Compatibility Issue Detection for Android Apps Based on Path-Sensitive Semantic Analysis

Sen Yang*‡, Sen Chen†‡, Lingling Fan§¶, Sihan Xu§, Zhanwei Hui‖, and Song Huang*¶
* Command and Control Engineering College, Army Engineering University of PLA, China
†College of Intelligence and Computing, Tianjin University, China
§College of Cyber Science, Nankai University, China
‖Academy of Military Science, China

***Abstract*—Android API-related compatibility issues have become a severe problem and significant challenge for app developers due to the well-known Android fragmentation issues. To address this problem, many effective approaches such as app-based and API lifetime-based methods have been proposed to identify incompatible API usages. However, due to the various implementations of API usages and different API invoking paths, there is still a significant weakness of existing approaches, i.e., introducing a massive number of false positives (FP) and false negatives (FN). To this end, in this paper, we propose PSDroid, an automated compatibility detection approach for Android apps, which aims to reduce FPs and FNs by overcoming several technical bottlenecks. Firstly, we make substantial efforts to carry out a preliminary study to summarize a set of novel API usages with diverse checking implementations. Secondly, we construct a refined API lifetime database by leveraging a semantic resolving analysis on all existing Android SDK frameworks. Based on the above two key phases, we design and implement a novel path-sensitive semantic approach to effectively and automatically detect incompatibility issues. To demonstrate the performance, we compared with five existing approaches (i.e., FicFinder, ACRYL, CIDER, IctAPIFinder, and CID) and the results show that PSDroid outperforms existing tools. We also conducted an in-depth root cause analysis to comprehensively explain the ability of PSDroid in reducing FPs and FNs. Finally, 18/30 reported issues have been confirmed and further fixed by app developers.**

***Index Terms*—Compatibility detection, Android app, Path-sensitive analysis, Semantic analysis**

I. Introduction

Mobile applications (apps) have become imperative to people's daily life. The diverse functionalities of Android apps such as accessing the Internet, chatting, and shopping make them inevitable [1], [2]. To facilitate the development of Android apps, Google releases the software development kit (SDK) [3] that provides Android application programming interfaces (APIs). App developers can customize and implement different functions by leveraging the APIs supported by the released SDKs. Yet, along with the high-frequency updates of the Android operating system ranging from Android 1.0 in 2008 to Android 13.0 in 2022 [4], these SDK versions have been updated accordingly from API level 1 to API level 33. Unfortunately, any evolution of the APIs may cause *compatibility issues* due to the fragmentation problem of Android [3], [5], [6], which may inconvenience end-users and significantly degrade user experience [7], [8].

To mitigate it, many approaches have been proposed to identify incompatibility issues in Android apps [5], [9]–[20], which can be divided into two categories: app-based compatibility issue detection [9], [12]–[15] and API lifetime-based compatibility issue detection [5], [10], [11], [16]. The former methods (e.g., FicFinder [9] and ACRYL [13], [14]) only focus on certain types of issues or APIs, because they are usually designed based on some pre-knowledge obtained by manual analysis. Therefore, compared with app-based methods, API lifetime-based methods achieve more promising results, owing to a more complete and comprehensive API lifetime extracted through modeling all the Android API frameworks, where the lifetime of an API means the time span of it being introduced and deprecated during SDK evolution. For example, Li et al. [10] proposed CID, a state-of-the-art approach using an Android API lifetime modeling from Android frameworks 1 to 25. CID utilizes the modeled API lifetime to analyze the changes in the history of Android APIs and further detects compatibility issues through static analysis.

Despite the progress made to alleviate the compatibility problem, there are still many deficiencies in existing solutions, caused by neglecting the context semantics of checking patterns of incompatible API usages, leading to a high False Positive (FP) rate and False Negative (FN) rate. (1) **High FP Rate.** ① Given the fact that the implementation of API usages has several variants, not just the most basic one (e.g., if (Build.VERSION.SDK_INT >= 28) { invokeA(); }), existing approachess [9]–[19] neglect these different implementations and mistake them for compatibility issues, causing false alarms. ② Besides, due to the inconsistency between the declaration in each SDK and that in the Android documentation [21], existing approaches [10]–[16], [20] relying on only the Android documentation or declaration in SDK cannot accurately extract the lifetime of APIs. ③ Thirdly, existing approaches [9], [10], [12]–[19] consider the incompatible API usages in third-party packages as true positives. However, some of them are neither directly nor indirectly invoked by the main packages of the apps. In other words, such incompatible

‡ These two authors contributed equally to this work.
¶ Lingling Fan and Song Huang are the corresponding authors (Emails: linglingfan@nankai.edu.cn, huangsong@aeu.edu.cn).

APIs would not cause incompatible issues to the target app. (2) **High FN Rate.** ① Existing methods [10], [13], [14], [16] consider an API to be used without incompatibility issues once a basic check is found in one of the invoking paths, however, there may be issues in other paths or the semantics of the check is incorrect. ② The inaccurate lifetime and check patterns of the above mentioned APIs can not only introduce FPs, but also cause FNs. Our goal of this paper is to reduce FPs and FNs, and improve the detection practicality.

To this end, we propose PSDroid, a novel approach to detecting API compatibility issues in Android apps. Specifically, we firstly inspected compatibility checks in 645 apps provided by [20], [22], and summarized four typical check patterns (i.e., direct check, ternary expression check, static field check, and try catch check). By proposing a path-sensitive semantic analysis, PSDroid takes into account all the check patterns and invoking paths of API usages. From the perspective of API lifetime modeling, three new features introduced by Kotlin [23] in the concurrent frameworks from 27 to 31 have been considered to construct a more accurate and complete set of API lifetime.

To evaluate the detection ability of PSDroid, we first build a ground-truth dataset containing 12 apps covering different types of compatibility issues and compare it with two state-of-the-art API lifetime-based approaches (i.e., CID [10] and IctAPIFinder [11]). The result shows PSDroid outperforms CID and IctAPIFinder, with precisions 100% vs. 62.5% vs. 62.5%, and recalls 100% vs. 62.5% vs. 62.5%. To further demonstrate the performance in reducing false positives and false negatives, we compared PSDroid with three app-based methods (i.e., FicFinder [9], CIDER [12], and ACRYL [13], [14]) and two lifetime-based approaches (i.e., CID [10] and IctAPIFinder [11]) on 645 real-world apps, followed by an in-depth root cause analysis to highlight the benefits of PSDroid. The results demonstrate PSDroid can sharply reduce FPs (reducing 96.6% and 71.4% potential false positives in CID and IctAPIFinder, respectively) and FNs (22.1% issues only detected by PSDroid). Finally, 30 detected compatibility issues have been reported to the developers, and 18 of them have been confirmed or fixed. The fix results and the positive feedback from app developers demonstrate the practicality of PSDroid.

In summary, we make the main contributions as follows.

- We are the first to systematically summarize four types of API usage check patterns (i.e., direct check, ternary expression check, static field check, and try catch check), which have a big influence on the detection of Android incompatibility issues.
- Several new features such as annotation have been augmented to improve the accuracy of modeling API lifetime (ranging from API level 1 to 31) by employing API semantic analysis.
- We design and implement a path-sensitive approach with semantic analysis, named PSDroid, which leverages the newly-summarized API usage check patterns and a refined API lifetime database to effectively detect compatibility issues.
- We conducted comprehensive experiments to demonstrate the effectiveness of PSDroid and better performance compared with five existing tools in significantly reducing false positives and false negatives. 18/30 reported issues have been acknowledged and fixed.

II. Related Work

A. App-based Compatibility Issue Detection

Wei et al. [9] proposed FicFinder to detect compatibility issues by modeling each pattern of issues as a pair of issue-inducing APIs and issue-triggering context (i.e., an API-context pair). Huang et al. [12] presented CIDER to detect the callback compatibility issues by constructing a callback invocation protocol inconsistency graph (i.e., PI-Graph), so as to capture the structural invocation protocol inconsistencies (causes of callback compatibility issues) across API levels occurring in an app. Scalabrino et al. [13], [14] extracted conditional API usages (i.e., CAUs) of handling evolution-induced API compatibility issues from 688 apps, and proposed ACRYL to detect suspicious compatibility API usages in a given app by using the most frequent CAUs. Xia et al. [15] proposed a machine learning-based approach named RAPID to investigate whether developers handled incompatible API invocation with a replacement implementation.

B. API lifetime-based Compatibility Detection

McDonnell et al. [5] conducted an empirical study to analyze the usage of fast-evolving APIs (e.g., changed and added), and they unveiled faster-evolving APIs are more vulnerable. Li et al. [10] proposed CID to model the lifetime of APIs based on the Android SDK frameworks. They record the API usage information of a given app and extract the usages causing evolution-induced compatibility issues. Similarly, He et al. [11] presented IctAPIFinder to detect API compatibility issues by using a context-sensitive data-flow method to capture the reachable Android OS versions for each API in a given app. Different from the framework-based methods of extracting API lifetime, Mahmud et al. [16] investigated the history versions of API difference reports collected from the Android documentation to model the lifetime of framework APIs. They implemented ACID, which uses a self-built miniature lexer and parser to locate incompatible API usages and generates class hierarchy to locate callback-related compatibility issues. Pei et al. [20] conducted a comparison study of the state-of-the-art compatibility issue detection tools (i.e., CIDER, FicFinder, IctAPIFinder, and CID). They found that lifetime-based tools yield more incompatibility issues than App-based tools, and there remain several limitations, i.e., outdated adapting to the rapid evolution of APIs, lack of characterizing semantics-changing APIs, etc.

C. Automated Compatibility Issue Repair

Besides the detection methods, some studies have been proposed to automatically repair incompatible APIs. Fazzini et al. [24] summarized fixes based on 15 apps and proposed AppEvolve to automatically update incorrect API usages by matching the summarized fixes. Haryono et al. [17] further improved AppEvolve and implemented CocciEvolve, which used the after update examples to perform API updates and

TABLE I: Related work summarization. ●: app-based detection; ○: API lifetime-based detection.

Category	Published time	Tools	Type	APIs that can cause compatibility issues	Semantic Analysis	Path-sensitive Analysis
Detection	2013	McDonnell et. al. [5]	○	Android SDK framework level 1 and API difference reports	✗	✗
	2016	FicFinder [9]	●	5 apps of frequent code revisions	✗	✗
	2018	CID [10]	○	Android SDK framework level 1 to 25	✗	✗
	2018	IctAPIFinder [11]	○	Android SDK framework level 10 to 27	✗	✓
	2018	CIDER [12]	●	Android documentations and 100 compatibility issues	✗	✗
	2019	ACRYL [13], [14]	●	11,863 snapshots of 668 apps	✗	✗
	2020	RAPID [15]	●	300,000 apps	✗	✗
	2021	ACID [16]	○	Google API difference reports	✗	✗
	2022	Literature Review [20]	○	Replicability and comparison of four detection tools	✗	✓
	N.A.	**PSDroid**	○	Android SDK framework level 1 to 31	✓	✓
Repair	2019	AppEvolve [24]	●	15 real-world apps	✗	✗
	2020	CocciEvolve [17]	●	10 most commonly-used APIs	✗	✓
	2021	AndroEvolve [18]	●	20 deprecated Android APIs	✗	✓
	2022	RepairDroid [19]	●	24 app compatibility issues	✗	✗

meanwhile provided semantic and configurable update scripts. Additionally, they extended CocciEvolve by providing AndroEvolve [18], which addressed the defects in CocciEvolve with data-flow analysis and variable name denormalization. In a concurrent work, Zhao et al. [19] proposed a generic patch description language to create fix templates for multiple types of compatibility issues (e.g., OS-induced, device-specific). They implemented RepairDroid, which leverages 22 manually summarized fix templates with control-flow and data-flow analysis to locate and repair incompatible API usages.

Finally, we highlight the main differences between our work (i.e., PSDroid) and existing studies in Table I. The column "Semantic Analysis" refers to whether the tool analyzes different semantic representations of constraints in different API usage check patterns. The column "Path-sensitive Analysis" refers to whether all paths that are relevant to the incompatible API usages are taken into consideration. From the last two columns, we can see that none of the existing methods analyzed different implementations of usage patterns when checking the API usages for their detection or repair tasks, which could lead to many false alarms in practice, while PSDroid leverages a semantic analysis method that is sensitive to various API usage check patterns when detecting compatibility issues. Meanwhile, most of these approaches investigated the incompatible APIs by string matching but neglected the impact of multiple paths. This limitation may cause false positives for compatibility issue detection and make it difficult for developers to locate and verify the potential issues. Although IctAPIFinder leveraged a path-sensitive analysis, it only focused on a simple API usage check pattern (i.e., direct check) instead of all the implementations of check patterns summarized in § IV-B.

III. Motivating Example

Since the Android SDK is evolving, so as the corresponding APIs, some APIs are deprecated or introduced in new SDK versions. Therefore, if an app uses deprecated APIs or uses APIs that have not been introduced into the used SDK version, the app may have incompatibility issues. For example, Fig. 1 shows the simplified code snippet of an app, GPSTest [25], where the class `Utils` [26] is used to check the runtime SDK versions. In this app, the minimum and maximum SDK versions to run it are set to 18 and 31, respectively (Line

```
1  minSDK=18; maxSDK=31;
2  public class Utils {
3      public static boolean hasGrantedPermission() {
4          if (Build.VERSION.SDK_INT < 28)
5              return true; } ...}
6  // Correct invoking path
7  public static boolean isCarrierSupported(...) { ...
8      return Build.VERSION.SDK_INT >= 28 ?
9          getRangeState()!= Gnss.ADR_STATE_UNKNOWN : false;}
10 // Incorrect invoking path
11 public void onGnssReceived(GnssEvent event) {
12     if (Utils.hasGrantedPermission()) {
13         csvFileLogger.writeGnssToFile(event); ...}
14 private void writeGnssToFile(event){ ...
15     // The lifecycle of this API is [24, 31]
16     measurement.getRangeState(); }
```

Fig. 1: Code snippet of `Utils` with incompatibility issues.

1). The incompatible API is *getRangeState()* because it was introduced in SDK version 24, while the minimum SDK version to run this app is 18. Hence, if this app is executed on a device under SDK version 24, it should not invoke *getRangeState()* during runtime, otherwise, it would throw a *NoSuchMethodError* exception, causing the crash or defects.

Specifically, the example declares one static method (i.e., *hasGrantedPermissions* at Line 3). In the correct invoking path (Lines 7-9), the app first checks whether the runtime SDK version is greater than or equal to 28 (i.e., $SDK\_INT >= 28$) in a ternary expression, and invokes the method *getRangeState()* if it is true. In this case, it will not cause incompatibility issues since it already checks for the runtime SDK version and ensures it invokes *getRangeState()* under an appropriate situation. While in the incorrect invoking path (Lines 11-13), the app checks whether the runtime SDK version is less than 28 ($SDK\_INT < 28$) by retrieving the specific global variable, and invokes *getRangeState()* if it is true. In this case, the runtime SDK may be less than 24, then the app invokes a non-existent API, causing an incompatibility issue.

From this example, the challenges can be summarized as: (1) The extraction of constraints for different check patterns across different classes and methods; (2) Semantic analysis of constraints in each API usage path considering the API lifetime. The first challenge is reflected by extracting the constraints of *hasGrantedPermission()* and analyzing which pattern it belongs to across all the classes' static methods. The second challenge is demonstrated by extracting the paths calling specific APIs (e.g., *Util.hasGrantedPermission* → *onGnssReceived()*), and conducting a semantic analysis of the con-

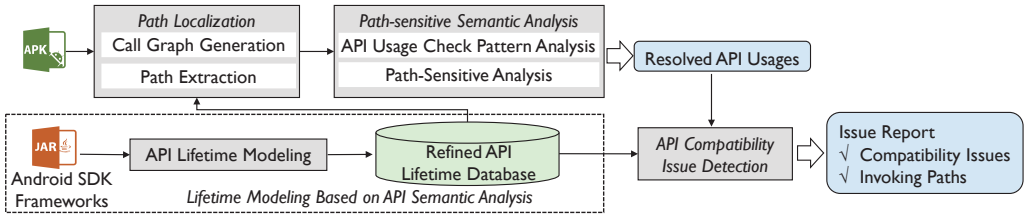

Fig. 2: An overview of PSDroid.

straints along each path and the lifetime of the involved API (i.e., *getRangeState*) to determine the incompatibility. In fact, it is also challenging to accurately model the lifetime of APIs due to different check implementations and different Android versions. However, existing techniques only check whether the runtime SDK version is checked directly in the if-clause, e.g., *if(Build.VERSION.SDK_INT >= 24) {invokeA();}*, while neglecting other semantic condition checks or other paths, causing false alarms.

IV. Approach

In this paper, we propose PSDroid, a path-sensitive approach with semantic analysis, to automatically detect incompatible API usages in Android apps. Fig. 2 presents an overview of PSDroid. It takes an apk as input, and outputs compatibility issues and incompatible invoking paths. PSDroid contains four key phases: (1) *Path Localization*, which first generates the method call graph and then analyzes the call graph to extract all paths relevant to API usages; (2) *Path-sensitive Semantic Analysis*, which considers various patterns of conditional checks in terms of API usages, and extracts all paths that may cause incompatibility issues; (3) *lifetime Modeling Based on API Semantic Analysis*, which refines the lifetime of each API and constructs a database through summarizing 4 types of API updates in SDK versions from 1∼31; (4) *API Compatibility Issue Detection*, which evaluates all the resolved paths of API usages by utilizing the refined API lifetime database to detect incompatible API usage paths.

A. Path Localization

In this section, we aim to extract all the paths that invoke framework APIs.

1) ***Call Graph Generation****:* PSDroid first decompiles the input apk and extracts the call graph using Soot [27], a static analysis framework for Android. The call graph is defined as a tuple G = (M, E), where M is a set of methods of the apk, and E is a set of directed edges connecting two methods. Since Android apps have multiple entry points (e.g., UI interactions, callbacks), we used FlowDroid [28], a precise inter-procedural control-flow graph app analysis framework, to recognize them and generate an individual dummy main method to connect them in G. From the graph G, all the method-invoking sequences (i.e., invoking paths) can be extracted for further incompatibility analysis.

2) ***Path Extraction****:* Based on the generated call graph, PSDroid aims to distinguish the APIs used by the main packages or third party libraries in the app, to further check whether it could cause a runtime incompatibility issue. Before localizing APIs, PSDroid first constructed a database and collected the APIs that need to be localized by analyzing Android documentation [21]. Here, the collected APIs refer to the framework APIs defined in Android. It then traversed the method call graph to localize the used APIs by string-matching with the APIs in the database. For example, in Fig. 1, the API that PSDroid identified and localized is *getRangeState()* (Lines 9, 16), which is directly or indirectly invoked by three methods: *isCarrierSupported()* (Line 7), *onGnssReceived()* (Line 11), and *writeGnssToFile()* (Line 14).

Based on the extracted APIs in the call graph, PSDroid starts from each API (api), and performs a backward analysis to extract the methods that directly or indirectly invoke each Android API. Those methods and api form a sub-graph that shares the same end node (api). From the sub-graph, we extract all the method sequences (i.e., paths) that reach api, denoted by $P_i = \{m_{i1} \rightarrow m_{i2} \rightarrow m_{i3} \rightarrow \cdots \rightarrow api\}, i \in n$, where n is the number of paths in the sub-graph. During path extraction from the sub-graph, there might be recursive calls, making the paths to be infinite. To address it, PSDroid records all the visited methods in each path, and stops extracting paths if the path is with repeated nodes.

Since Android apps are event-driven, some callback methods (i.e., event handler) are triggered by user-events without explicitly being called by other methods. Therefore, m_{i1}, the starting node of a path, may be an event handler. Besides, m_{i1} might also be a deprecated method, which sometimes occurs due to frequent revisions of apps. Taking the example in Fig. 1 for illustration, two paths are finally extracted as the paths that invoke the API *getRangeState()* (Lines 9, 16). The first path is "*isCarrierSupported()* → *getRangeState()*", while the second path is "*onGnssReceived()* → *writeGnssToFile()* → *getRangeState()*". Note that, there may be third-party packages or other supporting libraries that eventually invoke framework APIs, however without being invoked by the main package, i.e., hanging libraries. In this case, we ignore such API usages in order to reduce irrelevant incompatibility API usages. Therefore, we filter out such paths from the extracted paths, and obtained the paths relevant to the app by checking whether the API is directly or indirectly invoked by a method in the main package. Finally, these paths will be further analyzed in the next section to determine whether there exist incompatibility issues when invoking the specific APIs.

B. Path-sensitive Semantic Analysis

In this section, for each extracted path, we aim to identify the patterns to invoke each API. To achieve it, we first

```
1   // Pattern 1: Direct check
2   public static boolean writeGnssToFile(event) {
3       if (Build.VERSION.SDK_INT >= 23)
4           getBufferSizeInFrames(); }
5   // Pattern 2: Ternary expression check
6   public static HashMap<String, String> getInstalledBrowsers() {
7       int packageMatcher =
8           Build.VERSION.SDK_INT >= Build.VERSION_CODES.M ?
9           PackageManager.MATCH_ALL : PackageManager.GET_DISABLED_COMPONENTS;
10  }
11  // Pattern 3: Static field check
12  public class GpsUtils {
13      public static boolean isBackgroundLocationAware() {
14          return Build.VERSION.SDK_INT >= 29; }
15  }
16  public class CollectorService extends Service {
17      void onCreate() {
18          if (GpsUtils.isBackgroundLocationAware()) {
19              startForeground(ID, notification, LOCATION);
20          } else startForeground(ID, notification);
21  }}
22  // Pattern 4: Try-catch check
23  public Class Example{
24      void onCreate(){
25          try{
26              CopyFiles(p1,p2);
27          }catch(Exception e){
28          }
29  }
30  public void CopyFiles(String path1, String path2) {
31      Files.copy(Path.get("path1"),Path.get("path2"),Option.REPLACE_EXISTING);
32  }
```

Fig. 3: Four typical implementations of API usage check patterns.

summarize the usage patterns of invoking APIs in different SDK levels and further check potential incompatibility issues.

*1) **API Usage Check Pattern Analysis**:* To be aware of the context semantics of API-related variables, we inspected the compatibility checks in 1,375 real-world apps collected by [22]. This dataset consists of Github commits related to the revisions of SDK_INT value, which contains various types of API usage check patterns in real-world apps. We then summarize four typical implementations of API usage check patterns, which are often used to avoid evolution-induced incompatibility issues. We spent 2 person-months analyzing API usage check patterns and summarized four patterns as shown in Fig. 3.

***Pattern 1*: Direct check.** The most common practice is to check the value of SDK_INT (i.e., the runtime SDK version) directly in the "`if`" statement. In Fig. 3, Telegram FOSS [29] checks the runtime SDK version in the "`if`" statement ("`if (Build.VERSION.SDK_ INT >= 23)`" at Line 3) before invoking the API *getBufferSizeInFrames()*. In fact, this API is introduced to the Android framework in API level 23, thus, without the constraint (*SDK_INT >= 23*), it would cause incompatibility issues when invoking the API.

***Pattern 2:* Ternary expression check.** Apart from checking the runtime SDK version in the `if` statement, developers can also use ternary expressions as the constraint to invoke the specific API. This pattern is commonly used where the return value of the target API is a parameter of another method. In Fig. 3, a ternary expression (Line 8) is used as the constraint to determine the API that should be invoked at runtime. If it is satisfied at runtime, it would call *MATCH_ALL()*, otherwise, *GET_DISABLED_COMPONENTS()* would be called.

***Pattern 3:* Static field check.** The third usage pattern is to obtain the runtime SDK version check by defining a static variable or a static method, and the runtime SDK version (i.e., the value of SDK_INT) is indirectly checked in the variable or the method. For example, in Fig. 3 the API whose usage needs to be checked is *startForeground() (Line 19)*. The app checks for the runtime SDK version by defining and invoking a static method *isBackgroundLocationAware()* (Line 18), where the result is returned as the constraint condition of the "`if`" statement. With different return values, the app would invoke different "overloading" forms of the same API. Thus, if the runtime SDK version is greater than or equal to 29, the API *startForeground(int, notification, int)* would be invoked in *CollectorService()*, otherwise, another API *startForeground(int, notification)* would be invoked. In addition to storing the check results in a static method or a variable, developers sometimes store the current SDK version in static fields and reuse them in other classes as well.

***Pattern 4:* Try-catch check.** This pattern is to invoke potential incompatible APIs in a try-catch statement, which is commonly used when developers use unfamiliar APIs in third party libraries. For example, in Fig. 3, the API *copy()* (Line 32) is introduced at SDK level 26, which is wrapped in a third party API *CopyFiles() (Line 31)*. And developers need to invoke it in a try-catch statement (Lines 25-28) to prevent potential issues.

For the above four patterns, in addition to adding usage patterns directly before calling the API, developers usually use wrapper methods to check whether to invoke the API, i.e., the constraint is checked in the callee methods that finally would invoke the API. That is also one of the reasons why the call path of an API is required to be extracted, and the constraints along the path should be extracted as well. As shown in Fig. 1, the API is invoked by the wrapper method (i.e., *writeGnssToFile()*) with a static method check (i.e., *hasGrantedPermission()*).

TABLE II: Frequency of different API check patterns.

	Pattern 1 & 2	Pattern 3	Pattern 4	Total
#APP	503 (78.0%)	249 (38.6%)	452 (70.1%)	645
#API	84,461 (86.1%)	8,072 (8.2%)	17,992 (18.3%)	98,137

To investigate the frequency of each pattern in real-world apps, we investigated 645 real word apps from the dataset [20] by using FlowDroid [28]. Table II displays the result of the 645 apps and 98,137 API usages with check patterns extracted from the apps. We can see the most commonly-used pattern is direct check and ternary check, accounting for 86.1% of API usage paths, and 78.0% apps use such a pattern at least once to check the API usage constraint, while the static field check and try-catch pattern account for 38.6% and 70.1% in 645 apps, respectively. Developers may also use multiple checking patterns to avoid compatibility issues in one API usage path. Based on the 4 summarized API usage check patterns, we then illustrate the mechanism of how to analyze the constraint semantics in them along with each API call path in the following path-sensitive analysis section.

*2) **Path-sensitive Analysis**:* For different paths with different types of API usage check patterns, we first need to separately extract the constraints in each path. Since there may be several SDK version checks (i.e., constraints) along the path, we thus resolve them to investigate whether there exist SDK version checks before invoking APIs, and what exactly

the semantics of the constraints are.

For each API call path, to extract the semantics of SDK version checks from the four usage patterns, we first record the variables and static methods related to SDK version checks, and extract the context of each variable and static method by extracting the relevant constraints. Specifically, in each method, we first generate the Control Flow Graph (CFG) [30], and extract the patterns for each method call. We employ different analysis strategies for different patterns. For *Direct check* and *Ternary expression check*, they are both converted to a branch structure, and PSDroid records where *Build.VERSION.SDK_INT* is defined or used, and the constraint in the branch, such as the constraint *SDK_INT >= 23* (Line 3) in Fig. 3. In fact, there is no difference in handling these two patterns at bytecode level with SOOT [27]. We categorize them into two types because they are summarized from the source code of real-world apps, where they are totally different. It is worth mentioning that source-code-level analysis tools such as ACRYL [13] and ACID [16] only considered *Direct check* and neglected *Ternary expression check*, which would miss the semantics in the latter pattern.

For *Static field check*, since the API usage constraint may be in various forms, the semantics of different forms of patterns should be extracted and recorded in different ways. To achieve it, PSDroid identifies and records SDK check-related static variables and static methods. For static variables, PSDroid identifies how these variables are defined or used. Typically, such static variables are usually defined to store the comparison results of the runtime SDK and a specific SDK version, for example, "`public static boolean RuntimeSDK = Build.VERSION.SDK_INT >= 19`". For static methods, PSDroid focuses on the return value to observe whether SDK check-related results are returned to the caller method. The returned result is typically used as the constraint to invoke a specific API. For example, "`return Build.VERSION.SDK_INT >= 29`". For *Try-catch check*, PSDroid first identifies the instruction that invokes incompatible APIs and then recognizes whether it is in the block of a try-catch structure by traversing the CFG. We assume that all exceptions would be caught by default if the API induces errors.

After identifying different types of patterns, PSDroid performs a CFG-based analysis to match them with the API or the wrapped method in the path. Note that there may be a complicated situation where one API is invoked several times in a single method, but with different forms of SDK version checks. In this case, PSDroid extracts patterns for each call separately. In this way, PSDroid can extract the SDK check-related patterns for each path.

Taking the example in Fig. 1 for illustration, PSDroid obtains two resolved API usage paths for the API *getRangeState()*, as shown in Fig. 4. In each method, the constraint related to SDK version checks is extracted. The constraint in the method *isCarrierSupported()* (Line 7) is "`>=28`", which is extracted from a ternary expression check. The constraint in the method *onGnssReceived()* (Line 11) is "`<28`", which

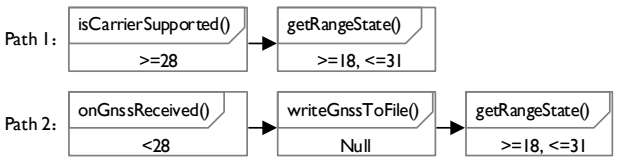

Fig. 4: An example of resolved API usages.

is extracted from a static method check. "Null" indicates no SDK-related checks in the method. Besides the patterns along the path, for the end of each path, the minimum and maximum SDK versions of the app will also be extracted (i.e., 18 and 31) and used as the default runtime SDK version range of the target API *getRangeState()*. In the next section, the resolved SDK version range of each API will be extracted and compared with the API lifetime, in order to identify the incompatibility issues.

C. API Lifetime Modeling

The lifetime of an API indicates the time interval that the API is introduced and deprecated in the Android framework, which is also the basis for identifying API incompatibility issues. In this paper, we consider the SDK versions from 1∼31 since the latest version that provides API declarations is 31 (Android 12.0), and automatically model the lifetime of each API. To identify APIs, PSDroid first analyzes API declarations for each SDK version based on Android documents [21]. However, different declarations may refer to APIs with the same semantics but with different declarations [31], [32]. Such inconsistency may introduce false alarms for API compatibility detection based on API lifetime, thus, we need to identify such different APIs, treat them equally (i.e., unify them to one API), and update the lifetime of each API based on the SDK documentation [33]. Consequently, we summarize 4 types of such API updates as follows.

Annotation. Since SDK version 29, some APIs are declared with annotation (e.g., *@LayoutResint*) to indicate the attribute of their parameters, however, there are actually no differences when using these APIs. By investigating the Android documentation, we find two typical types of annotations: (1) **Replacement**, where some parameters of the API are directly replaced with annotation. For example, as shown in Fig. 5, in SDK version 1∼28, the API *setContentView(int)* (Line 2) is declared with the parameter *int*. While since SDK version 29, this parameter has been replaced with *@LayoutResint*. (2) **Prefix**, where some parameters of the API are annotated with a prefix. For example, the API *ColorStateList: void <init>(int[][],int[])* (Line 3) in Fig. 5 is declared with parameters *(int[][],@ColorInt int[])* (Line 5) since SDK version 29, indicating the attribute of the second parameter is ColorInt. We unify these parameters by replacing or deleting the annotations, and update the lifetime of APIs accordingly, e.g., the lifetime of *setContentView(int)* is updated to 1∼31.

Return value type. The Android framework introduces Kotlin [23] in SDK version 29 as the programming language for Android apps, although there is no difference in analyzing apps written in Java and Kotlin at the bytecode level, some APIs with the same semantics are written in different names/forms

```
1   // Annotation
2   API 1~28: <android.service.dreams.DreamService: void setContentView(int)>
3             <android.content.res.ColorStateList: void <init>(int[][],int[])>
4   API 29~31:<android.service.dreams.DreamService: void setContentView(@LayoutResint)>
5             <android.content.res.ColorStateList: void <init>(int[][],@ColorInt int[])>
6   // Return value type
7   API 1~28: <android.text.AlteredCharSequence: Java.lang.CharSequence subSequence(int,int)>
8   API 29~31:<android.text.AlteredCharSequence: CharSequence subSequence(int,int)>
9   // Extends method
10  API 1~28: <java.util.concurrent.TransferQueue: boolean tryTransfer (E, long,
11  java.util.concurrent.TimeUnit)>
12  API 29~31:<java.util.concurrent.TransferQueue extends java.util.concurrent.Blocking
13  Queue: boolean tryTransfer(E,long,java.util.concurrent.TimeUnit)>
14  // Generic type
15  API 1~25: <android.view.View: android.view.View  findViewById(int)>
16  API 26~28:<android.view.View: T findViewById(int)>
17  API 29~31:<android.view.View: T findViewById(@IdResint)>
```

Fig. 5: Augmentation types of API lifetime.

and should be taken into consideration to accurately model the API lifetime. For example, the API *subSequence(int, int)* (Line 7) in Fig. 5 declares that the data type of its return value is *java.lang.CharSequence* in SDK version 1∼28, while it has been changed to *CharSequence* since SDK version 29 (Line 8). Similarly, *java.lang.Runnable* evolves to *Runnable*, *java.lang.Iterable* evolves to *Iteratable*, etc. PSDroid unifies these forms and updates the lifetime of the related APIs accordingly, e.g., the lifetime of the API *subSequence(int, int)* is updated to 1∼31. Such reduced class naming will affect the lifetime modeling of APIs, if not correctly modeled, different forms of the same API will be regarded as two APIs with different lifetimes, which would cause false alarms for lifetime-based methods.

"*Extends*" method and generics. Some API methods are reorganized to explicitly indicate the super class by using "*extends*", or use generic data types as their variable arguments, which challenges the reliability of the API lifetime. For example, in Fig. 5, the API *tryTransfer(...)* (Line 12) declares that its methods extend the class *BlockingQueu* in SDK versions 29 to 31. Similarly, for API *findViewById* (Lines 16, 17), it employs the generic type `T` after SDK version 26 but the actual semantic and usage of this API remains unchanged. PSDroid unifies them as well and updates the lifetime accordingly.

Based on the above 4 types of API updates, PSDroid models the API lifetime automatically. Specifically, PSDroid first extracts all API declarations of API levels 1∼31 from Android documents [21] and stores them in lists L_i where $i = 1, \cdots, 31$, representing the API level. It then updates the declaration of each API in L_i according to the above 4 features by string mapping and replacement, e.g., replacing the API declaration *CharSequence* with *java.lang.CharSequence* in $L_j (j = 29, 30, 31)$. To specify the lifetime of each API, PSDroid traverses all the APIs involved in API levels 1∼31, checks the existence of each API in each API level L_i, and finally obtains the introduced SDK version and deprecated SDK version of each API to systematically model their lifetimes.

D. API Compatibility Issue Detection

Based on the extracted SDK-related patterns along each path and the augmented lifetime of each API, in this section, we aim to identify whether the app has potential incompatibility issues by comparing the runtime constraints of SDK versions to invoke the APIs and the lifetime of APIs. Specifically, for each path ($path$) resolved in § IV-B, the target API is denoted by api, PSDroid first solves the patterns extracted in $path$. If there exists a solution that satisfies the constraints, we record it as the runtime range of the SDK versions for invoking api, denoted by $R_{runtime}$, and the actual lifetime of api is denoted by R_{life}. If $R_{runtime} \nsubseteq R_{life}$, which means the app may run on a SDK version v where $v \in R_{runtime}$ while $v \notin R_{life}$, causing incompatibility issues when calling api. PSDroid identifies such incompatibility issues and further reports how and where this issue occurs by computing the difference set between $R_{runtime}$ and R_{life} to localize the root cause. For example, for Path 2 shown in Fig. 4, the intersection of the extracted runtime API usage is $[18, 28)$ and the lifetime of the target API is $[24, 31]$. Since $[18, 28) \nsubseteq [24, 31]$, PSDroid would report an issue, because the app may be run on devices with SDK version 18∼23 where the API *getRangeState()* has not been introduced but invoked, which would cause incompatibility issues.

V. Experiments

In this section, we evaluate the effectiveness of PSDroid by answering the following research questions:

RQ1: Can PSDroid detect different types of API compatibility issues?

RQ2: Can PSDroid effectively detect API compatibility issues?

RQ3: Can PSDroid outperform existing tools in detecting API compatibility issues?

RQ4: What is the feedback in terms of the issue reports of PSDroid?

A. RQ1: Detection Ability of PSDroid

Setup. In this experiment, we build a ground truth dataset containing 12 apps that cover different types of compatibility API usages, as shown in Table III. The first six apps come from the CID benchmark dataset [10], and the remaining ones are constructed with four types of compatibility API usage check patterns: (1) *Basic with unrelated If-then*, where there is indeed an "If-then" check about the SDK version, while the issue-induced API (e.g., *AlarmManager.setExact*) is actually not protected by the "If-then" check after analyzing the CFG of the method. (2) *Wrong If-then protection*, where the condition in the "If-then" check is incorrectly set to protect the issue-induced API. (3) *Static protection*, where the issue-induced API is protected by a static field. (4) *New API*, where the API (e.g., *CellSignalStrengthNr.getCsiRsrp*) is introduced in recent API levels. We compare the detection results of PSDroid with the state-of-the-art lifetime-based tools (i.e., CID, IctAPIFinder), which has been demonstrated to be more effective than app-based tools [20].

Result. Table III shows the comparison results. PSDroid resolves all incompatible API usages while CID and IctAPIFinder correctly resolve six usages (marked with ⊛ and ◯), and they both report three false positives and three false negatives, whose precisions are 100%, 62.5%, 62.5% and recalls are 100%, 62.5%, 62.5%, respectively. The main reason for the false negatives is that CID did not perform the CFG

TABLE III: Comparison results of existing tools on the ground truth apps. ⊛: true issue; ∗: false issue;◯: no issue; ∅: missed issue.

App	Relevant API	CID	IctAPIFinder	PSDroid
Basic	AlarmManager.setExact	⊛	⊛	⊛
Forward	AssetInputStream.getAssetInt	⊛	∅	⊛
GenericType	TreeMap.replace	⊛	∅	⊛
Inheritance	transitionManager.go	⊛	⊛	⊛
Varargs	KeyProtection.Builder.<init>	⊛	⊛	⊛
If-then protection	AlarmManager.setExact	◯	◯	◯
Basic with unrelated If-then	AlarmManager.setExact	∅	⊛	⊛
Wrong If-then protection	Webview.getWebViewClient	∅	⊛	⊛
Static method protection	VibratorManager.vibrate	∗	∗	◯
Static variable protection	VibratorManager.vibrate	∗	∗	◯
Try-catch protection	Files.copy	∗	∗	◯
New API	CellSignalStrengthNr.getCsiRsrp	∅	∅	⊛
#True Positives	-	6	6	12
#Precision	-	62.5%	62.5%	100%
#Recall	-	62.5%	62.5%	100%

analysis on each method (i.e., *Basic with unrelated If-then*) and semantic analysis on the API usage check patterns (i.e., *Wrong If-then protection*). For IctAPIFinder, the reason for the false negatives is the lack of lifetime knowledge of the specific APIs. The main reason for all the false positives is that CID and IctAPIFinder did not consider different implementations of API check patterns.

Answer to RQ1: PSDroid can effectively detect different types of compatible API usages, achieving 100% precision and 100% recall rate on our ground truth dataset, where the precisions of CID and IctAPIFinder are 62.5% and 62.5%, and the recalls are 62.5% and 62.5%, respectively. It indicates PSDroid outperforms state-of-the-art incompatibility issue detection tools.

B. RQ2: Effectiveness of PSDroid

Setup. To evaluate the effectiveness of PSDroid, we exploit it to detect API compatibility issues on a baseline dataset built from the AndroidCompass dataset [22]. Specifically, we manually checked 1,200 changes in AndroidCompass to build the dataset. To obtain the apps with API compatibility issues, we first extract the commits that only revise one line of code in a method, which might probably revise compatibility issues from our experience, and manually verified the changes to ensure they were modified to check the SDK version. Finally, we obtained the versions before and after the commit to construct our baseline dataset (with 36 commits from 30 app updates), as shown in Table IV.

Based on the constructed dataset, we use PSDroid to detect compatibility issues in both old and new versions of the apps, and the detected issues are recorded and analyzed separately. We classify all detected API compatibility issues into three groups to conduct an in-depth analysis. (1) "*Without Checks*": which represents the issues that have no SDK version checks at all for all the related API invocations. (2) "*Partial Checks*": which represents the issues that have correct SDK version checks in some of the invoking paths, but fail to protect API usages in all invoking paths. (3) "*Incorrect Checks*": which denotes the detected issues with incorrect SDK checks for all related API invocations.

Result. Table IV shows the detection results, where "Fixed" and "Newly Introduced" denote the numbers of issues fixed and newly introduced in the new version, respectively. Note that an app update may include both new functions and fixed issues, as well as new issues introduced by new functions. As shown in Table IV, PSDroid detected 347 API incompatibility issues (i.e., 98 newly-introduced + 249 remained issues) on the new versions of 30 apps, and 361 issues (i.e., 112 fixed + 249 unfixed issues) on the old versions. Overall, it can be seen that more than half of the detected issues (i.e., 78.39%) are caused by "*Without Checks*", indicating a majority of API incompatibility issues have not been noticed by developers. Besides, we observe that the number of detected issues with partial checks is larger than that of issues with incorrect checks in all the usage paths, and the number of those two issues in new versions is also larger than those in old versions. A possible reason is that although some developers are aware of API compatibility issues, they lack the information to locate and correctly fix related issues.

To validate the correctness of issues detected by PSDroid, we further sample 25% compatibility issues of each app (i.e., 177 issues in total) and manually check whether the APIs are correctly used without incompatibility issues. It is found that **ALL** the extracted API usage paths are correct, while 11 detected issues (6.2%) in specific paths are false positives due to the inconsistency between the declared APIs in the declaration file of each SDK version and the APIs that can be used in each SDK version from the Android documentation. For example, the API *View: String toString()* is not declared in SDK frameworks before 17, while it can be invoked before version 17 from the Android documentation. Since the API lifetime is modeled based on the declaration of each SDK, such inconsistencies may occur and cause false positives.

Answer to RQ2: PSDroid can effectively detect API incompatibility issues with high precision (93.8%) on our dataset. Besides "without checks", the other two types of issues (i.e., partial checks and incorrect checks) are also prevalent which can be uniquely detected by PSDroid.

C. RQ3: Comparison with Existing Tools

Setup. To demonstrate the tool advantages compared with the existing incompatibility detection tools, we compare PSDroid with five existing approaches (i.e., FicFinder [9], CID [10], IctAPIFinder [11], CIDER [12], and ACRYL [13], [14]) on 645 real-world apps, which are collected in [20]. Note that the dataset is a partial ground truth dataset, which means that these apps have compatibility checking functions but are unable to be verified whether the checking is correct or not. Other tools listed in Table I are not used since they are either not publicly available or we did not get a reply from the authors. Note that, since CID was published a few years ago with API lifetime modeled based on SDK versions 1∼25, to make a fair comparison, we upgrade CID (denoted by CID^{+}) by augmenting the SDK versions as 1∼31 (the same as PSDroid). Due to the limitation of each tool (e.g., cannot be processed by the underlying tool or causing out-of-memory errors), some apps cannot be successfully analyzed. Therefore, we record and compare the success rate of the analysis and the detected issues and the time cost of PSDroid on analyzing apps.

Result. Table V shows the comparison results with existing tools, where PSDroid ^{ALL} represents the incompatible API

TABLE IV: Detection results of PSDroid.

App Name	Old Version	# Issues Detected by PSDroid				New Version	# Issues Detected by PSDroid				#Fixed	#Newly Introduced
		All	Without Checks	Partial Checks	Incorrect Checks		All	Without Checks	Partial Checks	Incorrect Checks		
App Manager	2.5.21	19	19	0	0	2.5.22	25	24	0	1	1	7
Step and Height counter	1.3	0	0	0	0	1.5	0	0	0	0	0	0
Step and Height counter	1.22	15	14	0	1	1.3	0	0	0	0	15	0
App Manager	2.5.20	15	15	0	0	2.5.21	19	19	0	0	0	4
WiGLE WiFi Wardriving FOSS	2.51	5	1	1	3	2.6	5	1	1	3	0	0
GPSTest	3.9.1	0	0	0	0	3.9.2	0	0	0	0	0	0
microMathematics Plus	2.20.1	6	6	0	0	2.21.0	7	7	0	0	0	1
Tusky	12.1	1	1	0	0	13.1	2	1	1	0	1	2
ProtonVPN	2.4.31.0	18	17	0	1	2.6.0.0	2	2	0	0	17	1
Termux Tasker	0.4	2	2	0	0	0.5	0	0	0	0	2	0
VirtualXposed	0.20.2	1	1	0	0	0.20.3	2	2	0	0	0	1
GPSTest	3.8.4	0	0	0	0	3.9.0	0	0	0	0	0	0
Limbo x86 PC Emulator	4.1.0	2	0	2	0	5.0.0	1	1	0	0	2	1
Nextcloud	3.14.1	15	13	0	2	3.15.1	18	17	0	1	3	6
VirtualXposed	0.19.0	1	1	0	0	0.20.2	1	1	0	0	1	1
RedReader	1.14	6	6	0	0	1.15	6	6	0	0	0	0
Simple Keyboard	74	4	4	0	0	75	4	4	0	0	0	0
Tower Collector	2.5.1.68	16	15	0	1	2.6.0.69	3	2	0	1	14	1
Mupen64Plus AE	3.0.87	12	11	1	0	3.0.246	10	9	0	1	8	6
Telegram FOSS	7.1.3	42	38	4	0	7.2.1	54	50	4	0	0	12
AntennaPod	2.1.4	11	11	0	0	2.2.0	10	9	0	1	3	2
Pocket Paint	2.7.0	7	7	0	0	2.7.1	3	3	0	0	4	0
Presence Publisher	2.1.1	50	19	24	7	2.2.0	63	27	26	10	4	17
App Manager	2.5.15	10	10	0	0	2.5.16	16	15	0	1	0	6
DSub	5.5.0	14	13	1	0	5.5.1	14	9	5	0	3	3
DSub	5.4.4	34	30	4	0	5.5.0	19	8	11	0	22	7
SkyTube	2.973	3	1	0	2	2.974	4	3	0	1	0	1
Shelter	1.5.1	0	0	0	0	1.6	0	0	0	0	0	0
Presence Publisher	2.0.0	39	19	15	5	2.1.0	46	20	15	11	7	14
BiglyBT	1.2.6.6	13	9	2	2	1.3.0.2	13	10	0	3	5	5
Total		**361**	283	54	24		**347**	250	63	34	**112**	**98**
Average		12.03	9.43	1.80	0.80		11.57	8.33	2.10	1.13	3.73	3.27
Percentage		100.00%	**78.39%**	**14.96%**	**6.65%**		100.00%	**72.05%**	**18.16%**	**9.80%**	-	-

TABLE V: #Issues detected by existing tools and PSDroid. #Succ. Apps: the number of apps that are successfully analyzed.

-	App-based methods			API lifetime-based methods			$PSDroid^{ALL}$	PSDroid
	FicFinder	CIDER	ACRYL	CID	CID$^+$	IctAPIFinder		
#Succ. Apps (rate)	587 (91.0%)	623 (96.6%)	596 (92.4%)	409 (63.4%)	416 (64.5%)	407 (63.1%)	643 (99.7%)	643 (99.7%)
#Apps with issues	132	22	195	341	375	353	562	481
#Issues	167	30	259	20,642	24,722	4,287	49,211	3,408

usages in the main code of apps, third-party libraries, and dead code in apps, while PSDroid represents issues that occur only in the main code of the apps. As we can see, PSDroid successfully analyzed almost all the apps, achieving a 99.7% success rate and outperforming others. CID and IctAPIFinder perform the worst, achieving only 63.4% and 63.1% success rates. For CID, the reason may be that it restores intermediate results which may cause an out-of-memory error when processing complex apps. For IctAPIFinder, the reason came from the limitation of Soot (old versions of Soot in particular). PSDroid detected 3,408 compatibility issues in 481 out of 643 apps, which is less than the number of issues in other lifetime-based tools (i.e., 24,722 issues in 375 apps by CID$^+$, 4,287 issues in 353 issues by IctAPIFinder). This might indicate that PSDroid can precisely and effectively detect compatibility issues, which will be discussed in the following FN analysis. For app-based tools, the number of apps with issues and detected issues (i.e., CIDER detected 30 issues in 22 of 632 apps) is far less than lifetime-based tools, which confirms that it is essential to model all API lifetime to harvest incompatible APIs so as to support automated detection of compatibility issues in Android apps [20]. On average, PSDroid takes 70.3 seconds per app for analysis, which is comparable to existing related tools [9]–[11], [13], [16].

To make the results clearer, we highlight the comparison results in Fig. 6(a) and Fig. 6(b), where Fig. 6(a) shows the comparison between app-based detection approaches (i.e., FicFinder, CIDER, and ACRYL) and PSDroid, while Fig. 6(b) shows the comparison between the API lifetime-based approaches (i.e., CID$^+$, IctAPIFinder) and PSDroid. As a result, PSDroid covers the most overlap issues (i.e., 7 issues with FicFinder (4.2%), 42 issues with ACRYL (16.2%), 752 issues with IctAPIFinder (26.1%), 595 issues with CID$^+$ (3.0%)) while reducing many potential false positives (i.e., 18,906 issues in CID$^+$ (96.6%), 2,057 issues in IctAPIFinder (71.4%)) and many potential false negatives (i.e., 359 issues only detected by PSDroid (22.1%)). In addition to demonstrating the overlap and discrepancy of PSDroid with the other 5 existing approaches, we manually checked the discrepancy issues between PSDroid and other tools to investigate the FP and FN reasons.

Comparison with app-based approaches. We can see from Fig. 6(a) that there is only 7 overlapped issues between FicFinder and PSDroid and 42 overlapped issues between ACRYL and PSDroid, which are all caused by without check. Except the overlapped issues, the other issues detected by ACRYL and FicFinder contain many false positives, which are validated by manually investigating the constraint checks in the invoking paths. The first root cause for the false positives of ACRYL and FicFinder is that these detected incompatible APIs are in the third-party libraries and meanwhile not invoked by main packages. Another reason is that neither of them correctly recognize the version check semantics. For example, the app, WiGLE [34], uses a static field check to check the runtime SDK version, while both tools considered there is no check before invoking the API *checkSelfPermission()* and thus

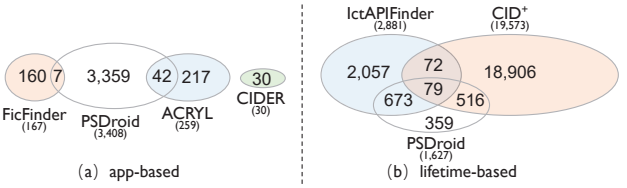

Fig. 6: Comparison with existing approaches.

report a compatibility issue, which is actually a false alarm. Besides, the two app-based tools have many false negatives, because they only take limited APIs into account instead of the complete API lifetime, and the check patterns defined by them may be easily outdated due to the rapid evolution of the Android operating system. There is no overlapped issue between CIDER and PSDroid, because CIDER can only detect a few callback-related compatibility issues.

Comparison with API lifetime-based approaches. We excluded apps that cannot be successfully analyzed by three lifetime-based approaches and the comparison results of the remaining 292 apps are shown in Fig. 6(b). Among those apps, CID$^+$, IctApiFinder, and PSDroid have reported 19,573, 2,881 and 1,627 compatibility issues, respectively.

• **FP analysis.** From Fig. 6(b), we find that most of the compatibility issues detected by CID$^+$ are caused by "out-of-path API usages" and the incompleteness of API lifetime modeling. "out-of-path API usages" indicates the API is used beyond a valid path, such as in a dead branch that cannot be invoked by the main package, which will not induce runtime issues in the apps. Most of the false positives introduced by IctAPIFinder are caused by neglecting different checking variants and the incompleteness of API lifetime modeling. Specifically, *(1)* a large number of APIs are never directly or indirectly invoked by the main packages (i.e., out-of-path API usages), causing a great number of irrelevant issues. *(2)* Many new features have a great effect on the API lifetime modeling, which introduces false positives both in main packages and third-party packages. *(3)* Another reason is that the detection approaches did not consider different check patterns and would mistakenly treat other types of check patterns as without checks. In other words, the improvement on the types of check patterns (§ IV-B1), the method of API lifetime modeling (§ IV-C), and the path-sensitive analysis (§ IV-B2) in PSDroid can significantly reduce false positives in the reported API incompatibility issues.

• **FN analysis.** For the false negatives of IctAPIFinder and CID$^+$ in Fig. 6(b), we summarize four root causes as follows. (1) *Path-Insensitive.* CID and IctAPIFinder did not perform a semantic analysis of all types of API usage check patterns for all the invoking paths. CID only employs one simple check pattern (i.e., direct check), and if the runtime SDK version has been checked in any of the paths to the API, it considers there are no incompatibility issues. (2) *Semantics Loss.* CID only checks if there is an SDK-related constraint before invoking the API, without analyzing the semantics of the constraint, i.e., the specific constraint such as "*VERSION.SDK_INT<28*". If there is an SDK-related constraint, CID regards the API as protected and would not cause incompatibility issues. (3) *Lack of CFG analysis.* CID does not perform a control flow graph (CFG) analysis to extract patterns. If there are two potential incompatible APIs in one method (one with a constraint and the other one without a constraint), CID mistakenly assigned the first constraint to the second API. (4) *Lack of API lifetime knowledge.* IctAPIFinder has no knowledge of features of newly-introduced APIs, leading to an incomplete API lifetime.

TABLE VI: Feedback from app developers.

APP	#Stars	Version	IssueID	Issue State	Incompatible API
Music Player GO	1.1k	v4.4.20	#383 #463	Fixed Confirmed	onRequestPermissionsResult() startActivityAndCollapse()
Owncloud	3.4k	v2.20	#3736	Fixed	isStreamMute()
Triger	105	v3.4.3	#73	Fixed	onRequestPermissionsResult()
Anki-Android	5.3k	v2.16alpha49	#10469	Confirmed	onReceivedHttpError()
AmazeFileManager	4k	v3.6.7	#3194	Responded	setBlockModes()
MaterialFBook	126	V4.0.3	#235	Waiting	getColor()
Gpstest	1.1k	v3.9.16	#583	Confirmed	asVerticalAccuracy()
Markor	2.2k	v2.9.0	#1640	Confirmed	now()
MicroMathematics	316	v2.22.0	#116 #115	Waiting Waiting	getTreeDocumentId() checkSelfPermission()
Osmeditor4android	262	v17.0.3.0	#1566	Confirmed	showContextMenu()
AppManager	1.8k	v3.0.0-alpha03	#687	Confirmed	getNextBucket()
Phonograph	2.7k	v1.3.5	#953	Waiting	onApplyWindowInsets()
ConnectBot	1.9k	v1.9.8	#1153 #1154	Waiting Waiting	GetActiveNetworkInfo() get()
AntennaPod	4.4k	v2.6.2	#6009 #6008	Responded Responded	getConnectionInfo() GetActiveNetworkInfo()
OpenBoard	1.7k	v1.4.3	#686	Waiting^C	invalidateOutline()
PSLab	2k	v2.0.10	#2352	Waiting^C	getLong()
Pixiv Shaft	2.3k	v3.2.21	#477	Waiting	getDynamicShortcuts()
Forecastie	766	v1.2.11	#682	Waiting	getActiveNetworkInfo()
Omni-Notes	2.4k	v6.0.5	#863 #864	Fixed Confirmed	getPackageName() toggleSoftInput()
Open sudoku	314	v3.8.1	#10	Fixed	requestApplyInsets()
Vanilla	922	v1.10	#1143	Waiting	getJobId()
BikeSharingHub	11	v2.0.6	#40	Waiting	onRequestPermissionsResult()
SuntimesWidget	226	v0.14.7	#620 #621	Confirmed Fixed	setBackgroundTintList() onVisibilityAggregated()
Runnerup	614	v2.4.5.0	#1119	Confirmed	stopLeScan()

[1] Waiting^C: the issue caused a crash.

Answer to RQ3: PSDroid outperforms existing tools in alleviating false positives and false negatives. With path-sensitive semantic analysis based on four usage patterns, PSDroid can efficiently localize API usage paths invoked by main package methods and resolve the accurate semantic of usage patterns, which sharply reduces FPs (i.e., reducing 96.6% and 71.4% potential false positives in CID$^+$ and IctAPIFinder, respectively) and FNs (22.1% issues only detected by PSDroid) in the experimental dataset.

D. RQ4: Feedback from App Developers

To evaluate the usefulness of PSDroid, we collected active repositories used in existing studies [35], [36] from Github and Gitlab, most of which with high stars are also available on Google Play. We then analyzed their up-to-date versions with PSDroid and have reported 30 issues to developers for their feedback, as shown in Table VI. For each issue, we reported: (1) the invoking path with incompatibility issues, (2) the root cause, and (3) the repair suggestion. To avoid overwhelming app developers, for each app, we submitted no more than two errors. So far, 18/30 reported issues have been confirmed or fixed. Remaining 12 issues are already confirmed by ourselves, and waiting for responses from developers. Some instances are shown as follows.

• ***Activity: void onRequestPermissionsResult() [23, 31].*** This API is in the app, Music Player GO [37], which is a video play app and has 929 stars on Github. As shown in Fig. 7, the app overrides this API to automatically receive request permission results. However, the lifetime of *onRequestPermissionsResult* is identified to be 23∼31, which may cause crashes if it

```
1  minSDK=21; maxSDK=31;
2  override fun onRequestPermissionsResult(request, permission){...
3  +  if (VersioningHelper.isMarshmallow())
4         super.onRequestPermissionsResult(request, permissions, Results)
5  }
6  object VersioningHelper {...
7      fun isMarshmallow() = SDK_INT >= Build.VERSION_CODES.M
8  }
```

Fig. 7: A fixed issue by Music Player GO.

is invoked on SDK version 21 or 22. The app developer confirmed this issue [38] and fixed [39] it by adding a static field check (Lines 3, 7) before invoking it (Line 4). This issue is also detected [40] and fixed [41] in App Trigger [42].

• ***WebViewClient: void onReceivedHttpError() [23, 31].*** This API is in the app, Anki-Android [43], which is a spaced repetition flashcard app to memorize things and has 4.7k stars on Github. It overrides this API to display a "help" page in case of missing images (e.g., a 404 from the resource that was attempted in the WebView) in the method *onReceivedHttpError()*. The minimum SDK version to run it is 21. However, the lifetime of this API is 23∼31, which would introduce incompatibility issues when the app runs on devices with SDK version 21 or 22. The app developer confirms this issue on GitHub [44] and appreciates us providing such a static analyzer that outperforms existing tools (i.e., precise invoking path and root cause) to help them localize this issue.

• ***Builder: Builder setBlockModes() [23, 31].*** This API is in the app, AmazeFileManager [45], which is a file manager app and has 3.8k stars in Github. The minimum SDK version to run this app is 14, while the API is introduced in SDK version 23. The developer confirmed on Github [46] that they did not add an SDK version check when invoking this API. And they temporarily make a workaround (i.e., add an annotation *@RequiresApi(api = 23)*) to suppress compiling warnings, which might also cause an unpredictable crash [47].

Besides the fixing solutions from developer feedback, we also investigate some common fixing practices. (1) *Add conditional check*. In some cases, developers only add an SDK check for one of the paths instead of all the API usage paths, which may still result in incompatible API usages in other invoking paths. For example, from GPSTest 3.9.1 to 3.9.2 [25], *VERSION.SDK_INT >= 28* is added to fix the issue [48] related to *getRangeState*, however, it still results in an incompatibility issue with partial checks. (2) *Delete API usages with incorrect checks.* We find some developers delete incorrect checks together with the incompatibility API usage to fix issues. From OpenBoard 1.4.2 to 1.4.3 [49], an API usage with an incorrect check was deleted [50] to fix the issue related to the API *hasGlyph(java.lang.String)*. From our observation, deleting the code containing incompatible API usage is a common practice for developers to fix API incompatibility issues. 15 issues were fixed in this way from Dsub 5.4.4 to 5.5.0 [51].

Answer to RQ4: Some of the detected issues are confirmed by developers, and while the fixing solutions vary, some even introduce new incompatibility issues. Developers tend to simply add a check or delete incompatibility API usages.

VI. Discussion

Limitations. First, since PSDroid is built on the static analysis framework, Soot [27], it inevitably inherits limitations from Soot, such as the inability to handle reflective calls, native code, and multi-threading features, and thus cannot recognize unresolved types in Soot. Besides, currently PSDroid is not equipped with the ability to eliminate the "dead" invoking paths from the dead code. In other words, if some dead code in the app invokes an API without any check, PSDroid would report an issue. In fact, this should not be an issue since it is not actually invoked by the app. Another limitation is that PSDroid cannot solve complex patterns when computing the runtime SDK version range for a specific API, such as combined constraints within a single condition.

Threats to Validity. The validity of this study may be subject to some threats. **(1)** The accuracy of API lifetime modeling. PSDroid models API lifetime based on an API document maintained by Google, which contains the definitions of public methods in each SDK version [21]. However, it is found that there may exist inconsistencies between the API document and the Android framework code occasionally. For instance, in the document, the API *isEmpty()* was deleted after SDK version 29, but in the framework code, it is not deprecated [52], which may result in false positives in PSDroid. **(2)** The bias of manual analysis. Since there exists no ground truth real-world dataset for API compatibility issue detection, we manually checked 1,200 code changes to build an up-to-date dataset that might have bias. Similarly, to validate the correctness of issues detected by PSDroid, we sampled 25% compatibility issues for each app and manually checked them for evaluation. Although manual analysis might introduce bias, we mitigate it by three authors cross-validating the benchmark dataset and the experimental results.

VII. Conclusion

In this paper, we propose PSDroid, a path-sensitive semantic analysis approach for automated detection of API compatibility issues in Android apps. PSDroid models the lifetime of framework APIs, extracts relevant API invoking paths with four usage patterns, concludes three types of API compatibility usage issues, and localizes incompatible API usages. Experiment results demonstrate the effectiveness of PSDroid and the superiority over existing tools in reducing false positives and false negatives. 18/30 reported issues are also confirmed by developers.

VIII. Data Availability

The tool, experimental dataset, and results are publicly available at https://github.com/PSDroid2022.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China (Grant No. 62102197, 62102284, 62202245), and the National Key Research and Development Program of China (Grant No. 2018YFB1403400).

REFERENCES

[1] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for Android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 596–607.

[2] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically distilling storyboard with rich features for Android apps," *IEEE Transactions on Software Engineering*, 2022.

[3] J. Steele and N. To, *The Android developer's cookbook: building applications with the Android SDK*. Pearson Education, 2010.

[4] Google. (2022) Android API Levels. [Online]. Available: https://apilevels.com

[5] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the Android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79.

[6] 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.

[7] D. Guilardi, J. Nicácio, B. M. Napoleão, and F. Petrillo, "Are apps ready for new Android releases?" in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, 2020, pp. 66–76.

[8] T. Mahmud, M. Khan, J. Rouijel, M. Che, and G. Yang, "Api change impact analysis for Android apps," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 894–903.

[9] 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*, 2016, pp. 226–237.

[10] 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*, 2018, pp. 153–163.

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

[12] 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.

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

[14] S. Scalabrino, G. Bavota, M. Linares-Vásquez, V. Piantadosi, M. Lanza, and R. Oliveto, "Api compatibility issues in Android: Causes and effectiveness of data-driven detection techniques," *Empirical Software Engineering*, vol. 25, no. 6, pp. 5006–5046, 2020.

[15] H. Xia, Y. Zhang, Y. Zhou, X. Chen, Y. Wang, X. Zhang, S. Cui, G. Hong, X. Zhang, M. Yang *et al.*, "How Android developers handle evolution-induced api compatibility issues: A large-scale study," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 886–898.

[16] T. Mahmud, M. Che, and G. Yang, "Android compatibility issue detection using api differences," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 480–490.

[17] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic Android deprecated-api usage update by learning from single updated example," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 401–405.

[18] S. A. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano, and G. Muller, "Androevolve: Automated update for Android deprecated-api usages," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 1–4.

[19] Y. Zhao, L. Li, K. Liu, and J. Grundy, "Towards automatically repairing compatibility issues in published Android apps," in *The 44th International Conference on Software Engineering (ICSE 2022)*, 2022.

[20] P. Liu, Y. Zhao, H. Cai, M. Fazzini, J. Grundy, and L. Li, "Automatically detecting API-induced compatibility issues in Android apps: a comparative analysis (replicability study)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2022. [Online]. Available: https://doi.org/10.1145%2F3533767.3534407

[21] Google. (2022) APIs Declaration in Android framework base. [Online]. Available: https://github.com/aosp-mirror/platform_frameworks_base/blob/android-11.0.0_r1/api

[22] S. Nielebock, P. Blockhaus, J. Krüger, and F. Ortmeier, "Androidcompass: A dataset of Android compatibility checks in code repositories," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 535–539.

[23] Google. (2022) Calling java from kotlin, mapped types. [Online]. Available: https://kotlinlang.org/docs/java-interop.html#mapped-types

[24] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for Android apps," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 204–215.

[25] (2021) Gpstest. [Online]. Available: https://github.com/barbeau/gpstest

[26] (2021) Gpstest. [Online]. Available: https://github.com/barbeau/gpstest/blob/c1ffd27b536c71d1a459b34aeff7330e05c4df43/GPSTest/src/main/java/com/android/gpstest/util/SatelliteUtils.java

[27] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[28] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[29] Google. (2021) Telegram foss. [Online]. Available: https://github.com/Telegram-FOSS-Team/Telegram-FOSS

[30] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[31] Google. (2022) Colorint, denotes that the annotated element represents a packed color int. [Online]. Available: https://developer.android.google.cn/reference/kotlin/androidx/annotation/ColorInt

[32] ——. (2022) Layoutres, denotes that an integer parameter, field or method return value is expected to be a layout resource reference. [Online]. Available: https://developer.android.google.cn/reference/androidx/annotation/LayoutRes

[33] ——. (2022) Documentation for app developers. [Online]. Available: https://developer.android.google.cn/docs

[34] (2021) Wigle wifi wardriving foss. [Online]. Available: https://github.com/wiglenet/wigle-wifi-wardriving

[35] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for Android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130. [Online]. Available: https://doi.org/10.1145/3468264.3468620

[36] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485533

[37] (2021) Music-player-go. [Online]. Available: https://github.com/enricocid/Music-Player-GO

[38] PSDroid. (2022) Detected issues in music player go. [Online]. Available: https://github.com/enricocid/Music-Player-GO/issues/383

[39] M. P. GO. (2022) Fixing issues in music player go. [Online]. Available: https://github.com/enricocid/Music-Player-GO/commit/c348d2e7a3775181d68cc06bfdc89d8c3e1a15aa

[40] PSDroid. (2022) Detected issues in trigger. [Online]. Available: https://github.com/mwarning/trigger/issues/73

[41] Trigger. (2022) Fixing issues in trigger. [Online]. Available: https://github.com/mwarning/trigger/commit/b03385a69b488cadf9f077f06233118506253f3d

[42] (2021) Trigger. [Online]. Available: https://github.com/mwarning/trigger

[43] (2021) Anki-android. [Online]. Available: https://github.com/ankidroid/Anki-Android

[44] PSDroid. (2022) Detected issues in anki-android. [Online]. Available: https://github.com/ankidroid/Anki-Android/issues/10469

[45] (2021) Amazefilemanager. [Online]. Available: https://github.com/TeamAmaze/AmazeFileManager

[46] PSDroid. (2022) Detected issues in amazefilemanager. [Online]. Available: https://github.com/TeamAmaze/AmazeFileManager/issues/3194

[47] ssynhtn. (2018) Requiresapi vs targetapi -Android annotations. [Online]. Available: https://stackoverflow.com/questions/40007365/requiresapi-vs-targetapi-android-annotations/50578783#50578783
[48] GPStest. (2022) Fixing issues in gpstest. [Online]. Available: https://github.com/barbeau/gpstest/commit/59b8c7e8046e87e9678dfa546c1372c1991d62f5
[49] (2021) Openboard. [Online]. Available: https://github.com/dslul/openboard.git
[50] OpenBoard. (2022) Fixing issues in openboard. [Online]. Available: https://github.com/openboard-team/openboard/commit/23286e0e24ce13e917b92130136164adaa0bb0da
[51] (2021) Dsub. [Online]. Available: https://github.com/daneren2005/Subsonic.git
[52] Google. (2022) Vector defination. [Online]. Available: https://developer.android.google.cn/reference/java/util/Vector?hl=en#isEmpty()