Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps

Lili Wei, Yepang Liu, Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{lweiae, andrewust, scc}@cse.ust.hk

ABSTRACT

Android ecosystem is heavily fragmented. The numerous combinations of different device models and operating system versions make it impossible for Android app developers to exhaustively test their apps. As a result, various compatibility issues arise, causing poor user experience. However, little is known on the characteristics of such fragmentation-induced compatibility issues and no mature tools exist to help developers quickly diagnose and fix these issues. To bridge the gap, we conducted an empirical study on 191 real-world compatibility issues collected from popular open-source Android apps. Our study characterized the symptoms and root causes of compatibility issues, and disclosed that the patches of these issues exhibit common patterns. With these findings, we propose a technique named FicFinder to automatically detect compatibility issues in Android apps. FicFinder performs static code analysis based on a model that captures Android APIs as well as their associated context by which compatibility issues are triggered. FicFinder reports actionable debugging information to developers when it detects potential issues. We evaluated FicFinder with 27 large-scale open-source Android apps. The results show that FicFinder can precisely detect compatibility issues in these apps and uncover previously-unknown issues.

CCS Concepts

•**General and reference** → **Empirical studies;** •**Software and its engineering** → **Software testing and debugging; Software reliability;** *Software performance;* •**Human-centered computing** → **Smartphones;**

Keywords

Android fragmentation, compatibility issues

1. INTRODUCTION

Android is the most popular mobile operating system with over 80% market share [38]. Due to its open nature, a large number of manufacturers (e.g., Samsung, LG) choose to develop their mobile devices by customizing the original Android systems. While this has led to the wide adoption of Android smartphones and tablets, it has also induced the heavy fragmentation of the Android ecosystem. The fragmentation causes unprecedented challenges to app developers: there are more than 10 major versions of Android OS running on 24,000+ distinct device models, and it is impractical for developers to fully test the compatibility of their apps on such devices [4]. In practice, they often receive complaints from users reporting the poor compatibility of their apps on certain devices and have to deal with these issues frequently [7, 53].

Existing studies have investigated the Android fragmentation problem from several aspects. For example, Han et al. were among the first who studied the compatibility issues in Android ecosystem and provided evidence of hardware fragmentation by analyzing bug reports of HTC and Motorola devices [51]. Linares-Vásquez et al. [59] and McDonnell et al. [63] studied how Android API evolutions can affect the quality (e.g., portability and compatibility) and development efforts of Android apps. Recently, researchers also proposed techniques to help developers prioritize Android devices for development and testing by mining user reviews and usage data [55, 62]. Although such pioneer work helped understand Android fragmentation, little is known on the root cause of fragmentation-induced compatibility issues and how developers fix such issues in reality. In addition, existing studies have not fully investigated these issues down to app source code level and hence cannot provide deeper insights (e.g., common issue patterns and fixing strategies) to ease debugging and bug fixing tasks. As a result, there are no mature tools to help developers combat Android fragmentation and improve their apps' compatibility.

To better understand fragmentation-induced compatibility issues in Android apps, we conducted an empirical study on 191 real issues collected from popular open-source Android apps. The study aims to explore the following three research questions. For ease of presentation, we will refer to Fragmentation-Induced Compatibility issues as FIC issues.

- **RQ1: (Issue type and root cause):** *What are the common types of FIC issues in Android apps? What are their root causes?*
- **RQ2: (Issue symptom):** *What are the common symptoms of FIC issues in Android apps?*
- **RQ3: (Issue fixing):** *How do Android developers fix FIC issues in practice? Are there any common patterns?*

By investigating these research questions, we made interesting findings. For example, we found that FIC issues in Android apps can cause both functional and non-functional consequences and observed five major root causes of these issues. Among these root causes, frequent Android platform

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970312

API evolution and problematic hardware driver implementation are two dominant ones. Such findings can provide developers with guidance to help avoid, expose, and diagnose FIC issues. Besides, we observed from the patches of the 191 FIC issues that they tend to demonstrate common patterns. To fix FIC issues, developers often adopted similar workarounds under certain problematic software and hardware environments. Such patterns can be learned and leveraged to help automatically detect and fix FIC issues.

Based on our findings, we designed a static analysis technique, FicFinder, to automatically detect FIC issues in Android apps. FicFinder is driven by an API-context pair model proposed by us. The model captures Android APIs that suffer from FIC issues and the issue triggering contexts. The model can be learned from common issue fixing patterns. We implemented FicFinder on Soot [57] and applied it to 27 non-trivial and popular Android apps. FicFinder successfully uncovered 14 previously-unknown FIC issues in these apps. We reported these issues to the original developers of the corresponding apps. So far, we have received acknowledgment from developers on eight reported issues. Five of them were considered critical and have been quickly fixed. These results demonstrate the usefulness of our empirical findings and FicFinder. To summarize, we make the following major contributions in this paper:

- To the best of our knowledge, we conducted the first empirical study of FIC issues in real-world Android apps at source code level. Our findings can help better understand and characterize FIC issues while shedding lights on future studies related to this topic. Our dataset is publicly available to facilitate future research [23].
- We proposed an API-context pair model to capture the common patterns of FIC issues in Android apps. With this model, we can generalize developers' knowledge and practice in handling FIC issues and transfer such knowledge to aid many software engineering tasks such as automated issue detection and repair.
- We designed and implemented a technique, FicFinder, to automatically detect FIC issues in Android apps. The evaluation of FicFinder on 27 real-world subjects shows that it can effectively detect FIC issues and provide useful information to facilitate issue diagnosis and fixing.

2. BACKGROUND

While Android provides vendors of mobile devices with an open and flexible software infrastructure to quickly launch their products, it complicates the task of developing reliable Android apps over these devices. One of the known complications is induced by the infamous Android fragmentation problem that arises from the need to support the proliferation of different Android devices with diverse software and hardware environments [62]. Two major causes account for the severity of fragmentation.

- *Fast evolving Android platforms.* Android platform is evolving fast. Its API specifications and development guidelines constantly change. As shown in Table 1, the Android devices on market are running very different OS versions from 2.2 to 6.1 with API levels from level 8 to level 23 [20].
- *Myriad device models.* To meet market demands, manufacturers (e.g., Samsung) keep releasing new Android device models with diverse hardware (e.g., different screen sizes, camera qualities, and sensor compositions) and customize their own Android OS variants by modifying the original Android software stack. Device manufacturers make two typical customizations: (1) they implement lower-level system (hardware abstraction layer and hardware drivers) to allow the higher level system (Figure 1) to be agnostic about the lower-level driver implementations that are specific to device models; (2) they modify the higher level system to meet device models' special requirements (e.g., UI style). Such customizations induce variations across device models (see examples in Section 4.1).

Table 1: Android OS version distribution

Version	Codename	API level	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.2%
4.1.x	Jelly Bean	16	7.8%
4.2.x		17	10.5%
4.3		18	3.0%
4.4	Kitkat	19	33.4%
5.0	Lollipop	21	16.4%
5.1		22	19.4%
6.0	Marshmallow	23	4.6%

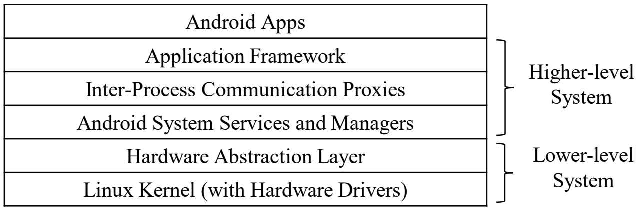

Android Apps
Application Framework
Inter-Process Communication Proxies
Android System Services and Managers
Hardware Abstraction Layer
Linux Kernel (with Hardware Drivers)
Higher-level System
Lower-level System

Figure 1: Android System Architecture [5]

Android fragmentation creates burden to app developers, who need to ensure the apps that they develop offer compatible behavior on diverse device models, which support multiple OS versions. This requires tremendous coding and testing efforts. In reality, it is impractical for developers to exhaustively test their apps to cover all combinations of device models and OS versions. Hence, compatibility issues have been frequently reported by app users [51, 55, 62].

3. EMPIRICAL STUDY METHODOLOGY

This section presents our dataset and analysis method to answer the three research questions RQ1–3.

3.1 Dataset Collection

Step 1: selecting app subjects. To study the research questions RQ1–3, we need to dig into the following data from real-world Android apps: (1) bug reports and discussions, (2) app source code, and (3) bug fixing patches and related code revisions. To accomplish that, we searched for suitable subjects on three major open-source Android app hosting sites: F-Droid [22], GitHub [24], Google Code [25]. We targeted subjects that have: (1) over 100,000 downloads (popular), (2) a public issue tracking system (traceable), (3) over three years of development history with more than 500 code revisions (well-maintained), and (4) over 10,000 lines of code (large-scale). We adopted these criteria because the FIC issues in the subjects such selected are likely to affect a large user population using diverse device models.

We manually checked popular open-source apps on the above-mentioned three platforms and found 27 candidates. Table 2 lists some examples. The table gives the demographics of each app, including: (1) project start time, (2) category, (3) brief description of functionality, (4) user rating, (5) number of downloads on Google Play store, (6) lines

Table 2: Android Apps Used in Our Empirical Study (1M = 1,000,000)

App Name	Project Start Time	Category	Description	Rating	Downloads	KLOC	# Revisions	# Issues	# FIC Issues
CSipSimple [18]	04/2010	Communication	Internet call client	4.3/5.0	1M - 5M	59.2	1,778	2,997	80
AnkiDroid [8]	07/2009	Education	Flashcard app	4.5/5.0	1M - 5M	58.1	8,282	3,002	34
K-9 Mail [28]	10/2008	Communication	Email client	4.3/5.0	5M - 10M	86.7	6,116	7,392	34
VLC [42]	11/2010	Media & Video	Media player	4.3/5.0	10M - 50M	28.3	6,868	544	24
AnySoftKeyboard [10]	05/2009	Tools	3rd-party keyboard	4.3/5.0	1M - 5M	70.7	2,788	1,634	19

Table 3: Keywords used in issue identification

device	compatible	compatibility	samsung
lge	sony	moto	lenovo
asus	zte	google	htc
huawei	xiaomi	`android.os.build`	

of code, (7) number of code revisions, (8) number of issues documented in its issue tracking system. The demographics show that the selected apps are popular (e.g., with millions of downloads) and highly rated by users. Besides, their projects all have a long history and are well-maintained, containing thousands of code revisions.

Step 2: identifying FIC issues. To locate FIC issues that affected the 27 candidate apps, we searched their source code repositories for two types of code revisions:

- The revisions whose change log contains fragmentation related keywords. Table 3 lists these case non-sensitive keywords, including the name of top Android device brands.
- The revisions whose code diff contains the keyword "android.os.build". We selected this keyword because the `android.os.Build` class encapsulates device information (e.g., OS version, manufacturer name, device model) and provides app developers with the interface to query such device information and adapt their code accordingly.

The keyword search returned many results, suggesting that all of our 27 candidate apps might have suffered from various FIC issues. We selected the five apps with the most number of code revisions and manually examined these revisions. Table 2 shows the information of the five apps. In total, 1,082 code revisions were found from the selected five apps. We carefully checked these revisions and understood the code changes. After such checking, we collected a set of 263 revisions, concerning 191 FIC issues. The number of revisions is larger than the number of issues because some issues were fixed by multiple revisions. Table 2 (last column) reports the number of found issues for each app subject. These 191 issues form the dataset for our empirical study.

3.2 Data Analysis

To understand FIC issues and answer our research questions, we conducted the following tasks. First, for each issue, we (1) identified the issue-inducing APIs, (2) recovered the links between issue fixing revisions and bug reports, and (3) collected and studied related discussions from online forums such as Stack Overflow [39]. Second, we followed the process of open coding, a widely-used approach for qualitative research [46], to analyze our findings and classify each issue by its type, root cause, symptom and fixing pattern.

4. EMPIRICAL STUDY RESULTS

4.1 RQ1: Issue Type and Root Cause

Although Android fragmentation is pervasive [51], researchers and practitioners have different understanding of the problem [30]. So far, there is no standard taxonomy to categorize FIC issues. Yet, such a taxonomy is crucial to understand how FIC issues in each category are induced and thereby mechanically detected. This motivates us to manually check the 191 issues in our dataset and construct a taxonomy following an approach adopted by Ham et al. [50]. Specifically, we first categorized the issues into two general types: *device-specific* and *non-device-specific*. The former type of issues can only manifest on a certain device model, while the latter type of issues can occur on most device models with a specific API level. Then we further categorized the two types of issues into subtypes according to their respective root causes. Table 4 presents our categorization results. In total, 112 (59%) of the 191 issues are device-specific and the remaining 79 are non-device-specific.

4.1.1 Device-Specific FIC Issues

Device-specific FIC issues occur when invoking the same Android API results in inconsistent behavior on different device models. These issues can cause serious consequences such as crashes and lead to poor user ratings [55]. We found 112 such issues in our dataset and identified three primary reasons for the prevalence of these issues in Android apps.

Problematic Driver Implementation. Out of the 112 issues, 56 are caused by the problematic implementations of various hardware drivers. The running of many Android apps relies on low-level hardware drivers. Different driver implementations can make an app behave differently across device models. To ensure consistent app behavior over multiple device models, app developers need to carefully adapt the code that invokes those Android APIs, which directly or indirectly depend on low-level hardware drivers. However, there are hundreds of such APIs on Android platform. It is impractical for developers to conduct adequate compatibility test covering all device models whenever they use such an API in their apps. Compatibility issues can be easily left undetected before the release of their apps.

Typical examples we observed are the issues caused by the proximity sensor APIs. According to the API guide, proximity sensors can sense the distance between a device and its surrounding objects (e.g., users' cheek) and invoking `getMaximumRange()` will return the maximum distance the sensor can detect [36]. In spite of such specified API guide, FIC issues often arise from the use of proximity sensors in reality. For instance, on devices such as Samsung Galaxy S5 mini or Motorola Defy, the return value of `getMaximumRange()` may not indicate the real maximum distance the sensor can detect. As a result, the proximity distance is incorrectly calculated, leading to unexpected app behaviors [3]. CSipSimple developers filed another example in its issue 353 [19]. The issue occurred on Samsung SPH-M900 whose proximity sensor API reports a value inversely proportional to the distance. This causes the app to compute a distance in reverse to the actual distance. The consequence is that during phone calls with CSipSimple, the screen would be touch sensitive when users hold their phones against their ears (users may accidentally hang up the call in such cases),

Table 4: Issue types and root causes for their occurrences

Type	Root Cause	Issue Example	# of FIC Issues
Device-Specific	Problematic driver implementation	CSipSimple bug #353	56
	OS customization	Android bug #78377	36
	Peculiar hardware composition	VLC bug #7943	20
Non-Device-Specific	Android platform API evolution	AnkiDroid pull request #130	67
	Original Android system bugs	Android #62319	12

```
  boolean proximitySensorActive = getProximitySensorState();
+ boolean invertProximitySensor =
+         android.os.Build.PRODUCT.equalsIgnoreCase("SPH-M900");
+ if (invertProximitySensor) {
      proximitySensorActive = !proximitySensorActive;
+ }
```

Figure 2: CSipSimple issue 353 (Simplified)

```
  Intent startSettings =
      new Intent(android.provider.Settings.ACTION_INPUT_METHOD_SETTINGS);
  startSettings.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
+ try {
      mAppContext.startActivity(startSettings);
+ } catch (ActivityNotFoundException notFoundEx) {
+     showErrorMessage("Does not support input method settings");
+ }
```

Figure 3: AnySoftKeyboard issue 289 (Simplified)

but would be touch insensitive when the users move their phones away from their ears. CSipSimple developers later tracked down the problem and fixed the issue by adapting their code to specially deal with the Samsung SPH-M900 devices, as shown in Figure 2.

OS customization. 36 of the 112 device-specific issues occurred due to non-compliant OS customizations made by Android device manufacturers. There are three basic types of customizations: (1) *functionality modification*, (2) *functionality augmentation*, and (3) *functionality removal*. In our study, we observed that all these three types of customizations can result in device-specific issues.

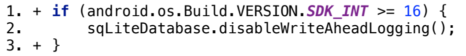
- *Functionality modifications.* Android device manufacturers often modify the original implementations of Android OS to facilitate customization. However, their modified versions may contain bugs or do not fully comply with the original Android platform specifications. This can easily cause FIC issues. For example, several issues in AnkiDroid, AnySoftKeyboard and VLC were caused by a bug in the `MenuBuilder` class of the `appcompat-v7` support library on certain Samsung and Wiko devices running Android 4.2.2, which was documented in Android issue tracker (Issue 78377) [6]. The bug report contains intensive discussions among many Android app developers, whose apps suffered from crashes caused by this bug.
- *Functionality augmentation.* Android device manufacturers often add new features to the original Android OS to gain advantages in market competition. Such functionality augmentation brings burden to app developers, who need to ensure that their apps correctly support the unique features on some devices and at the same time remain compatible on other devices. Unfortunately, this is a non-trivial task and can easily cause FIC issues. For example, after Samsung introduced the multi-window feature on Galaxy Note II in 2012, AnkiDroid and CSipSimple developers attempted to revise their apps to support this feature. However, after several revisions, the developers of CSipSimple finally chose to disable the support in revision 8387675 [18], because this new feature was not fully supported by some old Samsung devices and they failed to find a workaround to guarantee compatibility.
- *Functionality removal.* Some components in the original Android OS can be pruned by device manufacturers if they are considered useless on certain devices. An app that invokes APIs relying on the removed system components may crash, causing bad user experience. For example, issue 289 of AnySoftKeyboard [10] reported a crash on Nook BNRV350. On this device, the input method setting functionality is by default unavailable; hence invoking the API to start input method setting activity crashed the app. Developers fixed this issue in revision b68951a [10] by swallowing the `ActivityNotFoundException` and displaying a hint as shown in Figure 3. Another example is: VLC developers disabled the invocation of navigation bar hiding APIs on HTC One series running an Android OS prior to 4.2 in revision 706530e [42], because navigation bar was eliminated on these devices and invoking related APIs would lead to the killing of the app by the OS.

Peculiar hardware composition. The remaining 20 device-specific issues occurred due to the diverse hardware composition on different Android device models. It is common that different Android device models may utilize different chipsets and hardware components with very different specifications [53]. Such diversity of hardware composition can easily lead to FIC issues, if app developers do not carefully deal with the peculiarities of all hardware variants. For example, SD card has caused much trouble in real-world apps. Android devices with no SD card, one SD card and multiple SD cards are all available on market. Even worse, the mount points for large internal storage and external SD cards may vary across devices. Under such circumstances, app developers have to spend tremendous effort to ensure correct storage management on different device models. This is a tedious job and developers can often make mistakes. For instance, issue 7943 of VLC [43] reported a situation, where the removable SD card was invisible on ASUS TF700T, because the SD card was mounted to a specific point on this device model. Besides ASUS devices, VLC also suffered from similar issues on some Samsung and Sony devices (e.g., revisions 50c3e09 and 65e9881 [42]), whose SD card mount points were uncommon. To fix these issues, VLC developers hardcoded the directory path in order to correctly recognize and read data from SD cards on these devices. Besides SD cards, different screen sizes frequently caused compatibility issues (e.g., revision b0c9ae0 of K-9 Mail [28], revision 6b60085 of AnkiDroid [8]).

4.1.2 Non-Device-Specific FIC Issues

We observed 79 non-device-specific issues in our dataset. Their occurrences were mostly due to Android platform evolution or bugs in the original systems. These issues are independent from device models and can affect a wide range of device models running specific Android OS versions.

Android platform API evolution. Android platform and its APIs evolve fast. On average, there are 115 API updates per month [63]. These updates may introduce new APIs, deprecate old ones, or modify API behavior. 67 non-device-specific issues in our dataset were caused by such API evolutions. For instance, Figure 4 gives an example extracted from AnkiDroid revision 7a17d3c [8], which fixed an issue caused by API evolution. The issue occurred because



```
1. + if (android.os.Build.VERSION.SDK_INT >= 16) {
2.         sqLiteDatabase.disableWriteAheadLogging();
3. + }
```

Figure 4: AnkiDroid pull request 130 (simplified)

the app mistakenly invoked API `SQLiteDatabase.disableWriteAheadLogging()`, which was not introduced until API level 16. Invoking this API on devices with API level lower than 16 could crash the app. To fix the issue, developers put an API level check as a guarding condition to skip the API invocation on devices with lower level APIs (Line 1).

Original Android system bugs. The other 12 non-device-specific issues were caused by bugs introduced in specific Android platform versions. As a complicated system, Android platform itself contains various bugs during development. These bugs may get fixed in a new released version, while still existing in older versions and hence leading to FIC issues. For instance, K-9 Mail encountered a problem caused by Android issue 62319 [6], which only affects Android 4.1.2. On devices running Android 4.1.2, calling `KeyChain.getPrivateKey()` without holding a reference of the returned key would crash K-9 Mail with a fatal signal 11 after garbage collection. A member from the Android project explained that the issue was caused by a severe bug introduced in Android 4.1.2, which was later fixed in Android 4.2, and provided a valid workaround for app developers.

Answer to RQ1: *We observed five major root causes of FIC issues in Android apps, of which the platform API evolution and problematic hardware drive implementation are most prominent. It is challenging to ensure app compatibility on various device models with diverse software and hardware environments.*

4.2 RQ2: Issue Symptom

FIC issues can cause inconsistent app behavior across devices. Such behavioral inconsistency can be functional or non-functional. We discuss some common symptoms below.

168 of the 191 FIC issues are functional and performance issues. Their symptoms resemble those of conventional functional and performance bugs. The vast majority (160) of them are functional bugs. Performance issues account for a minority (8) of the 191 issues found. As aforementioned (Section 4.1), the functional issues can cause an app to crash (e.g., AnkiDroid issue 289 in Figure 4) or not function as expected (e.g., CSipSimple issue 353 in Figure 2). The performance issues can slow down an app. For instance, AnkiDroid developed a simple interface specifically for slow devices in revision 35d5275 [8]. They can also cause an app to consume excessive computational resources. For example, K-9 Mail fixed an issue in revision 1afff1e [28] caused by an old API `delete()` of `SQLiteDatabase`, which did not delete the journal file, resulting in a waste of disk space.

The remaining 23 issues affect user experience. These issues are not bugs but require code enhancements for the diversified features of different devices. Some of these issues lead to UI variations on devices with different screen sizes. For example, AnySoftKeyboard added `ScrollView` to better view the app on small-screen devices in revision 8b911f2 [10]. Other issues require support for device-specific customizations. For example, in revision e3a0067, AnkiDroid developers revised the app to use Amazon market instead of Google Play store on Amazon devices [8]. These issues present a major challenge to Android app developers [4, 53]. Earlier studies show that UI divergence on different-sized screens is a primary reason for bad user reviews of an app [55]. App developers are often forced to fix these issues by enhancing or optimizing their code to provide better user experience.

From the above discussion, we can also observe a key challenge in FIC testing. Less than 10% (15/191) of the issues result in a crash. To effectively detect the majority of issues, we need app-specific test oracles, which are hard to define.

Answer to RQ2: *FIC issues can cause both functional and non-functional consequences such as app crashing, app not functioning, performance and user experience degradation. Their symptoms can be app-specific, making it difficult to define oracles in compatibility testing.*

4.3 RQ3: Issue Fixing

To better understand how developers deal with FIC issues in practice, we studied the patches applied to the 191 issues in our dataset and also analyzed discussions in the related bug reports that we have successfully located. More specifically, we want to understand: (1) whether the patches exhibit common patterns, (2) how complicated the patches are, and (3) how app developers figured out these patches.

First, developers tend to call the patches "workarounds" rather than "fixes", because the root causes of the issues usually reside at the system level instead of the app level (see Section 4.1). Developers mostly resolved the issues by finding a way to work around the problems, making their apps compatible with problematic devices.

Common patching patterns. Second, although FIC issues are often caused by device-specific problems, many patches share common patterns:

- The most common pattern (137 out of 191 patches) is to check device information (e.g., manufacturer, SDK version) before invoking the APIs that can cause FIC issues on certain devices. One typical way to avoid the issues is to skip the API invocations (see Figure 4 for an example) or to replace them with an alternative implementation on problematic devices. This can help fix issues caused by problematic driver implementations, Android platform API evolutions, and Android system bugs.
- Another common strategy (14 out of 191 issues) is to check the availability of certain software or hardware components on devices before invoking related APIs and methods. This is a common patching strategy for FIC issues caused by OS customizations and peculiar hardware compositions on some device models.
- The remaining patches are app-specific workarounds and strongly related to the context of the FIC issues.

Patch complexity. Third, although the concrete patching strategies vary across issues, the patches themselves are usually simple and of small size (several lines of code). The most common pattern of patching only requires adding a condition, which checks device information, before the invocation of issue-inducing APIs. Similarly, checking the availability of certain software/hardware components usually only requires adding a `try-catch` block or a null pointer checking statement. For the 191 issues in our dataset, more than two-thirds of the patches are such simple ones.

Identifying valid patches. Albeit the patches are simple and of small size, this does not imply that it is easy for developers to debug and fix FIC issues. In fact, figuring out the root causes of the issues is a difficult task because it requires deep investigation of the software and hardware environments on Android devices. However, many device firmwares and implementation details of hardware drivers are mostly inaccessible to developers. Without the required information to locate the root causes, app developers derived workarounds by trial and error. In many cases, they asked for volunteers to help test different workarounds or sought advice from other developers whose apps suffered from similar issues. There are also cases in which app developers could not find a valid workaround and gave up on fixing the issues. For example, issue 1491 of CSipSimple [19] was caused by the buggy call log app on an HTC device. However, since the implementation details are not accessible, CSipSimple developers failed to figure out the root cause of the issue and hence left the issue unfixed. Below is their comment after multiple users reported similar issues (Issue 2436 [19]):

> *"HTC has closed source this contact/call log app which makes things almost impossible to debug for me."*

Answer to RQ3: *Locating the root cause of FIC issues is difficult in practice. Whereas, issue fixes are usually simple and demonstrate common patterns: checking device information and availability of software/hardware components before invoking issue-inducing APIs/methods.*

4.4 Implications of Our Findings

Here, we discussed how our findings in Section 4.1 to 4.3 can help developers combat FIC issues.

Compatibility testing. One critical challenge for compatibility testing of Android apps is the *huge search space* caused by fragmentation. Due to limited time and budget, it is impractical for developers to fully test all components of their apps on all device models running all supported OS versions. To address the challenge, existing work proposed approaches to prioritize devices for compatibility testing [55, 62]. However, prioritizing devices alone is not sufficient to effectively reduce the search space. The other two dimensions can still create a large number of combinations: OS versions and app components. The findings and dataset of our empirical study [23] further reduce the search space by providing extra clues about devices, APIs, and OS versions, which are more likely to cause FIC issues. Such information can guide developers to expose FIC issues with less test effort. For example, we discussed earlier that due to the problematic driver implementation, the proximity sensor APIs caused many FIC issues in popular apps. App developers can first spend their limited resources on testing components that use these APIs on the devices with problematic drivers.

Automated detection and repair of FIC issues. By studying the root causes and patches of the FIC issues, we observed that FIC issues recur (e.g., Android issue 78377 discussed in Section 4.1.1) in multiple apps and share similar fixes. It suggests the possibility to generalize the observations made by studying the 191 FIC issues to automatically detect and repair FIC issues in other apps. As discussed in Section 4.3, some app developers fixed the FIC issues in their apps by referencing the patches of similar issues in other apps. This indicates the transferability of such knowledge. To demonstrate the feasibility, we manually extracted 25 rules based on issues in our dataset and designed a static analysis tool to check for rule violations in Android apps. We will show by experiments that the tool can help detect real and unreported FIC issues.

5. ISSUE MODELING AND DETECTION

With our empirical findings, we propose an automated technique named FicFinder to detect FIC issues in Android apps. FicFinder is based on a model that captures issue-inducing APIs and the issue-triggering context of each pattern of FIC issues. It takes an Android app's Java bytecode or `.apk` file as input, performs static analysis, and outputs its detected compatibility issues. In the following, we first explain our modeling of compatibility issue patterns and then elaborate on our detection algorithm.

5.1 API-Context Pair Model

To automatically detect FIC issues, we need to find a formalism by which the issues can be adequately modeled. By studying the root causes and fixes of FIC issues, we observed that many issues exhibit common patterns: *they are triggered by the improper use of an Android API, which we call issue-inducing API, in a problematic software/hardware environment, which we call issue-triggering context.* This observation motivates us to model each pattern of compatibility issues as a pair of issue-inducing API and issue-triggering context, or *API-Context pair* in short. For APIs, we follow their standard definition. For each issue-trigger context, we formulate it using the following context-free grammar.

$$\begin{array}{lcl} Context & \rightarrow & Condition \mid Context \wedge Condition \\ Condition & \rightarrow & Software\_env \mid Hardware\_env \mid API\ usage \end{array}$$

As shown in the grammar, an issue-triggering context is defined as a conjunction of the following conditions of an issue-inducing API's running environment: software environment, hardware environment, or API usage. More specifically, software environment may consist of information related to API level, software configurations and so on. Hardware environment may contain information about device brand, model, hardware composition and so on. API usage describes how the concerned API is used, comprising information about arguments, calling context and so on. According to this model, invoking an issue-inducing API will cause compatibility issues if the conditions in issue-triggering context are satisfied. Let us illustrate this using an example extracted from two issues that affected AnkiDroid (discussed in pull request 128 and 130 [8]).

$$\left[\begin{array}{l}\textbf{API: } \texttt{SQLiteDatabase.disableWriteAheadLogging()}, \\ \textbf{Context: } \text{API\_level} < 16 \wedge \text{Dev\_model} \mathrel{!=} \text{“Nook HD”}\end{array}\right]$$

This simple example models the issue that invoking the API `SQLiteDatabase.disableWriteAheadLogging()` on devices other than "Nook HD" running Android systems with an API level lower than 16 will lead to app crashing [8]. Such API-context pairs can be generated from fixed compatibility issues and used to detect unknown issues.

5.2 Compatibility Issue Detection

API-context pair extraction. 168 of our studied 191 fixed compatibility issues can be modeled as API-context pairs to facilitate the detection of FIC issues. For designing and implementing FicFinder, we selected 25 API-context pairs that are most likely to recur across apps and have the following properties: (1) they are statically checkable in the sense that checking the issue-triggering context does not require information that can only be acquired at runtime, and (2) they can affect apps running on popular devices. We included the latter requirement because some issues in our dataset can only affect apps running on very old device models, which may only affect a tiny fraction of users, and detecting them may not be very useful to developers.

FicFinder algorithm. Algorithm 1 describes FicFinder's analysis process. It takes two inputs: (1) an Android app (Java bytecode or `.apk` file), and (2) a list of API-context pairs. It outputs a set of detected FIC issues and provides information to help developers debug and fix these issues. The algorithm works as follows. For each API-context pair *acPair*, it first searches for the call sites of the concerned API (Line 2). For each call site *callsite*, it then performs two checks (Line 4): (1) it checks whether the usage of API matches the problematic usage defined in *acPair* if there is a condition regarding API usage, (2) it checks whether the configuration of the app matches the problematic configuration defined in *acPair* if there is a condition regarding software configuration. The checks for API usage and software configurations are specific to the issue-triggering context *CTX* defined in each *acPair*. For instance, for an API requiring a lowest SDK version, the algorithm will check the minimum SDK version specified in the app's manifest file. If it is newer than the required SDK version, invocations of this API will not be regarded as issues. If both checks pass, the algorithm proceeds to check other software and hardware environment related conditions defined in *acPair*. These checks require analyzing the statements that the issue-inducing API invocation statement transitively depends on. For this purpose, the algorithm performs an inter-procedural backward slicing on *callsite* and obtains a *slice* of statements (Line 5). More specifically, the backward slicing is performed based on the program dependence graph [47] and call graph. Initially, only the API invocation statement is in the slice. The algorithm then traverses the program dependence graph and call graph, adding statements into the slice. One statement will be added if there exists any statement in the slice that depends on it. This process repeats until the size of the slice converges. Next, the algorithm iterates over all statements in *slice* (Lines 6–9). If any statement in *slice* checks the software and hardware environment related conditions defined in *acPair*, the algorithm conservatively considers that the app developers have already handled the potential compatibility issues (Lines 8–9). Otherwise, the algorithm warns developers that their app may have a compatibility issue and outputs information to help them debug and fix the issues (Lines 10–11). Such information are actionable and include the call sites of issue-inducing APIs and issue-triggering contexts.

Implementation. We implemented FicFinder on top of the `Soot` analysis framework [57]. FicFinder leveraged `Soot`'s program dependence graph and call graph APIs to obtain the intra- and inter-procedural slices. Theoretically, if the slicing results are sound (i.e., each obtained slice contains all the statements that an issue-inducing API invocation statement depends on), our detector will not report false positives. However, in practice, the soundness of slicing results highly hinges on the quality of the statically-constructed call graphs and program dependence graphs. It is well-known that statically constructing precise and sound call graphs and program dependence graphs for Java programs (Android apps are mostly written in Java) is challenging due to the language features such as dynamic method dispatching and reflections [48]. For this reason, FicFinder could report false warnings when analyzing some apps. Our evaluation results show that such false warnings are few.

Algorithm 1: Decting FIC issues in an Android app

```
   Input  : An Android app under analysis,
            A list of API-context pairs acPairs
   Output: Detected compatibility issues
1  foreach acPair in acPairs do
2      callsites ← GetCallsites(acPair.API)
3      foreach callsite in callsites do
4          if MatchAPIUsage(callsite, acPair.CTX) and
           MatchSWConfig(callsite, acPair.CTX) then
5              slice ← BackwardSlicing(callsite)
6              foreach stmt in slice do
7                  issueHandled ← false
8                  if stmt checks acPair.CTX then
9                      issueHandled ← true

10             if issueHandled == false then
11                 report a warning and debugging info
```

6. EVALUATION

In this section, we evaluate FicFinder with real-world Android apps. Our evaluation aims to answer the following two research questions:

- **RQ4: (Issue detection effectiveness):** *Can FicFinder, which is built with API-context pairs extracted from our collected 191 FIC issues, help detect unknown FIC issues in real-world Android apps?*
- **RQ5: (Usefulness of FicFinder):** *Can FicFinder provide useful information for app developers to facilitate the FIC issue diagnosis and fixing process?*

To answer the two research questions, we conducted experiments on 27 actively-maintained open-source Android apps selected from F-Droid database [22].[1] Table 5 reports the basic information of these apps, which include: (1) app name, (2) category, (3) the revision used in our experiments, (4) lines of code, (5) number of stars if the app is hosted on GitHub, and (6) user rating and downloads if the app is available on Google Play store. As we can see from the table, these apps are diverse (covering 10 different app categories), non-trivial (containing thousands of lines of code), and popular on market (achieved thousands to millions of downloads). For the experiments, we applied FicFinder to the latest version of these apps for the detection of FIC issues. We considered those APIs found in the 25 API-context pairs as *issue-inducing*. The experiments were conducted on a MacBook Pro with an Intel Core i5 CPU @2.8 GHz and 8 GB RAM. We now report our experimental results.

6.1 RQ4: Issue Detection Effectiveness

To evaluate FicFinder's performance, we applied it to the latest version of 27 app subjects. We configured FicFinder to report: (1) the call sites of the issue-inducing APIs that can cause compatibility issues (warnings) and (2) the call sites of the issue-inducing APIs where developers have already provided workarounds to avoid potential compatibility issues (good practices). We manually checked all the warnings and the source code of the corresponding apps to categorize

[1] The 27 experimental subjects are different from the 27 subjects used in our empirical study (Section 3.1). We did not use all our 27 empirical study subjects for experiments because some of them are not actively maintained these days.

Table 5: Experimental Subjects and Checking Results

ID	App Name	Category	Latest Revision no.	KLOC	# Stars	Rating	# Downloads	Checking Results			Issue ID(s)
								TP	FP	GP	
1	IrssiNotifier [27]	Communication	ad68bc3	5.2	130	4.8	5K–10K	0	0	0	–
2	1Sheeld [1]	Tools	b49c98a	47.1	46	4.7	10K–50K	2	0	4	4^{b}
3	ConnectBot [16]	Communication	49712a1	23.0	602	4.6	1M–5M	0	2	1	–
4	AnkiDroid [8]	Education	dd654b6	58.1	533	4.5	1M–5M	0	0	11	–
5	AntennaPod [9]	Media & Video	6f15660	65.0	1,142	4.5	100K–500K	2	0	0	1883^{b}
6	Conversations [17]	Communication	1a073ca	39.1	1,655	4.5	5K–10K	3	1	1	1813, 1818^{c}
7	VLC [42]	Media & Video	4588812	28.3	–	4.4	10M–50M	0	0	1	
8	c:geo [14]	Entertainment	5f58482	78.8	672	4.4	1M–5M	1	0	2	5695^{c}
9	Kore [29]	Media & Video	0b73228	29.5	152	4.4	1M–5M	0	0	1	–
10	AnySoftKeyboard [10]	Tools	d0be248	70.7	260	4.4	1M–5M	2	1	2	639^{b}
11	Transdroid [41]	Tools	28786b0	29.9	377	4.4	100K–500K	5	1	0	295^{a}
12	K-9 Mail [28]	Communication	74c6e76	86.7	2,473	4.3	5M–10M	1	0	4	1237^{c}
13	CSipSimple [18]	Communication	fd1e332	59.2	130	4.3	1M–5M	0	0	5	–
14	Telegram [40]	Communication	a7513b3	574.9	4,721	4.2	50M–100M	1	0	10	–
15	WordPress [44]	Social	efda4c9	113.2	1,139	4.2	5M–10M	0	0	14	–
16	OpenVPN for Android [32]	Communication	9278fa4	585.1	278	4.2	1M–5M	0	0	2	–
17	Brave Android Browser [13]	Personalization	65cd91d	160.9	850	4.2	500K–1M	5	0	0	754^{a}, 759
18	PactrackDroid [34]	Communication	1090758	7.3	4	4.2	10K–50K	2	0	0	12^{b}
19	QKSMS [37]	Communication	73b3ec4	63.3	1,071	4.1	100K–500K	1	0	1	474^{a}
20	Open GPS Tracker [31]	Travel & Local	763d1e2	19.4	5	4.1	100K–500K	3	0	0	446
21	BankDroid [11]	Finance	f491574	34.2	240	4.1	100K–500K	0	0	0	–
22	Evercam [21]	Tools	c4476de	26.1	19	4.1	10K–50K	0	0	4	–
23	Bitcoin Wallet [12]	Finance	3235281	18.9	558	4.0	1M–5M	0	0	0	–
24	ChatSecure[15]	Communication	30f54a4	183.8	967	4.0	500K–1M	0	0	1	–
25	iNaturalist [26]	Education	1e837ca	37.0	14	4.0	50K–100K	0	0	15	–
26	ownCloud [33]	Productivity	cfd3b94	49.1	1,467	3.7	100K–500K	0	0	0	–
27	PocketHub [35]	–	a6e9583	32.7	7,735	–	–	18	0	0	997^{b}
28	**Total**	–	–	–	–	–	–	46	5	79	–

"–" means not applicable. Superscript "a" means the issues were acknowledged and developers agreed to fix in future. "b" means the issues have already been fixed. "c" means the issues do not seriously affect the app and developer chose not to fix them. Bug reports of device-specific issues are underlined.

the warnings into true positives and false positives. The results are reported as "TP" and "FP" in Table 5. For good practices, we also report them as "GP" in the table.

Table 5 shows that FicFinder reported 51 warnings and 46 of them are true positives (precision is 90.2%). The true positives were detected by five different API-context pairs. The number of distinct API-context pairs is relatively small compared to the number of warnings detected by them. This indicates that some commonly used issue-inducing APIs were not well handled by app developers. In some apps used in our empirical study, new warnings were reported. It should be noted that the warnings were detected by rules derived from other studied subjects. For example, the warning in AnySoftKeyboard is detected by a rule originally derived from K-9 Mail. This provides evidence for transferability of the knowledge of FIC issues across apps. We manually inspected the remaining false positives and identified two major causes. First, some implicit calling contexts in the programs were missed by FicFinder and thus induced false warnings. For example, the issue-inducing APIs in ConnectBot are transitively called by a system event handler, which was introduced together with the APIs. In such cases, the APIs will never be invoked on a device with an improper API level. Without identifying the event handler's specification, FicFinder would report the call sites of the APIs as warnings. Second, other false positives were mostly caused by the incomplete call graphs and program dependence graphs constructed by `Soot`, which led to missing dependencies in the slices of issue-inducing APIs' call sites.

Despite FicFinder's high precision, it may miss real FIC issues because it was implemented based on a small number of API-context pairs manually extracted from our empirical study. This can be improved by automating the API-context pair extraction process, which we plan to explore in our future work. In addition, FicFinder adopts conservative strategies to match issue-triggering context. Such strategies may filter out possible call sites of issue-inducing APIs, leading to false negatives. However, they ensure the high precision of analysis results, which is important for automated bug detection tools [56].

Besides warnings, FicFinder also found 79 good practices in 17 out of the 27 apps analyzed, covering 16 distinct API-context pairs. This confirms that FIC issues are common in Android apps and developers often fix such issues to improve the compatibility of their apps.

We further reported 45 of our detected 46 true issues to the corresponding app developers for confirmation. There is no public issue tracking system for Telegram, so we did not report the issue detected in this app. In total, we submitted 14 bug reports, whose IDs are provided in Table 5. Each bug report includes all the true warnings related to each issue-inducing API. We also provided issue-related discussions, API guides and possible fixes in the bug reports to help developers diagnose the issues. Encouragingly, eight of the 14 reported issues have been acknowledged by developers. Among the eight acknowledged issues, we observed that: (1) five issues were quickly fixed by app developers (marked with "^{b}" in Table 5) and (2) the remaining three will be fixed according to the developers' feedback. Other than the eight acknowledged ones, three of our 14 submitted bug reports are still pending. The bug reports for the remaining three issues were closed or labeled as "won't fix" by developers as the reported issues do not cause serious consequences to the apps (marked with "^{c}" in Table 5). We will further discuss them in Section 6.2.

Answer to RQ4: *The API-context pairs extracted from existing FIC issues can help FicFinder effectively detect unknown FIC issues in real-world Android apps. FicFinder's analysis precision is high.*

6.2 RQ5: Usefulness of FicFinder

As we mentioned earlier, besides generating warnings, our tool can also provide information including issue related discussions, API guides, and possible fixes to help developers diagnose and fix FIC issues. To study whether such information is helpful, we included them in our submitted bug reports. We learnt developers' opinions on FIC issues by communicating with them and made interesting findings.

First, for four of the five fixed issues, we observed that developers adopted our suggested fixes. For the other issue, the developers adopted an alternative patch, which was semantically equivalent to our suggested one. This shows that the issue diagnosis information and fix suggestions provided by FicFinder are useful to developers. It also indicates that the knowledge and developers' practice learned from our studied FIC issues can be transferable across apps.

Second, we observed one case where the app developers did not have the concerned device model to verify our suggested fix for a device-specific issue (Brave Android Browser Issue 754). Unlike non-device-specific issues, which can often be validated by checking the platform's API guides, verifying device-specific issues and their fixes requires developers to have access to the concerned device models running the affected Android platform versions. Sometimes, this task can be hard. For example, Brave Android Browser developers replied to our bug report:

> *" Unfortunately, I have no chance to test the fix. Could I maybe send you an apk with the fix and you can check it on your problem device? Or maybe you know how can I check it (perhaps using an emulator)?"*

The developers replied that they did not have the affected device models to check our reported issue. Unfortunately, neither did we succeed in finding the affected device models (e.g., Samsung S4 Mini with Android 4.2.2) in Amazon Device Farm [2], which is a widely-used cloud-based platform for testing apps on real phones. This suggests the need for future work to validate detected FIC issues and associated fixes by other means such as crowdsourcing. For example, developers may leverage our reported issue-triggering contexts to identify the likely affected groups of users from user reviews and use the issue-inducing APIs provided by FicFinder to prepare test scenarios and seek these users' further feedback and confirmation.

Third, while our bug reports facilitate developers to identify potential FIC issues in their apps, whether to fix such issues depends on the apps' functionality and purpose. For three of our reported issues, developers chose not to fix them. For instance, we reported to K-9 Mail developers that the behavior of `AlarmManager.set()` has changed since API level 16: using this API on devices with a higher API level does not guarantee the scheduled task to be executed exactly at the specified time. However, the developers considered that such API behavior change would not significantly affect their app's main functionality (i.e., their scheduled tasks do not necessarily need to be executed at a specific time) and did not proceed to fix the issues, but linked our bug report to two existing unresolved bugs. Similarly, developers of c:geo and Conversations (Issue 1818) closed our bug reports on issues of this API due to the same reason. On the other hand, there are also apps whose functionality could be affected by this issue. The app 1Sheeld is an example. The developers confirmed and quickly fixed the reported issue.

Answer to RQ5: *FicFinder can provide useful information to help developers diagnose and fix FIC issues. Future research efforts can be made on addressing the challenges of manifesting FIC issues and verifying fixes on affected device models.*

7. DISCUSSIONS

7.1 Threats To Validity

Subject selection. The validity of our empirical study results may be subject to the threat that we only selected five open-source Android apps as subjects. However, these five apps are selected from 27 candidate apps as they contain most FIC issues. The apps are also diverse, covering four different categories. More importantly, we observed that the findings obtained from studying the 191 real FIC issues in them are useful and helped locate previously-unknown FIC issues in many other apps.

FIC issue selection. Our FIC issue selection strategy may also pose threat to the validity of our empirical study results. We understand that using the strategy, we could miss some FIC issues in dataset preparation. For example, our keywords do not contain less popular Android device brands. To reduce the threat, we made effort to find the code revisions whose diff contains code that checks the device information encapsulated in the `android.os.Build` class. This indeed helped us find issues that happened to many small brand devices (e.g., Wiko and Nook). On the other hand, some existing empirical studies also selected issues by keyword searching in the issue tracking system of open-source apps [51, 61]. However, such strategies are not very suitable for our work due to two major reasons. First, FIC issues usually do not have specific labels and thus are mixed with various other issues. Second, as app developers often suggest users to provide device information when reporting bugs, simply searching device related keywords will return too many results that are irrelevant to FIC issues. With our current strategy, we already obtained 191 real FIC issues, which are sufficient for our study.

Restricted scope of studying FIC issues induced by Android APIs. Another potential threat is that we only studied FIC issues related to Android APIs. While we indeed observed cases where compatibility issues are caused by using third-party library APIs and native libraries, we restricted the scope of our current study to FIC issues related to Android APIs. We made this choice because FIC issues induced by Android APIs likely have a broader impact than those arising from third-party or native libraries.

Evolving Android device market. As Android device market evolves, existing device models or OS versions will be gradually phased out. FIC issues detected by API-context pairs learned from existing issues may eventually get outdated and finally disappear from the market.

Errors in manual checking. The last threat is the errors in our manual checking of FIC issues. We understand that such a process is subject to mistakes. To reduce the threat, we followed the widely-adopted open coding approach [46] and cross-validated all results for consistency.

7.2 Automated API-Context Pair Extraction

In this paper, we manually inspected our collected FIC issues and extracted API-context pairs for issue detection.

This is a labor-intensive process. In reality, it is impractical to manually identify and extract such pairs from a large number of real FIC issues. It would be interesting to automate the process and learn these API-context pairs from highly popular Android apps, which likely have addressed common FIC issues. As discussed earlier, the most common pattern of fixing FIC issues is to check device information before invoking certain APIs. Therefore, it is possible to extract API-context pairs by correlating Android APIs with the code that checks device information. The correlation can be established via static or dynamic code analysis. We will explore this possibility in our future work.

8. RELATED WORK

Android fragmentation has been a major challenge for Android app developers [45, 53]. Recent studies have explored the problem from several aspects. In this section, we discuss some representative ones.

Understanding Android fragmentation. Several studies have been performed to understand problems of Android fragmentation. Han et al. were among the first who explored the Android fragmentation problem [51]. They studied the bug reports related to HTC and Motorola in Android issue tracking system [6] and pointed out that Android ecosystem was fragmented. Researchers have reported various findings related to Android fragmentation. For example, Li et al. pointed out that app usage patterns are sensitive to device models [58]. Pathak et al. reported that frequent OS updates represent the largest fraction of user complaints about energy bugs [65]. Liu et al. observed that a notable proportion of Android performance bugs occur only on specific devices and platforms [61]. Wu et al. studied the effects of vendor customizations on security by analyzing Android images from popular vendors [68]. Later, Zhou et al. proposed an approach to detecting security flaws caused by Android device driver customizations [70]. These studies documented various functional and non-functional issues caused by Android fragmentation, but did not aim to understand the root causes of such issues, which are different from our study.

In addition, the HCI community also found that different resolutions of device displays have brought unique challenges in Android app design and implementation [64, 69]. Holzinger et al. reported their experience on building a business application for different devices considering divergent display sizes and resolutions [52]. In our dataset, we also found that these issues were part of FIC issues on Android platforms and they can seriously affect user experience.

App testing for fragmented ecosystem. Android fragmentation brought new challenges to app testing. To address the challenges, Kaasila et al. designed an online system to test Android apps across devices in parallel [54]. Halpern et al. built a record-and-replay tool to test Android apps across device models [49]. While these papers proposed general frameworks to test Android apps across devices, they did not address the problem of huge search space for FIC issue testing (Section 4.4).

Several recent studies aimed to reduce the search space in Android app testing by prioritizing devices. For example, Vilkomir et al. proposed a device selection strategy based on combinatorial features of different devices [67]. Khalid et al. conducted a case study on Android games to prioritize device models to test by mining user reviews on Google Play store [55]. The most recent work by Lu et al. proposed an automatic approach to prioritizing devices by analyzing user data [62]. It is true that by prioritizing devices, testing resources can be focused on those devices with larger user impact. However, these studies alone are not sufficient to significantly reduce the search space when testing FIC issues. The reason is that the combinations of different platform versions and issue-inducing APIs are still too many to be fully explored. In our work, we addressed the challenge from a different angle by studying real-world FIC issues to gain insights at source code level (e.g., root causes and fixing patterns). By this study, we observed common issue-inducing APIs and the corresponding software and hardware environments that would trigger FIC issues. Such findings can further help reduce the search space during app testing and facilitate automated issue detection and repair.

Android API evolution. Constantly evolving Android platforms cause Android fragmentation. Existing work reported that the chosen SDK version and the quality of the used Android APIs are major factors that affect app user experience [66]. McDonnell et al. investigated how changing Android APIs affect API adoption in Android apps [63]. Linares-Vásquez et al. studied the impact of evolving Android APIs on user ratings and StackOverflow discussions [59, 60]. They have observed that apps using change-prone and fault-prone APIs tend to receive lower user ratings and changes of Android API have motivated app developers to open discussion threads on StackOverflow. These studies focused on understanding the influence of using fast-evolving Android APIs on the development procedure and app qualities, but did not study the concrete symptoms or fixes of FIC issues induced by such API changes. Our work not only covered FIC issues with various root causes but also studied concrete FIC issues so that we can provide new insights and facilitate FIC issue detection and repair.

9. CONCLUSION AND FUTURE WORK

In this paper, we conducted an empirical study to understand and characterize FIC issues in Android apps at the source code level. We investigated 191 real FIC issues collected from five popular open-source Android apps to understand their root causes, symptoms, and fixing strategies. From the empirical study, we obtained several important findings that can facilitate the diagnosis and fixing of FIC issues, and guide future studies on related topics. Based on our findings, we proposed to use API-context pairs that capture issue-inducing APIs and issue-triggering contexts to model FIC issues. With this model, we further designed and implemented a static analysis technique FicFinder to automatically detect FIC issues in Android apps. Our evaluation of FicFinder on 27 large-scale subjects showed that FicFinder can detect many unknown FIC issues in the apps and provide useful information to developers.

In future, we plan to explore the possibility of leveraging crowdsourcing to help developers verify certain device-specific FIC issues and validate possible fixes. We also plan to enhance the detection capability of our technique by mining new FIC issues in popular apps using the proposed API-context framework.

10. ACKNOWLEDGMENTS

This research was supported by RGC/GRF Grant 611813 of Hong Kong and 2016 Microsoft Research Asia Collaborative Research Program.

11. REFERENCES

[1] 1Sheeld: The Arduino Shield. https://github.com/Integreight/1Sheeld-Android-App.

[2] Amazon Device Farm. https://aws.amazon.com/device-farm/.

[3] Android - Proximity Sensor Accuracy - Stack Overflow. http://stackoverflow.com/questions/16876516/proximity-sensor-accuracy/29954988#29954988.

[4] Android Fragmentation Visualized (August 2015). http://opensignal.com/reports/2015/08/android-fragmentation/.

[5] Android Interfaces and Architecture. https://source.android.com/devices/index.html.

[6] Android Open Source Project – Issue Tracker. https://code.google.com/p/android/issues/.

[7] Android's Fragmentation Problem. http://www.greyheller.com/Blog/androids-fragmentation-problem/.

[8] AnkiDroid. https://github.com/ankidroid/Anki-Android.

[9] AntennaPod. https://github.com/AntennaPod/AntennaPod.

[10] AnySoftKeyboard. https://github.com/AnySoftKeyboard/AnySoftKeyboard/.

[11] BankDroid. https://github.com/liato/android-bankdroid.

[12] Bitcoin Wallet. https://github.com/bitcoin-wallet/bitcoin-wallet/.

[13] Brave Android Browser. https://github.com/brave/browser-android.

[14] c:geo. https://github.com/cgeo/cgeo.

[15] ChatSecure for Android. https://github.com/guardianproject/ChatSecureAndroid.

[16] ConnectBot. https://github.com/connectbot/connectbot.

[17] Conversations. https://github.com/siacs/Conversations.

[18] CSipSimple. https://github.com/r3gis3r/CSipSimple.

[19] CSipSimple – Issues. https://code.google.com/archive/p/csipsimple/issues/.

[20] Dashboards | Android Developers. http://developer.android.com/about/dashboards/index.html.

[21] Evercam. https://github.com/evercam/evercam-play-android.

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

[23] FicFinder Project Website. http://sccpu2.cse.ust.hk/ficfinder/.

[24] GitHub. https://github.com/.

[25] Google Code. https://code.google.com/.

[26] iNaturalistAndroid. https://github.com/inaturalist/iNaturalistAndroid.

[27] IrssiNotifier. https://github.com/murgo/IrssiNotifier.

[28] K9-Mail. https://github.com/k9mail/k-9.

[29] Kore. https://github.com/xbmc/Kore.

[30] On Android Compatibility. http://android-developers.blogspot.com/2010/05/on-android-compatibility.html.

[31] Open GPS Tracker. https://github.com/rcgroot/open-gpstracker.

[32] OpenVPN for Android. https://github.com/OpenVPN/openvpn.

[33] ownCloud. https://github.com/hypery2k/owncloud.

[34] PactrackDroid. https://github.com/firetech/PactrackDroid.

[35] PocketHub. https://github.com/pockethub/PocketHub.

[36] Position Sensors | Android Developers. http://developer.android.com/guide/topics/sensors/sensors_position.html.

[37] QKSMS. https://github.com/qklabs/qksms.

[38] Smartphone OS Market Share, 2015 Q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[39] Stack OverFlow. http://stackoverflow.com/.

[40] Telegram. https://github.com/DrKLO/Telegram.

[41] Transdroid. https://github.com/erickok/transdroid.

[42] VLC – Android. https://code.videolan.org/videolan/vlc-android.

[43] VLC Bug Tracker. https://trac.videolan.org/vlc/.

[44] WordPress for Android. https://github.com/wordpress-mobile/WordPress-Android.

[45] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet?(E). In *ASE*, pages 429–440, 2015.

[46] J. W. Creswell. *Qualitative Inquiry and Research Design: Choosing Among Five Approaches (3rd Edition)*. SAGE Publications, Inc., 2013.

[47] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[48] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In *OOPSLA*, pages 108–124, 1997.

[49] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem. In *ISPASS*, pages 215–224, 2015.

[50] H. K. Ham and Y. B. Park. Designing Knowledge Base Mobile Application Compatibility Test System for Android Fragmentation. *IJSEIA*, 8(1):303–314, 2014.

[51] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *WCRE*, pages 83–92, 2012.

[52] A. Holzinger, P. Treitler, and W. Slany. Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones. In *CD-ARES*, pages 176–189. 2012.

[53] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real Challenges in Mobile App Development. In *ESEM*, pages 15–24, 2013.

[54] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala. Testdroid: Automated Remote UI Testing on Android. In *MUM*, page 28, 2012.

[55] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *FSE*, pages 610–620, 2014.

[56] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. Automated Unit Testing of Large Industrial Embedded Software Using Concolic Testing. In *ASE*, pages 519–528, 2013.

[57] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot Framework for Java Program Analysis: A Retrospective. In *CETUS*, 2011.

[58] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng. Characterizing Smartphone Usage Patterns from Millions of Android Users. In *IMC*, pages 459–472, 2015.

[59] 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 *FSE*, pages 477–487, 2013.

[60] 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 *ICPC*, pages 83–94, 2014.

[61] Y. Liu, C. Xu, and S. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*, pages 1013–1024, 2014.

[62] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, and F. Feng. PRADA: Prioritizing Android Devices for Apps by Mining Large-Scale Usage Data. In *ICSE*, 2016.

[63] T. McDonnell, B. Ray, and M. Kim. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *ICSM*, pages 70–79, 2013.

[64] F. Nayebi, J.-M. Desharnais, and A. Abran. The State of the Art of Mobile Application Usability Evaluation. In *CCECE*, pages 1–4, 2012.

[65] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In *HotNets*, page 5. ACM, 2011.

[66] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. What are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications. In *ICSME*, pages 301–310, 2015.

[67] S. Vilkomir and B. Amstutz. Using Combinatorial Approaches for Testing Mobile Applications. In *ICSTW*, pages 78–83, 2014.

[68] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *CCS*, pages 623–634, 2013.

[69] D. Zhang and B. Adipat. Challenges, Methodologies, and Issues in the Usability Testing of Mobile Applications. *IJHCI*, 18(3):293–308, 2005.

[70] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *SP*, pages 409–423, 2014.