SAINTDroid: Scalable, Automated Incompatibility Detection for Android

Bruno Silva*, Clay Stevens*, Niloofar Mansoor*, Witawas Srisa-An*, Tingting Yu†, Hamid Bagheri*
*School of Computing, University of Nebraska-Lincoln, USA
†Electrical Engineering & Computer Science, University of Cincinnati, USA
{bruno, clay.stevens, niloofar}@huskers.unl.edu, tingting.yu@uc.edu, {witawas, bagheri}@unl.edu

***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 run-time crashes, providing a poor user experience. This paper presents SAINTDroid, a holistic compatibility analysis approach that seamlessly examines both the application code and the framework code by gradually loading and analyzing classes as needed during the compatibility analysis to enable efficient and scalable identification of various types of crash-leading Android compatibility issues. We applied SAINTDroid to 3,590 real-world apps and compared the analysis results against the state-of-the-art techniques, which corroborates that SAINTDroid is up to 76% more successful in detecting compatibility issues while issuing significantly fewer false alarms. The experimental results also show that SAINTDroid is remarkably (up to 8.3 times and four times on average) faster than the state-of-the-art techniques.**

***Index Terms*—Android compatibility, program analysis, software evolution**

I. Introduction

Android is the leading mobile operating system representing over 80% of the market share [1]. The meteoric rise of Android is largely due to its vibrant app market [2], 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 [3]. 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 [4], [5]. 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 [6]. This has been referred to as "death on update" [7]–[12]. One major source of incompatibility after update came with Android ADF version 6.0 (API-level 23), when 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 allows users to grant/revoke permissions at run-time [13], which creates a new class of permissions-related incompatibility issues.

Recent research efforts have studied compatibility issues [14]–[16], but existing detection techniques target only certain types of APIs. For example, Huang et al. [14] only targets API callbacks related to app component lifecycles (e.g., loaded/unloaded); identifying them requires significant manual labor [14] and thorough inspection of incomplete documentation [16]. 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 [17]. Finally, these techniques [14], [18] have been shown to face 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 various type of API- and permission-induced mismatches by performing a scalable, context-sensitive static analysis of Android APKs. 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 [14], [18], [19]; 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 load and analyze all pertinent classes.

SAINTDROID has several advantages over existing work. First, SAINTDROID holistically analyzes application and

ADF in tandem by gradually loading and analyzing classes as needed during the compatibility analysis. SAINTDROID analysis, thus, can seamlessly move between the application code and the ADF code during the compatibility analysis. In contrast, prior techniques first analyze the ADF code separately and use the stored results of that complete analysis to resolve API usages [14], [19], [20]. Second, by actually analyzing app and ADF code, our approach has the potential to greatly increase the scope of analysis by automatically and effectively analyzing all code in the utilized APIs in an ADF version. Prior techniques only focus on specific types of APIs. Third, incrementally loading and analyzing classes allows 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 indicates it is up to 76% more successful in detecting compatibility issues among thousands of real-world apps, while issuing significantly fewer (11-52%) false alarms. It also successfully detects permission-induced mismatches that cannot be detected by state-of-the-art techniques. SAINTDROID is also 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 [21].[1]
- *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.

Section II illustrates various examples of Android compatibility issues. Section III provides an overview of SAINTDROID to effectively detect compatibility issues. Sections IV-V describe our empirical study and report the results. Finally, the paper concludes with a discussion of current limitations, and an outline of the related research and future work.

II. API / 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 I summarizes API- and permission-induced compatibility issues. We will later show how SAINTDROID helps identify these incompatibilities.

TABLE I: Three Types of Compatibility Issues in Android.

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

A. Android API Background

As of September 2019, there have been 27 releases of the Android API, most recently API level 29 [22]. 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) [23]. 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 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 API level supported by the app.[2]

B. 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 [24]. 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 (e.g.) crash the app or lead to an incorrect program state in the case of a missed callback.

We divide these API incompatibilities into two types (Table I): *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.

[1]The SAINTDROID tool is available for download at the project website, https://sites.google.com/view/saintdroid/

[2]According to the Google documentation, declaring this attribute is not recommended [24] but installing an older app on a newer device may still lead to unexpected behavior [5].

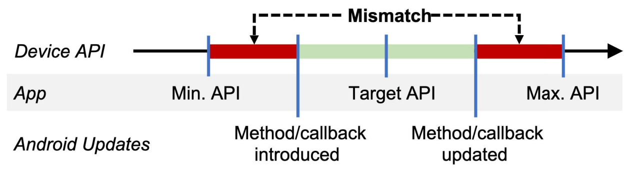

Fig. 1: Mismatch between app and device API levels, either before an invoked method/callback is introduced or after an invoked method/callback is modified by a version update.

```java
public class CustomPreferenceFragment
  extends PreferenceFragment {

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

Listing 2: API Callback Mismatch

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 I.

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) 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* [25] 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.

C. 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 [13]. 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 [26]. The goal of the new runtime permission system is to encourage developers to help users understand why an application requires the requested dangerous permission [27].

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

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

Listing 1: API Invocation Mismatch

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

Listing 3: Permissions Mismatch (tgt $\geq$ 23)

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 can be seen in the Android documentation [27].

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 from the app after installation, the app would crash while trying to use that permission—a permissions revocation mismatch. This behavior has been reported in real-world apps. *AdAway* [28], 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.

D. *Limitations of Existing Work*

One major shortcoming of the state-of-the-art approaches in detecting API compatibility issues is their inability to analyze application-under-analysis and the underlying ADF code in unison. Existing analysis techniques first load all code in the project and then perform analysis on the loaded code. This approach is both monolithic and memory intensive. Moreover, identifying compatibility issues requires analyzing both the application code and ADF in tandem. However, loading both the entire app and the ADF codebase is not feasible, due to high memory consumption. As such, the existing techniques analyze an app and the underlying ADF separately. For example, a state-of-the-art technique, called CIDER [14], conserves memory by using pre-defined models from the underlying ADF to represent API invocations and their callback counterparts. However, as previously reported, constructing ADF models is a daunting and error-prone task that may not be able to keep up with rapid releases of ADFs [29]. As will be shown in the evaluation section, 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 [19]. This approach creates a *conditional call graph* for each app to record method call information along with condition checkers related to the API level. The construction of this graph is done by analyzing the control flow of the app to identify API calls. From each API call, CID performs backward data-flow analysis to identify the presence of an API level check. Resolving API usage is then based on the list of API calls and information from the *conditional call graph*. To reduce memory usage, CID only analyzes the initial API call and does not analyze subsequent calls within the ADF [19]. As will be shown later, this approach misses incompatibility issues that exist deeper into the ADF code.

III. APPROACH

This section overviews our approach to automatically detect all three types of API- and permission-induced mismatches described in Table I. As depicted in Figure 2, SAINTDROID takes as input an app APK[3] 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 static analysis techniques, i.e., control flow and data flow analyses, to identify the API call sites and any conditional statements surrounding them; (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 the AUM and ARM components to effectively detect API- and permission-related mismatches in the app under analysis.

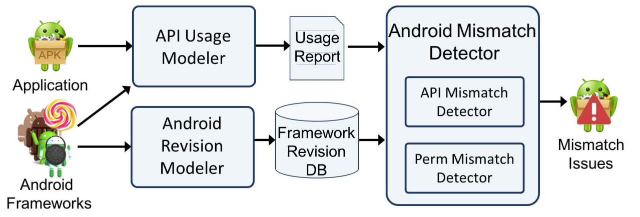

Fig. 2: Architecture of SAINTDROID

A. *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 (ICFG), augmented to account for implicit invocations (e.g., callbacks). The ICFG tracks the control flow within distinct invocations of Android Java processes; inter process communication (i.e., intents) are considered separate invocations starting from each message handler as the entry point. The produced ICFG 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.

Different from all the existing incompatibility detection techniques, SAINTDROID mimics the class-loading behavior of the Android Virtual Machine runtime to incrementally load classes and methods needed as part of the compatibility detection analysis. However, the underlying assumption of the existing program analysis techniques is that they operate in a closed-world fashion [30]. More specifically, existing static program analyzers (such as SOOT [31]) convert the code to an intermediate representation, e.g., *Jimple*. They then perform analyses to construct control-flow, data-flow, method-call, and points-to graphs. Any calls to external components (e.g., *ADF*,

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

Algorithm 1 Exploration of statically-analyzable Java classes

```
1:  procedure ExploreClasses(method)
    ▷ Input: method, APK
2:      class ← ClassLookup(method)
3:      loadedClass ← LoadClass(class)
4:      for each classMethod in loadedClass do
5:          GenerateControlFlow(classMethod)
6:          GenerateDataFlow(classMethod)
7:          AppendMCG(classMethod)
8:          if MethodNotInClass(classMethod) then
9:              AppendWorkList(classMethod)
10:     return true
```

native code) would be left as terminals in the method call graph and may not be analyzed.

SAINTDROID, unlike all the other incompatibility detectors, mimics the incremental loading behavior of the Android runtime during execution *while maintaining the completeness property of static analysis* by taking advantage of the way object-oriented languages organize classes to enable class loading at runtime. Algorithm 1 describes the exploration process. First, the algorithm uses a worklist that contains an initial list of methods to be explored, and loads classes to which they belong using a *Class Loader Virtual Machine* (*CLVM*). When a class is loaded, SAINTDROID statically analyzes its methods to build control- and data-flow graphs used to form a graph of method calls to other statically-discoverable classes; these method calls are appended to the worklist. Subsequently, those classes to which methods belong are loaded into the CLVM. SAINTDROID then processes the information extracted about each class by the CLVM to find the statements and calls needed to detect API mismatches (Section III-C). *An important point is that our novel class loader based exploration approach for compatibility detection blurs the boundaries between apps and libraries. Components are loaded as needed, and the method-call graph is generated as the analysis progresses.*

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 a novel class-loader (CLVM) based approach that incrementally discovers pertinent application and ADF classes via reachability analysis [32]. As a concrete example, if an application method calls an Android API, the class to which the API belongs would be loaded. By utilizing CLVM, SAINTDROID analyzes the application classes along with reachable ADF classes, rather than analyzing all ADF code. This analysis approach significantly reduces both peak memory footprint and memory consumption over time. Our technique is therefore 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 IV). Other state-of-the-art techniques [14], [18] directly load the entire code base into memory, and thereby face 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 focuses on creating models of the ADF to only identify API callback compatibility issues [14]. Third, while prior work focuses on the first level framework API calls, i.e., the first call to the framework from an app [19], the AUM analyzes method calls beyond that initial level, which 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[4] of the original application package, initially loaded at 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 classes that could be bound at run-time to references in the code, provided those classes can be statically discovered during analysis. 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 bound at run-time. The 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 loaded from a source outside the packages bundled in the APK.

B. 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[5], allowing SAINTDROID to determine which

[4]A dex file is an executable file containing compiled code for Android.

[5]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 [33].

Algorithm 2 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:                  mm ← mm ∪ {block}
8:      else if IsMethod(block) then
9:          mm ← mm ∪ FindApiIn(block, minLvl, maxLvl)
10:     else if IsGuardEnd(block) then
11:         (minLvl,maxLvl) ← (app.minSdk,app.maxSdk)
12:     return mm
```

Algorithm 3 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:                 mm ← mm ∪ {method}
6:     return mm
```

Algorithm 4 Detecting PRM mismatches

```
1: procedure DetectPermMismatch(app, graph, permMap)
   ▷ Input:  Decompiled APK, call/data flow graph, perm. map
   ▷ Output: List of detected mismatches
2:     dngrPerms ← GetDngrPermsFromManifest(app)
3:     if dngrPerms = ∅ then
4:         return ∅
5:     callGraph ← BuildCallGraph(app)
6:     if app.targetSdkVersion $\geq$ 23 then
7:         for each method in callGraph do
8:             if OverOnRequestPermsResult(method) then
9:                 return ∅
10:    mm ← ∅
11:    for each method in callGraph do
12:        dfg ← GetDataFlowGraph(graph, method)
13:        for each block in dfg do
14:            for each perm in dngrPerms do
15:                if permMap.IsUsingPerm(perm, block) then
16:                    mm ← mm ∪ {perm}
17:    return mm
```

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 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 ARM is realized in an entirely automated fashion, allowing support for future versions of the framework.

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, the ARM relies in part on PScout [34], one of the most comprehensive permission maps available for the Android framework, extended to include new mappings that would reflect more up to date Android API levels. Similar to the Android API database, permission maps are constructed once and reused in the subsequent analyses.

C. AMD: Android Mismatch Detector

The AMD analyzes the artifacts produced by the other modules shown in Figure 2 to identify both API- and permissions-related mismatches. The AMD checks for API compatibility issues (cf. Section II-B) using the following process:

Invocation mismatch: The detector uses Algorithm 2 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-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 accounts for guard conditions on the supported versions across methods, unlike other state-of-the-art techniques, such as LINT and CID.

Callback mismatch: The detector uses Algorithm 3 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 [16], through *automatically* detecting incompatible API callbacks without requiring any manual effort of compiling a list of candidate callbacks beforehand, thereby making it practical and widely applicable.

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 the algorithm, outlined in Algorithm 4, that checks permission-induced compatibility issues is as follows: First, extract dangerous permissions from the app's manifest (line 2). 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 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).

IV. Experimental Evaluation

For our evaluation, we developed a custom implementation of SAINTDroid using Jitana [32] to drive our static analysis. We also used APKTool [35] for extracting apps' manifest files. SAINTDroid's implementation only requires the availability of Android executable files, and not the original source code. SAINTDroid can be used by developers, end-users, and third-party reviewers to assess app compatibility. SAINTDroid's tool and experimental data are available [21]. We used our implementation to answer these 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?

A. Objects of Analysis

To evaluate the accuracy of our analysis technique and compare it against the other compatibility analysis tools, we used two suites of benchmark Android apps, CiD-Bench [19] and Cider-Bench [14], developed independently by other research groups. CiD-Bench contains seven benchmark apps and Cider-Bench contains 20 apps. The authors of these benchmarks also reported known vulnerabilities. We use these vulnerabilities as our evaluation baseline. For example, we evaluate the effectiveness of our approach by observing the number of reported vulnerabilities in these two benchmarks that SAINTDroid can detect. If our approach detects a new issue, we manually inspect whether the issue indeed exists. 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;[6] therefore, they are excluded from the analysis, leaving a total of 19 apps. Using the same benchmark apps as prior research allows us to compare our results against them and bolsters our internal validity.

To evaluate the implications of our tool in practice, we collected over 3,000 real-world Android apps: (1) 1,391 apps from FDroid [37], a software repository that contains free and open source Android apps; and (2) 2,300 apps from AndroZoo [38], 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 total apps.

[6]The benchmark apps were built using Gradle [36], 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.

B. 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 [19], Cider [14], and Lint [20].

CiD is a state-of-the-art approach for detecting Android compatibility issues. It has been publicly released, and we were 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 [14] 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 [18] as a possible baseline technique. 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.

Dependent Variables. To measure accuracy, we compare the number of detected compatibility issues with known issues as reported by prior work [14], [19]. 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.

C. Study Operation

We conducted 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 ran each analysis once since the techniques are based on static analysis. To handle uncontrollable factors in our experiments addressing RQ3 (performance), we repeated the experiments three times and measured 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 performed four consecutive analysis attempts with Lint, and report the average analysis time of the last three analyses.

V. Results and Analysis

A. RQ1: Accuracy

Table II 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

TABLE II: Comparison between SAINTDROID, CID, CIDER, and LINT. TP, FP and FN are represented by symbols ☑, ☒, ☐, respectively, along with the number detected.

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

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; CID wrongly reported compatibility issues in those cases due to its insensitivity to the context of each API call (i.e., it does not track guard conditions across function calls). These API invocation issues will most frequently lead to app crashes, as the user-defined code attempts to invoke API methods that are not defined for the device API level. 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 VI. 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. CIDER is unable to examine Android apps for API invocation compatibility issues. We interpret these results to show SAINTDROID provide better accuracy than the other three techniques.

API Callback Compatibility Issues (APC). SAINTDROID successfully detects 40 callback compatibility issues out of 42 in the objects of analysis, with no false positives. These compatibility issues cause a variety of undesirable results, from crashes to inaccurate app state, depending on the individual app; if the user-defined code does not implement an expected callback (or implements a now-deprecated callback), the app may miss necessary initialization or other event handling. 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. SAINTDROID automatically identifies potential callback mismatches across all classes in the Android API. 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* [39] and *Kolab notes* [40] 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 techniques do not detect any of the runtime permission compatibility issues.

B. RQ2: Real-World Applicability

To evaluate SAINTDROID in practice, we analyzed a set real-world apps from multiple repositories (cf. Section IV). 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 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 II-C).

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 II).

We then investigated the SAINTDROID's results and report some samples of our findings to illustrate the sorts of incompatibilities detected by SAINTDROID. 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. For permission mismatch in particular, we report one sample of each as all mismatches of those categories follow a nearly identical pattern; while each app may request/use a different permission, the structure of the mismatch will be the same. For mentioned apps available on the Google Play store, we report the number of downloads to provide an indication of each app's popularity. Neither F-Droid nor AndroZoo provide download or usage statistics.

API invocation mismatch. In the *Offline Calendar* app [41], 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* [42] is a conference companion app (10,000+ downloads). 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 API level 15, which does not support that callback method, and in turn may not properly propagate the new hotspot location to the view. 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* [40] is a note-taking app that synchronizes with other apps (1,000+ downloads). It exhibits a permission request mismatch. The app targets API 26 and uses the WRITE_EXTERNAL_STORAGE permission, but does not implement the methods to request the permission at runtime. If the permission is not granted when the user attempts to save/load data to/from an SD card, the action will fail. To resolve the mismatch, the developers must implement the runtime permissions request system.

Permission revocation mismatch. *AdAway* [28] 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.

TABLE III: Experiments performance statistics in seconds.

	App	SAINTDROID	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**

C. RQ3: Performance

Table III 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. For the smaller apps in the CID-Bench set, LINT required the least time for the analysis, as its analysis only examines direct calls to the API without considering the context or control flow. This translates to better analysis speed for small apps, but (as shown in Table II) reduces the accuracy of LINT's results. The performance benefits of SAINTDROID are more apparent for the larger apps in CIDER-Bench. Also note that CID fails to completely analyze four apps. 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

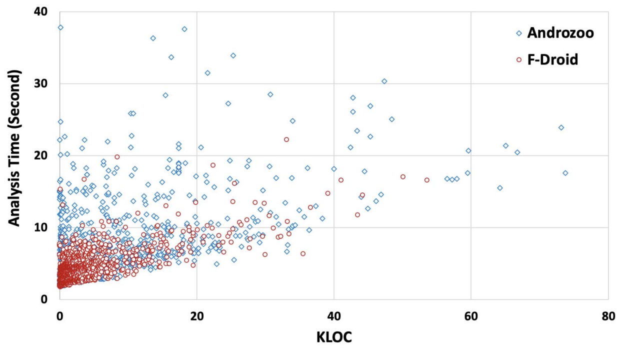

Fig. 3: Scatter plot representing analysis time for compatibility checking of Android apps using SAINTDROID.

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 the app's small KLOC. On the other hand, the app in the right side of the diagram, closer to 80 KLOC, loads only one third the library classes of the aforementioned app, providing simpler 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 efficiently than the state-of-the-art approaches, we conducted a further performance evaluation, comparing the amount of resources and analysis effort 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 use during the analysis of real-world apps. 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 (four times more memory) to perform the same analysis. We interpret this data as demonstrating SAINTDROID's effectiveness in practice.

D. Threats to Validity

The primary threat to external validity in this study involves the object programs. We have studied a smaller set of benchmark programs developed and released by prior research [14], [19] 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

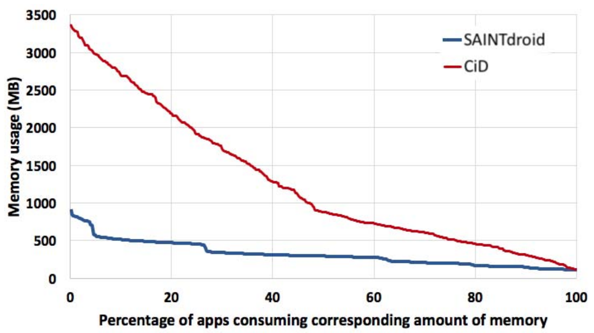

Fig. 4: Amount of memory used by SAINTDROID and CID when analyzing real-world Android apps.

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 compare our results with those previously reported to ensure correctness. The primary threat to construct validity is that we study efficiency measures relative to applications of SAINTDROID, but do not yet assess whether the approach helps software engineers or analysts address dependability and security concerns more quickly than current approaches.

VI. Discussion and Limitations

By performing the analysis incrementally, SAINTDROID's analysis cost is amortized over time. Start up time is also short as only a small portion of the code is needed to begin the analysis. Furthermore, the amount of memory that must be committed to store code and perform incremental analysis is small, substantially enhancing the scalability of our approach.

Like any approach relying on static analysis, SAINTDROID is subject to false positives. A promising avenue of future research is to complement SAINTDROID—which is a static analysis tool—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 V, 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 explores the classes explicitly defined in the app, as the dynamically-generated classes are unavailable prior to runtime. 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.

VII. Related Work

Android analysis has received a lot of attention since its inception [43]–[60]. This section discusses 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 important aspect of software maintenance [46], [48], [49], [50], [52], [54], [56], [57], [58], [60]. McDonnell et al. [46] studied Android's fast API evolution (115 API updates/month), and noticed developers' hesitation in embracing the fast-evolving APIs because they can be more defect-prone than other types of changes [54]. Bavota et al. [48] showed that applications with higher user ratings use APIs that are less change- and fault-prone compared to applications with lower ratings. Li et al. [50] investigated the frequency with which deprecated APIs are used in the development of Android apps, considering the deprecated APIs' annotations, documentation, and

TABLE IV: Comparing SAINTDROID to the state-of-the-art of compatibility detection techniques.

	API	APC	PRM
CID [19]	✓	✗	✗
CIDER [14]	✗	✓	✗
IctApiFinder [18]	✓	✗	✗
LINT [20]	✓	✗	✗
SAINTDROID	✓	✓	✓

removal consequences along with developers' reactions to APIs deprecations. These prior efforts clearly motivate the need to address issues relating to 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 IV, 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 API invocation compatibility issues (API), API callback compatibility issues (APC), and permission-induced compatibility issues (PRM). Wu et al. [16] investigated side effects that may cause runtime crashes even within an app's supported API ranges, inspiring subsequent work. Huang et al. [14] 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 only four API classes.CIDER thus provides no analysis of other classes nor of 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 [16]. 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 [20] 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—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 original Java source code rather than APKs and further requires the project to be built in the Android Studio to conduct the analysis. Unlike LINT, SAINTDROID operates directly on Dex code. While LINT claims to be able to detect API incompatibility issues, our experimental results indicate that LINT is not as effective as SAINTDROID.

Li et al. [19] provided an overview of the Android API evolution to identify cases where compatibility issues may arise in Android apps. They also presented CID, which (a) models the API lifecycle, (b) uses static analysis to detect APIs within the app's code, and (c) extracts API methods from the Android framework to detect backward incompatibilities. CID supports compatibility analysis up to API level 25. In comparison, SAINTDROID supports up to the most recent Android platform (API level 29). Moreover, in contrast to SAINTDROID, CID does not consider incompatibilities regarding the runtime permission system.

Wei et al. [15] 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. Conversely, Wang et al. [61] recently studied permissions-related issues in specific and developed a taxonomy of eleven classes of permissions-related issues based on the new runtime permission system. SAINTDROID focuses on incompatibility due to changes in the Android framework itself, which corresponds to Types 1-3 in their taxonomy. We also note that they categorize the *cause* of the issue, not its presentation within each app. SAINTDROID's automatic analysis would also catch issues that classified as being caused by the developer (Type 7-8).

VIII. 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 VI, 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. Another avenue for future work is to develop a complementing code synthesizer to help repair apps that do not properly handle detected mismatches.

References

[1] "Android market share," https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, 2019.

[2] "Google play apps," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2019.

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

[4] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. 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*. ACM, 2014, pp. 83–94.

[5] 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.

[6] V.-V. 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.

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

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

[9] 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/.

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

[11] "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.

[12] "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.

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

[14] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, "Understanding and detecting callback compatibility issues for android applications." in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 532–542.

[15] L. Wei, Y. Liu, and S.-C. 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*. ACM, 2016, pp. 226–237.

[16] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "Measuring the declared sdk versions and their consistency with api calls in android apps," in *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 2017, pp. 678–690.

[17] M. 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.

[18] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 167–177.

[19] L. Li, T. F. Bissyandé, H. Wang, and J. 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*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 153–163. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213857

[20] "lint," http://tools.android.com/tips/lint, 2019.

[21] "SAINTDROID repository," https://sites.google.com/view/saintdroid/, 2021.

[22] Google, "Android 10," https://developer.android.com/about/versions/10, 2019.

[23] "Android versions," https://en.wikipedia.org/wiki/Android version history, 2018.

[24] "Using sdk in android apps," https://developer.android.com/guide/topics/manifest/uses-sdk-element, 2019.

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

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

[27] "Android runtime permissions," https://source.android.com/devices/tech/config/runtime_perms, 2019.

[28] "AdAway," https://github.com/AdAway/AdAway/releases/tag/v3.0.2, 2019.

[29] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using Java pathfinder," *SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012.

[30] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the International Conference on Software Engineering*, May 2011, pp. 241–250.

[31] R. Vallée-Rai, "Soot: A Java Bytecode Optimization Framework," Master's thesis, McGill University, 2000.

[32] Y. Tsutano, S. Bachala, W. Srisa-An, G. Rothermel, and J. Dinh, "An efficient, robust, and scalable approach for analyzing interacting android apps," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 324–334.

[33] "sdkmanager," https://developer.android.com/studio/command-line/sdkmanager, 2019.

[34] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[35] "Apktool," https://ibotpeaches.github.io/Apktool/, 2019.

[36] "Gradle build tool," https://gradle.org, 2019.

[37] "F-Droid," https://f-droid.org/, 2019.

[38] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 468–471.

[39] "Foss Browser," https://github.com/scoute-dich/browser/commit/e08f5b6, 2019.

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

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

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

[43] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-an, and X. Luo, "DINA: detecting hidden android inter-app communication in dynamic loaded code," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 2782–2797, 2020. [Online]. Available: https://doi.org/10.1109/TIFS.2020.2976556

[44] M. Hammad, H. Bagheri, and S. Malek, "Deldroid: An automated approach for determination and enforcement of least-privilege architecture in android," *J. Syst. Softw.*, vol. 149, pp. 83–100, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2018.11.049

[45] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android roms via differential analysis." in *USENIX Security Symposium*, 2016, pp. 1153–1168.

[46] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 70–79.

[47] H. Bagheri, J. Wang, J. Aerts, and S. Malek, "Efficient, evolutionary security analysis of interacting android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018.* IEEE Computer Society, 2018, pp. 357–368. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00044

[48] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change-and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[49] S. Scalabrino, G. Bavota, M. Linares-Vasquez, M. Lanza, and R. Oliveto, "Data-driven solutions to detect api compatibility issues in android: an empirical study," in *Mining Software Repositories (MSR), 2019 IEEE/ACM 16th Working Conference on*. IEEE, 2019, pp. 288–298.

[50] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android apis," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 254–264.

[51] M. Hammad, H. Bagheri, and S. Malek, "Determination and enforcement of least-privilege architecture in android," in *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*. IEEE Computer Society, 2017, pp. 59–68. [Online]. Available: https://doi.org/10.1109/ICSA.2017.18

[52] M. Lamothe and W. Shang, "Exploring the use of automated api migrating techniques in practice: An experience report on android," 2018.

[53] B. R. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, and D. Garlan, "Architecture modeling and analysis of security in android systems," in *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. Tekinerdogan, U. Zdun, and M. A. Babar, Eds., vol. 9839, 2016, pp. 274–290. [Online]. Available: https://doi.org/10.1007/978-3-319-48992-6_21

[54] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. 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*. ACM, 2013, pp. 477–487.

[55] H. Bagheri, J. Wang, J. Aerts, N. Ghorbani, and S. Malek, "Flair: efficient analysis of android inter-component vulnerabilities in response to incremental changes," *Empir. Softw. Eng.*, vol. 26, no. 3, p. 54, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09932-6

[56] T. Luo, J. Wu, M. Yang, S. Zhao, Y. Wu, and Y. Wang, "Mad-api: Detection, correction and explanation of api misuses in distributed android applications," in *Proceedings of the 7th International Conference on Artificial Intelligence and Mobile Services*, 2018, pp. 123–140.

[57] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 308–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155604

[58] M. Mahmoudi and S. Nadi, "The android update problem: an empirical study," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 220–230.

[59] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-an, and X. Luo, "Detecting vulnerable android inter-app communication in dynamically loaded code," in *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*. IEEE, 2019, pp. 550–558. [Online]. Available: https://doi.org/10.1109/INFOCOM.2019.8737637

[60] P. Mutchler, Y. Safaei, A. Doupé, and J. Mitchell, "Target fragmentation in android apps," in *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2016, pp. 204–213.

[61] Y. Wang, Y. Wang, S. Wang, Y. Liu, C. Xu, S. Cheung, H. Yu, and Z.-l. Zhu, "Runtime permission issues in android apps: Taxonomy, practices, and ways forward," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.