps for callback compatibility issues. Lint not only identifies none of the verified mismatches, but also has a high rate of false warnings. Overall, the results show that SAINTDroid outperforms the other three techniques in terms of both precision and recall.

Permission-induced Compatibility Issues (PRM). According to the experimental results, SAINTDroid detects two cases of permission-induced compatibility issues in *FOSS Browser* [11] and *Kolab notes* [14] apps; these two apps request dangerous permissions and target an API level higher than 23, yet they do not follow the new runtime permission checking. Note that the other mismatch detection techniques do not detect any of the runtime permission compatibility issues. Next, we evaluate whether SAINTDroid is sufficiently robust to handle large real-world apps.

5.2 RQ2: Real-World Applicability

To evaluate the implications of our tool in practice, we applied SAINTDroid to real-world apps collected from a variety of repositories (cf. Section 4). SAINTDroid 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 group (i) attempt to use dangerous permissions without implementing the runtime permissions request system, and 1,206 apps (68.68%) in group (ii) are vulnerable to permissions revocation mismatches (cf. Section 2.3).

Since manually examining all 3,691 real-world apps prohibitively expensive, we sampled 60 apps where incompatibilities were detected and calculated the precision scores. We do not consider recall because the ground-truth incompatibilities are unknown. Among all 60 incompatibility issues, the precision scores for API invocation, callback, and permission incompatibilities are 85%, 100%, and 100%, respectively. The results are consistent with the ones obtained from the benchmark programs (cf. Table 2).

We then investigated the SAINTDroid'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 [16], 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. Yet, 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 [10] 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 interface. Setting the minSdkVersion to 21 would resolve the mismatch.

Permission request mismatch. Kolab Notes [14] 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 [3] 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.

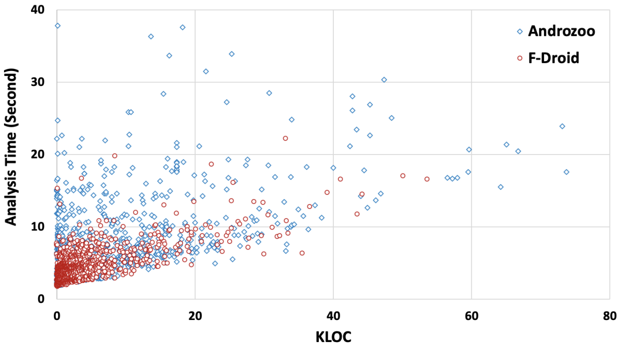

Figure 3: Scatter plot representing analysis time for compatibility checking of Android apps using SAINTDroid.

Table 3: Experiments performance statistics in seconds.

	App	SAINT Droid	CiD	Lint
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	–	35.1
	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
	Average	5.7	22.9	17.1

5.3 RQ3: Performance

Table 3 shows the analysis times of SAINTDroid, CiD, and Lint (in seconds). Dashes indicate that either the corresponding technique fails to produce analysis results after 600 seconds or crashes. As shown, the analysis time taken by SAINTDroid is significantly lower than those of CiD and Lint for almost all the apps. Also note that CiD fails to completely analyze four apps. The average analysis time taken by SAINTDroid, CiD, and Lint per app is 5.7, 22.9 and 17.1 seconds respectively, corroborating that SAINTDroid can efficiently vet Android apps for compatibility issues in a fraction of time taken by the other state-of-the-art tools.

Figure 3 presents the time taken by SAINTDroid to perform compatibility analysis on real-world apps. The scatter plot depicts both the analysis time and the app size. The experimental results show that the average analysis time taken by SAINTDroid, CiD, and Lint per app on real-world data sets is 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 have found outliers during the analysis. For example, the app in the top left corner in Figure 3 is a game application which extensively uses third party libraries, which took a considerable amount of time for the analyzer to compute the data structures required for the compatibility 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 less library classes than the aforementioned app, implicating in less complex graphs to analyze. Overall, the timing results show that SAINTDroid is up to 8.3 times (4.7 times on average) faster than the state-of-the art techniques, and 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 SAINTDroid performs more efficient than the state-of-the-art approaches, we conducted further performance evaluation, comparing the amount of resources and analysis efforts required by each approach. Specifically, we monitored the memory footprint required by each approach for performing analysis. Figure 4 shows a comparison of how much memory SAINTDroid and CiD are using during the analysis of real-world apps. According to the results, SAINTDroid 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 in practice for compatibility analysis.

6 DISCUSSION

Like any approach relying on static analysis, SAINTDroid is subject to false positives. A promising avenue of future research is to complement SAINTDroid with dynamic analysis techniques. Essentially, it should be possible to utilize dynamic analysis techniques to automatically verify incompatibilities identified through our conservative, static analysis based, incompatibility detection technique, further alleviating the burden of manual analysis.

As explained in Section 5, the majority of the false alarms are due to a limitation in SAINTDroid regarding dynamically-generated classes (e.g., WebView$1) that correspond to anonymous inner class declarations. When analyzing the code of each app, SAINTDroid only explores the explicit classes defined in the app, ignoring the dynamically-generated classes. Thus, any callback or method defined inside an anonymous inner class is invisible to SAINTDroid and is not included in the analysis. We plan to address this limitation in the future by including the dynamically-generated class definitions as well.

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.

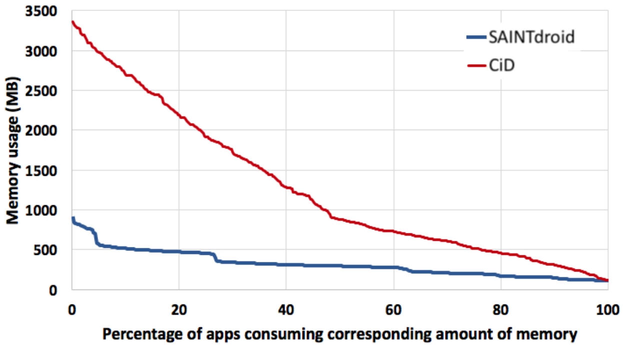

Figure 4: Amount of memory used by SAINTDroid and CiD when analyzing real-world Android apps.

Table 4: Comparing SAINTDroid to the state-of-the-art of compatibility detection techniques.

	API	APC	PRM
CiD [34]	✓	✗	✗
Cider [31]	✗	✓	✗
IctApiFinder [29]	✓	✗	✗
Lint [15]	✓	✗	✗
SAINTDroid	✓	✓	✓

API evolution. A large body of existing research focuses on the evolving nature of APIs, which is an important aspect of software maintenance [39], [23], [35], [33], [36], [27], [38], [41]. These research efforts explore the problems that are introduced by API changes. Among others, McDonnell et al. [39] studied Android's fast API evolution (115 API updates/month), and 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 [36]. Mutchler et al. [41] 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. [35] 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. [23] showed that applications with higher user ratings use APIs that are less change- and fault-prone compared to the applications with lower ratings.

These prior 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. SAINTDroid, on the other hand, is designed to be effective at detecting crash-leading API related issues.

API incompatibility. In Table 4, we compare the detection capabilities of SAINTDroid against the current state-of-the-art approaches. It is important to stress that SAINTDroid is the only solution that can automatically detect 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. [47] investigated side effects that may cause runtime crashes even within an app's supported API ranges, inspiring subsequent work. Huang et al. [31] 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. As such, 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 [47]. 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 [15] 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, SAINTDroid 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 SAINTDroid or Cider.

Li et al. [34] 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 upto the API level 25. In comparison, SAINTDroid offers automated extraction of the API database, and thereby supports up to the most recent Android platform (API level 28). Moreover, in contrast to SAINTDroid, CiD did not consider incompatibilities regarding the runtime permission system.

Wei et al. [46] 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 in an API level. Again, the result reported by their approach would be a subset of our detected issues.

8 CONCLUSION AND FUTURE WORK

This paper presents SAINTDroid, a novel approach and accompanying tool-suite for efficient analysis of various types of crash-leading Android compatibility issues. The experimental results of comparing SAINTDroid with the state-of-the-art in Android incompatibility detection corroborate its ability to efficiently detect more sources of potential issues, yielding fewer false positives and executing in a fraction of the time needed by the other techniques. Applying SAINTDroid to thousands of real-world apps from various repositories reveals that as many as 42% of the analyzed apps are prone to API invocation mismatch, 20% can crash due to API callback mismatch, and 40% of the apps can suffer from crashes due to permissions-related mismatch, indicating that such problems are still prolific in contemporary, real-world Android apps.

Besides what we already discussed in Section 6, we also plan as part of our future work to explore the trade-off between increasing analysis precision, e.g., through incorporating additional information such as CCFG for the compatibility analysis, and higher analysis overhead. In addition, we plan to develop a complementing code synthesizer to help repair apps that do not properly handle detected mismatches.

REFERENCES

[1] 2018. Android Versions. https://en.wikipedia.org/wiki/Android_version_history.

[2] 2018. Permissions in Android. https://developer.android.com/guide/topics/permissions/overview#permission-groups.

[3] 2019. AdAway. https://github.com/AdAway/AdAway/releases/tag/v3.0.2.

[4] 2019. Android Market Share. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[5] 2019. Android platform frameworks base. $https://github.com/aosp - mirror/platform\_frameworks\_base/releases$.

[6] 2019. Android Runtime Permissions. https://source.android.com/devices/tech/\config/runtime_perms.

[7] 2019. APKTool. https://ibotpeaches.github.io/Apktool/.

[8] 2019. SAINTDroid Repository. https://sites.google.com/view/saintdroid/.

[9] 2019. F-Droid. https://f-droid.org/.

[10] 2019. FOSDEM Companion. https://github.com/cbeyls/fosdem-companion-android/releases/tag/1.5.0.

[11] 2019. FOSS Browser. https://github.com/scoute-dich/browser/commit/e08f5b6.

[12] 2019. Google Play Apps. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[13] 2019. Gradle Build Tool. https://gradle.org.

[14] 2019. Kolab notes. https://github.com/konradrenner/kolabnotes-android/commit/14ba3c3.

[15] 2019. lint. http://tools.android.com/tips/lint.

[16] 2019. Offline Calendar. https://github.com/PrivacyApps/offline-calendar/releases/tag/v1.8.

[17] 2019. sdkmanager. https://developer.android.com/studio/command-line/sdkmanager.

[18] 2019. SimpleSolitaire. https://github.com/TobiasBielefeld/Simple-Solitaire/commit/1483ee.

[19] 2019. Using SDK in Android Apps. https://developer.android.com/guide/topics/manifest/uses-sdk-element.

[20] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 468–471.

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

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

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

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

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

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

[27] Mattia Fazzini and Alessandro Orso. 2017. Automated Cross-platform Inconsistency Detection for Mobile Apps. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 308–318. http://dl.acm.org/citation.cfm?id=3155562.3155604

[28] Google. 2019. Permissions Overview. https://developer.android.com/guide/topics/permissions/overview#permission-groups.

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

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

[31] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for Android applications.. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.

[32] iOS 2014. 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.

[33] Maxime Lamothe and Weiyi Shang. 2018. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. (2018).

[34] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 153–163. https://doi.org/10.1145/3213846.3213857

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

[36] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. 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*. ACM, 477–487.

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

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

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

[40] Michael Kassner. 2014. Beware of danger lurking in Android phone updates. http://www.techrepublic.com/article/beware-of-danger-lurking-in-android-phone-updates/.

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

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

[43] Mallisa Tolentino. 2015. 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/.

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

[45] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Software Engineering Notes* 37, 6 (Nov. 2012), 1–5.

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

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

[48] youtube 2014. 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/.