# PIVOT: Learning API-Device Correlations to Facilitate Android Compatibility Issue Detection

Lili Wei[‡], Yepang Liu[†*], Shing-Chi Cheung[‡]
[‡]*Dept. of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China*
[†]*Shenzhen Key Laboratory of Computational Intelligence, Southern University of Science and Technology, Shenzhen, China*
Email: {lweiae@cse.ust.hk, liuyp1@sustc.edu.cn, scc@cse.ust.hk}

***Abstract*—The heavily fragmented Android ecosystem has induced various compatibility issues in Android apps. The search space for such fragmentation-induced compatibility issues (FIC issues) is huge, comprising three dimensions: device models, Android OS versions, and Android APIs. FIC issues, especially those arising from device models, evolve quickly with the frequent release of new device models to the market. As a result, an automated technique is desired to maintain timely knowledge of such FIC issues, which are mostly undocumented. In this paper, we propose such a technique, PIVOT, that automatically learns API-device correlations of FIC issues from existing Android apps. PIVOT extracts and prioritizes API-device correlations from a given corpus of Android apps. We evaluated PIVOT with popular Android apps on Google Play. Evaluation results show that PIVOT can effectively prioritize valid API-device correlations for app corpora collected at different time. Leveraging the knowledge in the learned API-device correlations, we further conducted a case study and successfully uncovered ten previously-undetected FIC issues in open-source Android apps.**

***Index Terms*—Android fragmentation, compatibility, static analysis, learning**

## I. INTRODUCTION

The Android market is highly dynamic. To maintain market competitiveness, Android device vendors keep releasing new models running customized OS with unique features. As a result, a lot of device models with different customized OS versions are available at the same time in the market, making the Android ecosystem heavily fragmented. Compatibility issues induced by Android fragmentation have been recognized as a critical challenge in Android app development [38], [42], [51], [60]. These **F**ragmentation **I**nduced **C**ompatibility issues (*FIC issues* for short) can cause Android apps to exhibit inconsistent behavior across different devices. FIC issues can be categorized into two types: non-device-specific ones and device-specific ones [60]. FIC issues are *non-device-specific* if they can be triggered on any device model running a particular Android version, which is denoted by an integer (e.g., 27 for Android 8.1) known as an API level. FIC issues are *device-specific* if they can only be triggered on certain device models running particular API levels. Compared with non-device-specific ones, the search space for device-specific FIC issues is much enlarged with an additional dimension of device models. Hence, device-specific FIC issues are more difficult to detect.

We make two observations on device-specific FIC issues. First, their search space is large, consisting of three dimensions: *device models*, *API levels* and *Android APIs*. There are 24,000+ distinct device models running 10+ different API levels in the market [3], [7]. Each API level supports thousands of APIs. Detecting such FIC issues by checking if each combination of device model, API level and invoked API can induce inconsistent app behaviors is practically infeasible. Second, the search space is dynamic. It continually evolves with the release of new device models and API levels. For example, the number of distinct device models in 2015 was six times as many as in 2012 [25]. In addition, there have been two or more API level upgrades each year, which are commonly customized by Android device manufacturers and shipped with new device models. Detecting device-specific FIC issues in such a huge and evolving search space is challenging. Therefore, developers often realize the existence of FIC issues only after receiving users' complaints.

The observations motivate us to study how to automatically extract the knowledge of FIC issues, which can then be leveraged to effectively reduce the search space for FIC issue detection. In this paper, we focus on ***device-specific FIC issues***. These issues are common but challenging to resolve [60], [61]. While resources provided by Google (e.g., API Guide [2] and emulators) can help developers locate and patch non-device-specific FIC issues, no similar resources are provided for device-specific FIC issues. The root causes of device-specific FIC issues often reside in the proprietary systems customized by the device vendors. These systems are typically closed-source with little public documentation.

Several solutions have been proposed to address FIC issues but few of them tackled the problem of extracting the knowledge of FIC issues to reduce FIC issue search space. Khalid et al. [42], Lu et al. [51], and Vilkomir et al. [58] proposed techniques to prioritize the device models for testing Android apps. However, these prioritization techniques mainly consider the properties of device models but do not correlate the device models with FIC issues. Fazzini et al. [34] proposed DIFFDROID to identify GUI inconsistencies of Android apps when running on different platforms. DIFFDROID focuses on the oracle problem, i.e., how to determine the existence of FIC issues. It does not address the search problem of FIC issues.

In our previous work [60], [61], we found that FIC issues are commonly caused by invoking specific APIs on

* Yepang Liu is the corresponding author of this paper.

specific devices. In other words, *FIC issues mostly correlate issue-inducing APIs with affected device models*. Leveraging this finding, a static analysis technique, FICFINDER, was proposed to locate callsites of APIs that can induce FIC issues on affected device models. FICFINDER takes predefined **API-device correlations** as input for issue detection. In the previous work [60], [61], the API-device correlations were manually extracted from an empirical study. However, this manual approach is impractical because the FIC issue search space evolves quickly. It requires tremendous efforts to manually keep track of API-device correlations of FIC issues that continually arise from new device models. On the other hand, Li et al. [46] proposed a technique to analyze Android framework revision histories to learn patterns of FIC issues caused by Android framework evolution. Their technique relies on the **open-source** Android framework code. However, device-specific FIC issues, the target of this paper, mostly arise from **closed-source** systems customized by device vendors. Without source code, Li et al.'s technique is inapplicable.

In this paper, we propose a technique called PIVOT (A**PI**-De**V**ice c**O**rrela**T**or) to automatically learn the knowledge of device-specific FIC issues in terms of API-device correlations from existing Android apps. In the remaining part of this paper, we will refer to **device-specific FIC issues** as **FIC issues** for ease of presentation unless specified otherwise. Our insight is that *FIC issues are commonly handled by exercising an alternative execution path if the running device matches an issue-triggering model.* In other words, the handling of a FIC issue usually involves a condition that checks device information at runtime. An example of such conditions is given in Line 4 of Figure 1, where Nexus 4 is the issue-triggering model and Lines 4–6 provide an alternative execution path. Based on this insight, PIVOT extracts API-device correlations such as $\langle Parameters.setRecordingHint(boolean), \text{“}Nexus\ 4\text{”}\rangle$ by identifying the device-checking conditions and the API invocations guarded by them. Such learned knowledge can then be leveraged to facilitate FIC issue testing or infer rules/patterns to pinpoint FIC issues in Android apps via static analysis.

PIVOT extracts API-device correlations of FIC issues from a given Android app corpus and prioritizes the extracted correlations based on their likelihood to capture real FIC issues. *An outstanding challenge is to find valid API-device correlations from massive noises.* In our evaluation, the noises account for 97% of our sampled API-device correlations extracted from popular Android app corpora. Existing techniques in API precondition mining [52], [55], [62], [63] remove noises based on the assumption that most API usages in code corpora are correct (i.e., mistakes are rare). However, this assumption does not hold for APIs that can induce FIC issues due to two phenomena. First, since FIC issues continuously evolve, lots of FIC issues, especially the new ones, are likely left undetected in Android apps. Second, common code clones and library usages across Android apps can confuse the mining of FIC issues based on frequency. These cloned code snippets can be duplicated in many apps [32], [45]. These two phenomena can introduce noises that greatly affect the learning accuracy of FIC issues. To address this challenge, we devise a novel ranking strategy to prioritize extracted API-device correlations based on the likelihood that the analyzed apps have handled the corresponding FIC issues and the diversity of the originating occurrences of API-device correlations.

We implemented PIVOT on Soot [43] and ran it to learn API-device correlations from two app corpora, each of which comprises over 2,500 top-ranked apps on Google Play [13] collected at different time. PIVOT achieved a precision of 100% among the top five ranked API-device correlations and over 90% precision among the top ten for both of the corpora. It also successfully identified 49 distinct and valid API-device correlations capturing 17 different FIC issues among the top-ranked API-device correlations. We built an archive accordingly. It comprises the API-device correlations learned by PIVOT and the corresponding discussions collected from online resources. The issue archive is publicly available to Android developers [20] and is expected to grow in future.

To show the usefulness of our learned API-device correlations and the issue archive, we conducted a case study and encoded the knowledge of the FIC issues learned by PIVOT in a state-of-the-art FIC issue detection tool to locate undetected issues in open-source Android apps. We successfully found ten apps that suffer from these FIC issues. We further reproduced and reported our detected issues to the app developers. So far, seven reported issues have been acknowledged, among which four issues have been quickly fixed. This demonstrates that the API-device correlations learned by PIVOT can be used to facilitate FIC issue detection for Android apps. To summarize, we make three major contributions in this paper:

- We proposed and implemented the first technique to learn API-device correlations from existing Android apps and showed that such API-device correlations can be used to facilitate FIC issue detection for Android apps.
- We devised a new ranking strategy that can effectively identify valid API-device correlations capturing real FIC issues. Our evaluation results show that this strategy can significantly outperform traditional ones.
- We archived the FIC issues identified by PIVOT. With the archive, we conducted a case study and successfully revealed previously-unknown FIC issues in ten Android apps, many of which were confirmed or fixed by the app developers.

## II. PROBLEM FORMULATION & MOTIVATION

### A. FIC Issue Pattern & API-Device Correlations

The FICFINDER work [60], [61] found that FIC issues demonstrate patterns: they often arise when certain APIs are invoked on certain device models. We observed that although FICFINDER successfully detected previously-unknown FIC issues, its major limitation is that the patterns used for FIC issue detection were manually extracted from an empirical study. As the Android ecosystem evolves, the applicability of these once effective issue patterns gradually diminishes. For example, among the 25 issue patterns used by FICFINDER [60], 12 are

now outdated because the device models that can trigger the issues have faded out from the market.

This inspires us to develop a sustainable mechanism to keep the knowledge of FIC issues updated with respect to the evolution of Android ecosystem. We note that companies of popular commercial apps are likely to notice and patch FIC issues in their apps early once new issues emerge because of their large user base and rich maintenance resources. We therefore propose to learn API-device correlations from these high quality commercial Android apps to maintain a knowledge base of FIC issues. The learning leverages an observation that *FIC issues are mostly patched by applying workarounds on specific device models* [60], [61]. These workarounds often comprise a conditional statement that checks the runtime device information (e.g., the device model identifier) against predefined values (mostly constants). The runtime device information can be obtained by querying the `android.os.Build` class provided by the Android SDK. Such conditional statements characterize the existence of code snippets that handle FIC issues. For illustration, we show a code snippet that handles a low frame rate issue in camera preview on Nexus 4 in Figure 1. When invoking the camera API on Nexus 4 to take photos, the default frame rate for photo preview is low (6–10 frames per second), making the preview choppy. The patch applies a workaround specifically for Nexus 4 by setting the recording mode hint to true. At Line 4, the code checks the app's runtime environment. If the device model is Nexus 4, the workaround is applied (Lines 4–6). From this example, we observe that the conditional statement (Line 4) encapsulates the information of the device model (Nexus 4) that can trigger the FIC issue. The API call (Line 5) that is dependent on the conditional statement demonstrates a strong correlation with the FIC issue. *By identifying such conditional statements and API calls that are dependent on the conditions, we can recover API-device correlations to characterize FIC issues*.

To learn API-device correlations, PIVOT formulates each correlation as a pair of an API and an identifier for a device model (device identifier for short). In this paper, we focus on learning API-device correlations for Android SDK APIs rather than APIs of third-party libraries. The device identifier of a correlation describes an issue-triggering device model such as the name of the model or the manufacturer. In Figure 1, the API `setRecordingHint` depends on a conditional statement that checks the device identifier against "Nexus 4". In such a case, PIVOT will derive an API-device correlation: $\langle Parameters.setRecordingHint(boolean), \text{“}Nexus\ 4\text{”}\rangle$. Note that PIVOT does not assume that the APIs in API-device correlations are issue-inducing. They can also be issue-fixing ones such as the `setRecordingHint` in our example.

Currently, PIVOT does not include API levels in the extracted API-device correlations. This is because we found that it is uncommon (16.7%) to check both device models and API levels in the code snippets that handle FIC issues according to a previously published FIC issue dataset [60]. The underlying reason may be that many devices' operating system rarely gets major updates after the devices are shipped. As a result, most FIC issues can be patched without checking API levels. Therefore, we include only the information of APIs and device models in our API-device correlations.

```
1.    Camera mCamera = Camera.open();
2.    Camera.Parameters params = mCamera.getParameters();
3.    …
4.    if (Build.MODEL.equals("Nexus 4") {
5.        params.setRecordingHint(true);
6.    }
7.    …
8.    mCamera.setParameters(params);
9.    mCamera.startPreview();
```

Fig. 1. Patch for Camera Preview Frame Rate Issue on Nexus 4

### B. Application Scenarios of API-Device Correlations

API-device correlations can help reduce the search space of FIC issues and save testing efforts. For example, the API-device correlation extracted from Figure 1 suggests the existence of a FIC issue when using the camera of a Nexus 4. With such information, developers can focus on testing the app components that use cameras on Nexus 4. This can help trigger the FIC issue quickly. Otherwise, triggering the issue would require developers to extensively test their apps on a huge number of different devices. Let us analyze the reduction of testing efforts. Assume that developers carefully test their apps using Amazon Device Farm [8], a widely-used online testing platform providing over 200 device models including Nexus 4. According to an existing study [50], a popular Android app contains 50 entry methods on average. Then, app developers may need to test 10,000 ($200 \times 50$) combinations of entry methods and device models to trigger the issue. Such testing efforts are unaffordable for most development teams. As a result, FIC issues would likely be left undetected in the released apps. Comparatively, knowing the correlation between Android camera APIs and Nexus 4, developers can focus on testing a few entry methods that involve camera APIs on Nexus 4 to expose potential FIC issues before releasing their apps.

API-device correlations can also be analyzed to derive rules to automate FIC issue detection and patching. Let us consider the example in Figure 2 again. From the API-device correlation $\langle Parameters.setRecordingHint(boolean), \text{“}Nexus\ 4\text{”}\rangle$, we can infer that Android camera APIs may induce FIC issues on "Nexus 4". We can also infer that a possible patch is to invoke `setRecordingHint` and set the flag to `true`. We can further validate the observations by checking relevant resources online. For example, by a search on Google, we can find concrete evidence (e.g., [6], [14]) showing that camera APIs indeed can cause FIC issues on Nexus 4. Via such analyses, we can build a knowledge base of FIC issues containing their patterns and high-quality patches. The issue patterns can be used as inputs to compatibility analysis tools such as FICFINDER [60]. The issue patches can serve as templates to help fix FIC issues and support future research on automatically repairing FIC issues. To show the feasibility of this application scenario, in our evaluation (Section IV-B), we derived five FIC issue patterns based on learned API-device

```
1.      public boolean onKeyUp(int keyCode, KeyEvent event) {
2.          if ((keyCode == KeyEvent.KEYCODE_MENU) &&
3.                  isLGE()) {
4.              Log.i(TAG, "Applying LG workaround");
5.              openOptionsMenu();
6.              return true;
7.          }
8.          //App-specific code cloned from VLC
9.          View v = getCurrentFocus();
10.         …
11.         return super.onKeyUp(keyCode, event);
12.     }
13.     public boolean isLGE() {
14.         return Build.MANUFACTURER.compareTo("LGE") == 0;
15.     }
16.     protected void onCreate (Bundle savedInstanceState) {
17.         Log.i(TAG, "Activity created");
18.         …
19.     }
```

Fig. 2. Patch for A Crashing FIC Issue on Some LG devices

correlations. With these issue patterns, FICFINDER detected ten previously-unknown FIC issues in popular open-source Android apps. For instance, with the Nexus 4 camera issue pattern, FICFINDER detected three new issues. The developers also quickly fixed these three issues according to our suggested patch (i.e., invoking `setRecordingHint`).

### C. Motivating Example & Overview

**Options menu crash issue on LG devices.** Figure 2 shows a code snippet handling an infamous crashing FIC issue triggered by pressing the physical menu button on some LG devices if the corresponding options menu is customized. The LG's solution (Lines 2–7) helps avoid app crashes by explicitly opening the options menu (Line 5) rather than calling the `onKeyUp()` method of the super class. With the example, we now illustrate the major steps and challenges in learning valid API-device correlations from an Android app corpus.

**Step 1: Extracting API-device correlations.** API-device correlations can be extracted by identifying the conditional statements that check device information and the APIs guarded by these statements. We call these statements *device-checking statements*. A device-checking statement and its guarded API calls may reside in different methods. As shown in Figure 2, the device-checking statement resides in the method `isLGE()` (Line 14), while its guarded API call `openOptionsMenu()` resides in the method `onKeyUp()` (Line 5). Therefore, extracting API-device correlations requires inter-procedural analysis. To capture these correlations, PIVOT first builds an inter-procedural control flow graph for an Android app and then traverses the graph to identify the device-checking statements as well as the API calls depending on them.

**Step 2: Filtering noises.** API-device correlations extracted in the first step contain massive noises because APIs that are irrelevant to FIC issues may also be guarded by device-checking statements as part of app-specific logic. For example, the API `Log.i()` is invoked when the condition *"MANUFACTURER equals 'LGE' "* is satisfied. The API is called for profiling purpose but the inter-procedural analysis in the first step would generate a noisy API-device correlation, $\langle Log.i(), \text{“}LGE\text{”}\rangle$, which is unrelated to any FIC issues. According to our experiments (Section IV), 97% of our sampled API-device correlations generated by the first step are noises.

Filtering such massive noises is a major challenge in learning valid API-device correlations from Android apps. Existing API usage mining techniques [52], [55] cannot effectively help filter such noises due to two reasons. We explain the reasons and present the intuitions of PIVOT's solution below.

**Huge and evolving FIC issue search space.** The existing API usage mining techniques assume that the majority of API usages are correct and adopt conventional statistical metrics (i.e., confidence or support). However, *this assumption may not hold for API-device correlation learning.* Since the search space of FIC issues is huge and evolving, in practice, FIC issues are commonly left unhandled in released apps. As such, many valid API-device correlations, especially those related to new FIC issues, are not subject to high confidence or support.

We observe that the confidence of an API-device correlation within an app helps distinguish valid and noisy correlations. The intuition is that *within the same app: (1) The APIs irrelevant to FIC issues can be invoked at various places without device-checking statements. (2) Callsites of APIs related to the same FIC issues are often guarded by device-checking statements.* Figure 2 contains an example: the irrelevant API `Log.i()` invoked at Line 4, which is guarded by a device-checking statement, is also invoked in another method of the app without any device-checking statements (Line 17). This observation helps distinguish APIs that are relevant to FIC issues from irrelevant ones. Therefore, we propose a metric, *in-app-confidence*, that computes how often an API's invocation is guarded by a specific device-checking statement in an app. As illustrated, this metric helps identify irrelevant APIs that are also often invoked without checking device information.

**Code clones in Android apps.** Simply ranking API-device correlations by in-app-confidence and the frequency of an correlation's occurrences is insufficient because of code clones, which are common in Android apps [32]. Due to code clones, *noisy API-device correlations in cloned code snippets can recur in different apps.* This increases the noises and makes the valid API-device correlations indistinguishable. For example, Lines 9–10 in Figure 2 are app-specific code, which are unrelated to FIC issues. Line 9 invokes the API `Activity.getCurrentFocus()` to get the currently focused view. This line and the code denoted by "..." in Line 10 are cloned from a popular open-source video player, VLC [26]. In our evaluation, we observed that instances of noisy API-device correlations extracted from this cloned code snippet recurred in many apps in our collected app corpora. To mitigate this problem, our ranking strategy also considers the diversity of the occurrences of the extracted API-device correlations. More details of our approach will be presented in the next section.

## III. PIVOT APPROACH

As shown in Figure 3, PIVOT takes a corpus of Android apps as input and outputs a ranked list of API-device correlations. The process consists of two steps. First, the correlation

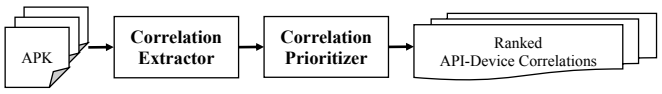


Fig. 3. Overview of PIVOT

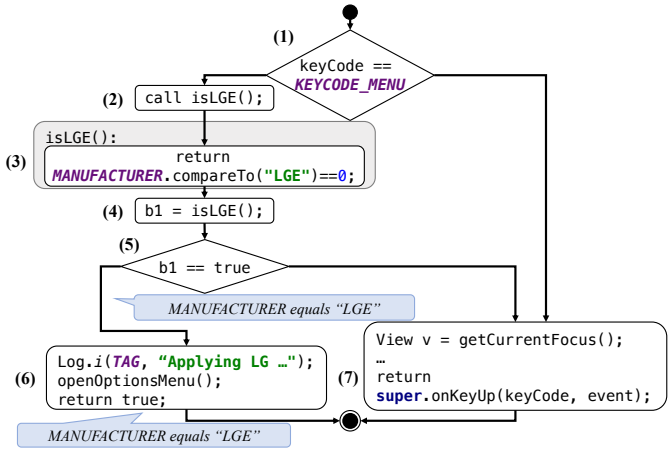


Fig. 4. Inter-Procedural Control Flow Graph of the Code in Figure 2

extractor performs static analysis to extract raw API-device correlations from the input apps. Then the correlation prioritizer ranks the extracted API-device correlations.

### A. Extracting API-Device Correlations

The correlation extractor performs inter-procedural static analysis on an input app in three steps:

**Building inter-procedural control flow graphs.** For each input app, the correlation extractor builds an inter-procedural control flow graph by combining the app's call graph and each method's control flow graph. Figure 4 shows the inter-procedural control flow graph for the code snippet in Figure 2. For ease of presentation, we label each block in the graph with a unique number. Note that Android apps are event-driven. When building inter-procedural control flow graphs, PIVOT does not consider the implicit control flow between different event handlers. As reported in a prior study, it is unlikely that FIC issue patches would cross event handlers [60]. This means that the pieces of code by which an API-device correlation is learned likely reside in the same event handler or its callees. As a result, eliminating control flows between event handlers may not significantly affect API-device correlation extraction.

**Identifying device-checking statements.** The correlation extractor then traverses the inter-procedural control flow graph to identify device-checking statements and evaluates the conditions imposed on each branch. Since device information is encapsulated by the class `android.os.Build`, a conditional statement is considered device-checking if it uses this class. In Figure 4, block 3 contains such a conditional statement (the block is in the method `isLGE()`).

**Computing device constraints for program blocks and deriving API-device correlations.** The correlation extractor then computes the device-related constraints induced by the conditions that should be satisfied to reach each program block. For ease of presentation, we refer to these constraints as device constraints. The device constraint for each block is the disjunction of the device constraints of all paths by which the block can be reached from the application entry points. As discussed, the pieces of code by which an API-device correlation is learned likely reside in the same event handler or its callees. Therefore, PIVOT considers each event handler to be an entry point. Finally, all APIs called in a block are paired with the device identifiers in the device constraint of this block to produce API-device correlations. In Figure 4, the method `onKeyUp()` is an event handler defined in the `Activity` class, and is therefore treated as an entry point for our static analysis. Since there is only one path (1–2–3–4–5–6) reaching block 6 from the entry point, the device constraint is computed as the conjunction of the device constraints associated with the edges on this path (i.e., *MANUFACTURER equals "LGE"*). By pairing up the APIs called in block 6 and the identifier in the device constraint, two API-device correlations are produced: $\langle Log.i(), \text{“}LGE\text{”}\rangle$,$\langle Activity.openOptionsMenu(), \text{“}LGE\text{”}\rangle$.

### B. Prioritizing API-Device Correlations

In this step, the API-device correlations extracted in the first step are prioritized based on their likelihood of capturing real FIC issues. As discussed in Section II-C, existing techniques cannot effectively filter invalid API-device correlations. To prioritize API-device correlations, we propose a new approach that leverages two metrics: *in-app confidence* and *occurrence diversity*. The two metrics are inspired by our observations discussed in Section II-C.

*1) In-App Confidence:* New FIC issues continually arise with the release of new device models and the evolution of Android platforms. These issues may not be commonly and quickly fixed in real-world Android apps. Therefore, the conventional metric *confidence*, which is based on an API-device correlation's popularity over the whole app corpus, cannot effectively prioritize API-device correlations. On the other hand, although FIC issues may not be commonly fixed, we observed that once developers of an app identify a real FIC issue, they tend to modify all callsites of the issue-inducing API within the app to fix the issue. The callsites of APIs irrelevant to FIC issues are mostly not guarded by any device constraints. As such, we propose to use *in-app confidence* (*IAC*) to prioritize API-device correlations. The IAC of an API-device correlation $c$ in an app $A$ is defined as:

$$IAC(A,c) = \frac{\#\ occurrences\ of\ c\ in\ A}{\#\ callsites\ of\ c.api\ in\ A} \quad (1)$$

where $c.api$ is the API of $c$. Intuitively, IAC computes how often the invocation of $c.api$ is guarded by a device-checking statement that checks the app's runtime device model against the device model captured in $c$.

The total in-app confidence metric $w_{IAC}(c)$ for an API-device correlation $c$ is the sum of IACs of all apps in the corpus that contain the API-device correlation:

$$w_{IAC}(c) = \sum IAC(A,c) \quad (2)$$

*2) Occurrence Diversity:* Intuitively, if an API-device correlation recurs in more apps, it is more likely to be valid. However, prioritization of API-device correlations based on their number of occurrences alone can be insufficient. Since code clones are common in Android apps, the API-device correlations extracted from cloned code can appear in many apps and get a high rank even if they have nothing to do with FIC issues (see Section II-C for an example). To mitigate this problem, we propose a new metric *occurrence diversity* to measure the diversity of the apps and methods, within which the instances of an API-device correlation $c$ are found. We denote it as $d(c)$.

We leverage *Shannon index* [56] to measure occurrence diversity. Shannon index was proposed to measure the diversity of the characters in a string and was later widely applied in ecology to measure the species diversity [31], [57]. Shannon index is computed based on the distribution of different groups of entities in a population and is defined as follows:

$$H = -\sum_{i=1}^{n} p_i \log p_i \tag{3}$$

where $p_i$ is the proportional abundance of the $i$-th group in a population. Intuitively, Shannon index is higher if there is a larger number of different groups in the population and the groups of entities are more evenly distributed. Using Shannon Index, we measure an API-device correlation $c$'s occurrence diversity at two levels: *app-level* and *method-level*.

**App-level diversity** measures the diversity of the apps that contain the instances of $c$. We use app names and app company names, which can be extracted from app markets, as identifiers to distinguish different groups of apps.

**Method-level diversity** measures the diversity of methods that contain the API callsites in $c$'s instances. We use package names of the methods' enclosing classes and method control flow structures to distinguish different methods. Package names can be extracted by static analysis but are subject to code obfuscation. To cope with obfuscation, we also measure the diversity of control flow structures using an existing technique [32] that detects code clones in Android apps by projecting the control flow structure of each method to a three-dimensional centroid and calculating distances between these centroids. This technique has been shown to be accurate and scalable. Since it is based on the control flow structures of methods, the technique is robust to code obfuscation. To calculate the control flow structure diversity, we first calculate centroids for all methods that contain the API callsites in $c$'s instances. Then the methods are clustered based on the centroids according to the single-linkage clustering algorithm [36]. Specifically, a method is added to an existing cluster if the distance between its centroid and the centroid of any method in this cluster is smaller than a threshold $\theta$, which is set to 0.05 in our experiment. The output clusters are considered as different control flow structure groups.

After grouping API-device correlation occurrences, we apply Shannon index to calculate the diversity of app name ($H_{app}$), app company name ($H_{company}$), method package name ($H_{package}$), and method centroid ($H_{centroid}$). The overall occurrence diversity of an API-device correlation $c$ is then calculated as follows:

$$\begin{aligned} d(c) = \log(1 \; + \; & H_{app}(c) \times H_{company}(c) \\ & \times H_{package}(c) \times H_{centroid}(c)) \end{aligned} \tag{4}$$

*3) Ranking Score:* The ranking score of an API-device correlation $c$ is then calculated by combining the above metrics:

$$S(c) = w_{IAC}(c) \times d(c) \tag{5}$$

The extracted API-device correlations are ranked based on their ranking scores. If the ranking score of an API-device correlation is higher, it is more likely to capture FIC issues.

## IV. Evaluation

We implemented PIVOT on top of Soot [43]. To evaluate the effectiveness of PIVOT and the usefulness of its learned API-device correlations, we study the following research questions:

- **RQ1 (Effectiveness of PIVOT)** *Can* PIVOT *effectively identify valid API-device correlations of real FIC issues from popular Android apps? Can our proposed ranking score outperform the metrics adopted by existing API precondition mining techniques?*
- **RQ2 (Usefulness of API-device correlations)** *Can the API-device correlations learned from popular apps facilitate FIC issue detection in other apps?*

To investigate the RQs, we conducted two experiments:

- **Experiment I:** To answer RQ1, we ran PIVOT on popular apps collected from Google Play and evaluated the precision of its learned API-device correlations. We compared the results with a baseline approach that adopts the ranking metric of a representative API precondition mining technique proposed by Nguyen et al. [52]. Since FIC issues evolve quickly, we ran PIVOT on two different app corpora collected in 2017 and 2018 and compared the results to evaluate whether PIVOT can effectively identify valid API-device correlations from popular apps at different time.
- **Experiment II:** To answer RQ2, we conducted a case study. We leveraged the API-device correlations learned by PIVOT in the first experiment to detect FIC issues in open-source Android apps and reproduced the detected issues.

The experiments were conducted on a Linux server running CentOS 7.4 with two Intel Xeon E5-2450 Octa Core CPU @2.1GHz and 192 GB RAM.

### A. RQ1: Effectiveness of PIVOT

*1) Experiment I Setup:* In this experiment, we evaluate the effectiveness of PIVOT to prioritize valid API-device correlations. In the following, we discuss the data collection process, baseline approach, ground truth, and evaluation metrics.

**Data collection.** To evaluate whether PIVOT can identify FIC issues that are active at different time, we applied it to two app corpora collected in 2017 and 2018, respectively. To prepare the two corpora, we crawled the APK files of the top 100 free apps for each category on Google Play in November 2017 and June 2018. The first collection contains 2,694 apps

TABLE I
STATISTICS OF OUR APP CORPORA

| | Corpus 2017 | Corpus 2018 |
|---|---|---|
| **# Apps** | 2,524 | 2,765 |
| **Total # classes** | 14,297,551 | 20,023,102 |
| **Total # methods** | 112,181,748 | 108,861,735 |
| **Average rating** | 4.1 | 4.1 |
| **Average # downloads** | 16,000,000+ | 14,000,000+ |

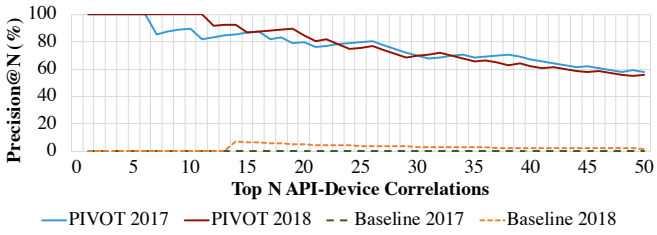


Fig. 5. Precision@N of PIVOT and Baselines

and we were able to run PIVOT on 2,524 of them. The second collection contains 3,054 apps and we were able to run PIVOT on 2,765 of them. Soot crashed when analyzing the remaining apps. The number of apps collected is not in hundreds because some APK files cannot be downloaded due to server errors. The analyzable apps formed our two app corpora. We will call them Corpus 2017 and Corpus 2018 hereinafter. The two app corpora slightly overlap: 853 apps are in both corpora but the app versions are different. We analyzed the APK files of the apps in the two corpora and collected the apps' meta data from Google Play [13]. Table I shows the statistics. As shown in the table, the two app corpora both have tens of millions of classes and over one hundred million methods. The apps are also of high quality and popular. The average rating is 4.1 out of 5. On average, each app received over 16 and 14 million downloads in Corpus 2017 and Corpus 2018, respectively.

**Baseline approach.** We compared PIVOT with a baseline approach adapted from a state-of-the-art API precondition mining technique by Nguyen et al. [52]. The technique learns API preconditions that involve API receivers and parameters. It originally does not support extracting device-related preconditions. To adapt the technique to learn valid API-device correlations, we integrated our Correlation Extractor (Section III-A) with the technique's filtering and ranking components. Specifically, the API-device correlations were first filtered by their support. Those API-device correlations that only occurred once in a corpus were filtered out. The API-device correlations were then ranked by the multiplication of method-level confidence and project-level (i.e., app-level) confidence. App-level confidence of an API-device correlation $c$ is computed as the ratio of the apps that contain $c$'s instances to the apps that contain callsites of $c$'s API. Similarly, method-level confidence of an API-device correlation $c$ is the ratio of the methods containing callsites of $c$'s API that are guarded by statements checking the device identifier modeled in $c$ to the methods that contain the callsites of $c$'s API.

**Ground truth: valid API-device correlations.** We consider an API-device correlation $c$ valid if calling the API in $c$ can trigger or fix FIC issues on the corresponding device model. To establish the ground truth, we manually inspected top 50 API-device correlations of each ranked list produced by PIVOT and the baseline approach. We also randomly sampled 50 API-device correlations from the unranked API-device correlations extracted by Correlation Extractor to show the massiveness of noises. We only manually validated top 50 API-device correlations because without an automated approach, it is unlikely for users to check a large number of API-device correlations produced by PIVOT or the baseline approach. When validating each API-device correlation, we first inspected the originating location of its instances and determined whether the API callsites are dependent on some statements checking the device identifier in the correlation. If yes, we proceeded to use the API signature, the name of API's enclosing class, and the device identifier as keywords to search on Google [12], GitHub [11], and Stack Overflow [23]. An API-device correlation $c$ is then considered valid only if we can find multiple sources (e.g., forum discussions or issue reports) confirming that (1) the API of $c$ can induce inconsistent app behavior on the device model specified in $c$, or (2) the API can be used to address the FIC issues that can be triggered on the device model. Recall the example in Figure 2, the patch suggested by LG developers was to explicitly invoke an API to open the options menu (Line 5) [16], [24]. As a result, the API-device correlation $\langle Activity.openOptionsMenu(), \text{“}LGE\text{”}\rangle$ is considered valid. We published the valid API-device correlations and the corresponding online discussions at our project website [20].

**Evaluation metric.** To measure the effectiveness of PIVOT and the baseline approach in prioritizing API-device correlations, we adopt the metric *Precision@N*, which reports the percentage of valid API-device correlations among the top *N* API-device correlations in a ranked list. Higher Precision@N indicates that the ranking strategy is more effective.

*2) Results of Experiment I:* PIVOT extracted 32,047 and 24,355 API-device correlations from Corpus 2017 and Corpus 2018, respectively. To evaluate the effectiveness of PIVOT, we (1) randomly sampled 50 API-device correlations extracted from each app corpus to show the massiveness of noises, (2) evaluated Precision@N ($N$ = 1, 2, 3, ..., 50) for top 50 API-device correlations in each ranked list, (3) compared the results of PIVOT for the two different app corpora, and (4) compared the results of PIVOT and that of the baseline approach.

**Massiveness of noises.** From the 100 randomly-sampled API-device correlations (50 for each app corpus), we only identified three valid ones that capture real FIC issues. None of these three valid API-device correlations was ranked to top 50 by PIVOT because they either only occurred once in one app or occurred only in cloned code snippets. This shows that noises are massive in the API-device correlations extracted from the app corpora (approximately 97%). Identifying valid API-device correlations is an outstanding challenge.

**Precision@N of PIVOT.** Figure 5 plots the results of Experiment I. The two solid lines show the results of PIVOT for the two app corpora. In both corpora, PIVOT achieves 100% precision among the top five API-device correlations and over 90% precision among the top ten. The precision gradually drops as N grows. This shows that PIVOT can effectively prioritize valid API-device correlations. PIVOT identified 29 valid API-device correlations among the top 50 (58% precision@50) for Corpus 2017 and 28 valid ones among the top 50 for Corpus 2018 (56% precision@50). Among these 29 and 28 valid correlations, eight are overlapped.This means that PIVOT identified a total of 49 distinct valid API-device correlations.

We further studied why PIVOT ranked some invalid API-device correlations among top 50. We identified two major reasons. First, some APIs are commonly used in different apps for specific purposes irrelevant to FIC issues. One typical example is the logging APIs that are widely used for runtime information collection. API-device correlations involving such APIs may receive a high ranking score because the occurrence diversity of such API-device correlations could be high. Second, most other invalid API-device correlations were ranked high due to the limitations of existing constraint solvers and simplifiers. When deriving device constraints for program blocks, we leveraged a popular constraint solver, Choco [53], and a rule-based constraint simplifier, Jbool Expressions [15], to simplify the device constraints. As Boolean Expression minimization is an NP-hard problem [39], our simplified device constraints may not be minimal. Thus, API calls that are not made under any device constraints could be mistakenly included and paired with device constraints.

**Comparison between the results of the two corpora.** As shown in Figure 5, PIVOT achieved similar precision@N for the two app corpora collected at different time. The valid API-device correlations among top 50 for Corpus 2017 and Corpus 2018 only slightly overlap as discussed above. This shows that PIVOT can learn valid API-device correlations from corpora of popular apps at different time.

**Comparison with baseline.** The two dashed lines in Figure 5 show the results of the baseline approach. PIVOT significantly outperforms the baseline approach. The baseline approach failed to rank any valid API-device correlations to top 50 for Corpus 2017 and only identified one valid correlation at the 14th position for Corpus 2018. We further studied why the baseline approach performed poorly. We found that most of its top-ranked API-device correlations involve rarely-used APIs. For example, the top 13 API-device correlations produced by the baseline approach for Corpus 2018 received 1.0 confidence yet none of them are valid. This shows that learning API-device correlations differs from mining API preconditions. General API precondition mining techniques cannot effectively identify valid API-device correlations.

### *B. RQ2: Usefulness of API-Device Correlations*

*1) Experiment II Setup:* We built an archive of 17 distinct FIC issues based on the 49 valid API-device correlations learned by PIVOT. There are fewer distinct FIC issues than API-device correlations because multiple API-device correlations could refer to the same FIC issue. For example, an API that triggers an FIC issue and another API that addresses this FIC issue can result in two valid API-device correlations according to our approach. For each issue, our archive provides: (1) the API-device correlations of the issue learned by PIVOT, (2) the package IDs of the apps from which PIVOT learned these correlations, and (3) the related issue discussions we collected from online resources to validate the API-device correlations. This archive is also publicly available [20].

To investigate the usefulness of the learned API-device correlations, we conducted a case study on open-source Android apps. In the study, we leveraged FICFINDER [60], a static analyzer that detects FIC issues in Android apps, to detect and reproduce undiscovered instances of our archived FIC issues.

**Issue selection**. From our archive, we selected five FIC issues to carry out the case study. We provide videos to demonstrate the inconsistent app behaviors caused by these issues on our project website [20]. We selected these five issues because: (1) the device models that can trigger the issues are available on Amazon Device Farm [1] or WeTest [28] and (2) the inconsistent app behaviors caused by the issues are observable on the online testing platforms. The first criterion enables us to reproduce the later detected FIC issues on online testing platforms. The second criterion allows us to have a clear oracle to determine the occurrence of FIC issues.

**App selection.** To find undiscovered instances of the five FIC issues in real-world Android apps, we collected the latest version of 44 apps on F-Droid [9]. All these 44 apps (1) have at least 50 stars on GitHub [11], (2) have over 500 commits, (3) have at least one push during a five-month period before the case study, and (4) contain at least one callsite of any API related to the five FIC issues. These criteria ensure that our selected apps (1) are popular and well-maintained and (2) could be liable to the selected FIC issues. Note that although the apps are open-source, some of them are also popular on Google Play. For example, Barcode Scanner has received over 100 million downloads on Google Play (Table II).

**Issue detection and reproduction.** For each of the selected issues, we encoded it as a rule in FICFINDER's API-Context Pair format. We then ran FICFINDER using these rules as input to analyze the 44 app subjects. FICFINDER reported warnings for 35 of these subjects. We manually inspected these 35 apps and excluded those that require special hardware/software environments to run (e.g., specific server setups) from this study. As a result, we focused on reproducing 19 detected issues in 19 apps using Amazon Device Farm [1] and WeTest [28].

*2) Usefulness of API-Device Correlations:* Among the 19 detected issues, we successfully reproduced ten in ten different apps. We failed to reproduce the remaining nine due to three major reasons. First, some apps did not exhibit inconsistent behavior as they only use the minor functionalities of the issue-inducing APIs. For example, some apps use the `Camera` API to turn on the flash light to use the phone as a torch. In such cases, the camera preview was not displayed and the issues cannot be observed. Second, FICFINDER generated

TABLE II
CASE STUDY SUBJECTS AND REPORTED ISSUES

| ID | App Name | Category | Latest Revision No. | KLOC | # Stars | Rating | # Downloads | Issue ID(s) |
|---|---|---|---|---|---|---|---|---|
| 1 | Barcode Scanner [4] | Shopping | c8da0c1 | 43.2 | 16,152 | 4.1 | 100M+ | 959[b] |
| 2 | NewsBlur [17] | News & Magazines | 0f0f1f1 | 17.4 | 4,759 | 3.8 | 50K+ | 1084 |
| 3 | Xabber [29] | Communication | 925e996 | 46.2 | 1,509 | 4.1 | 1M+ | 800 |
| 4 | OctoDroid [18] | Productivity | e491d76 | 43.2 | 876 | 4.5 | 100K+ | 821[a] |
| 5 | Simple Task [22] | Productivity | 0572a62 | 4.1 | 288 | 4.7 | 10K+ | 862[a] |
| 6 | Face Slim [10] | - | c62bb57 | 3.5 | 206 | - | - | 319[a] |
| 7 | Simple Camera [21] | Tools | 8aebf00 | 2.2 | 123 | 4.2 | 100K+ | 120[b] |
| 8 | Walleth [27] | Finance | 929216c | 9.1 | 100 | 4.3 | 5K+ | 201[b] |
| 9 | Calendula [5] | Health & Fitness | 39e6e96 | 26.3 | 76 | 4.5 | 1K+K | 92[b] |
| 10 | Open Manga [19] | - | ac3cd27 | 20.7 | 51 | - | - | 47 |

"–" 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.

false positives because it failed to recover issue workarounds that already exist in the apps. Third, we failed to reach the API callsites of several apps because they crashed prematurely.

Table II gives the information of the ten apps, whose FIC issues were successfully reproduced. These apps (1) contain thousands of lines of code (large-scale), (2) cover different app categories (diversified), (3) receive over 50 stars on GitHub and thousands of downloads on Google Play (popular), and (4) are rated at least 3.8 on Google Play (decent quality). We reported the ten reproducible issues to the original app developers to seek their feedback. The issue IDs of our reports are provided in the last column of Table II. Among the ten reported issues, seven have been acknowledged and four were immediately fixed. For example, the issue 92 of Calendula would crash the app when invoking the built-in `DatePicker` API on some popular Samsung devices running Android 5.0 (e.g., Galaxy Note 5). Upon receiving our report, the app developer quickly fixed the issue with a workaround documented in our issue archive [20].

These results show that API-device correlations can help detect FIC issues in Android apps and reproduce the issues on the corresponding issue-triggering device models. The information provided by our FIC issue archive (Section IV-B1) is actionable and can facilitate the detection and diagnosis of FIC issues. Specifically, the device identifiers and APIs specified in API-device correlations can help reduce the search space of FIC issues. With the information, developers can design tests to cover the app components that call FIC issue-inducing APIs and execute these tests on the device models that can trigger the FIC issues to expose potential issues.

## V. DISCUSSIONS

### A. Threats to Validity

**Quality of input app corpora.** PIVOT can only identify valid API-device correlations from apps that contain code snippets handling FIC issues. It is unlikely that such code snippets would exist in apps that are not well-maintained. As such, the quality of app corpora can affect the performance of PIVOT. It is suggested to build an app corpus with popular apps on app stores as we did in our experiments. PIVOT can then extract valid and useful API-device correlations.

**Other code patterns to handle FIC issues.** PIVOT learns API-device correlations from code snippets handing FIC issues, which feature device-checking statements with the use of class `android.os.Build`. Note that there can be other code patterns to handle FIC issues. PIVOT may not discover API-device correlations for all FIC issues. Nevertheless, the practice of checking device information was shown to be common in practice when handling FIC issues [60], [61].

**Imprecise static analysis.** PIVOT performs static analysis on inter-procedural control flow graphs to extract API-device correlations. It is possible that the graphs generated by static analysis are imprecise or unsound [37]. Because of such imperfectness, PIVOT may miss some API-device correlations or generate invalid API-device correlations. However, as PIVOT learns API-device correlations from large app corpora, such problems can be mitigated in PIVOT's final output.

### B. Comparison with FICFINDER

In our previous studies [60], [61], we conducted an empirical study and published the empirical study dataset. We also proposed FICFINDER to detect FIC issues in Android apps. The goals of FICFINDER and PIVOT are different. While FICFINDER aims to detect FIC issues with a given set of predefined patterns, PIVOT aims to learn the knowledge of FIC issues from popular apps. As shown in Section IV-B, the knowledge learned by PIVOT can serve as input to FICFINDER and help detect previously-unknown FIC issues.

We did not apply PIVOT to the apps used by the empirical study in FICFINDER [60], [61] because the empirical study involved only five apps, making it difficult for PIVOT to distinguish valid API-device correlations from noises. Alternatively, we compared the FIC issues in FICFINDER's dataset with those learned by PIVOT. Ten of the 49 valid API-device correlations learned by PIVOT are related to FIC issues in FICFINDER's dataset. Among the other 39 valid API-device correlations that were not included in FICFINDER's dataset, 14 of them concern FIC issues that emerged in 2017 (FICFINDER's dataset was published in 2016). This confirms that PIVOT can identify new valid API-device correlations.

### C. Automated API-Device Correlation Validation

In our experiments, we manually validated API-device correlations. The manual process is subject to errors. To reduce human efforts and provide more reliable validation results, we plan to study automated validation of API-device correlations in future. Particularly, we plan to combine static and dynamic analysis to synthesize test apps from the apps that contain top-ranked API-device correlations. With such test apps, we will be able to exercise the APIs on the device models specified by API-device correlations to validate the correlations.

## VI. Related Work

### A. Android Fragmentation Issues

Several studies have been conducted to understand the problems induced by Android fragmentation. Han et al. [38] mined Android issue tracking system and provided evidence of Android fragmentation. Other studies pointed out that Android fragmentation could induce various consequences. Liu et al. [48] and Hu et al. [40] found that a notable proportion of performance issues and WebView bugs are specific to device models. Li et al. [44] found that app usage patterns are sensitive to device models. Fan et al. [33] found that the compatibility issues induced by fragmentation commonly cause framework crashes. These studies pointed out the issues induced by Android fragmentation but did not propose techniques to address them.

Other studies proposed techniques to mitigate the problems induced by Android fragmentation. For example, several techniques were proposed to prioritize device models for Android app testing. Vilkomir et al. [58] selected Android devices based on combinatorial methods to cover different device characteristics. Khalid et al. [42] proposed to prioritize testing devices based on the user ratings of other apps from the same category. Lu et al. [51] designed PRADA, which prioritizes device models based on the usage data of similar apps. These techniques prioritize device models but are not specifically designed to catch FIC issues. As a result, they cannot provide actionable information to guide FIC issue detection. Fazzini et al. [34] designed DIFFDROID to identify GUI inconsistencies of Android apps when they run on different Android platforms. DIFFDROID compares GUI models of an app when running on different device models, but such comparisons do not help reduce the search space of FIC issues. Huang et al. [41] studied compatibility issues induced by the evolution of callback APIs, whose scope is different from ours. Our previous studies [60] conducted an empirical study of FIC issues in open-source apps and designed a static analyzer FICFINDER to detect FIC issues. FICFINDER requires a given list of FIC issue patterns, which were manually derived. However, manually deriving issue patterns requires intensive human efforts and may not be practical as FIC issues are constantly evolving. Li et al. [46] proposed techniques to automatically learn patterns of compatibility issues caused by Android framework API evolution, whose scope is different from our study (we study device-specific FIC issues). In comparison, PIVOT automatically learns API-device correlations of device-specific FIC issues from a given Android app corpus. Our case study shows that the learned API-device correlations are useful and can help detect previously-unknown FIC issues in real-world Android apps.

### B. Mining API Usage Patterns

Various techniques have been proposed to mine API usage patterns from code repositories. The majority of them aim to mine API co-occurrence relationships. PR-Miner [47] and DynaMine [49] were proposed to mine sets of APIs that frequently co-occur in code repositories. MAPO [62], [63] was among the first techniques to mine frequently-used API sequences. Follow-up studies further improved the pioneering techniques [30], [35], [59]. Most of them rely on mining techniques and adopt light-weight static analysis without analyzing code control dependencies.

Several techniques were proposed to infer API calling preconditions via static analysis and mining code repositories. Ramanathan et al. [54] inferred preconditions that must hold before API invocations from code revision histories. Nguyen et al. [52] mined API preconditions with regard to the API arguments and receivers. These techniques target at mining general API preconditions that may not be relevant to FIC issues. In addition, these techniques leverage traditional filtering metrics such as confidence to identify valid API preconditions. As shown in our evaluation, such metrics cannot effectively prioritize valid API-device correlations. In contrast, PIVOT focuses on learning API-device correlations and features a new and effective ranking strategy to prioritize the correlations.

## VII. Conclusion

In this paper, we proposed the first automated API-device correlation learning approach, PIVOT, to facilitate FIC issue detection. To effectively identify valid API-device correlations, PIVOT performs inter-procedural static analysis to extract API-device correlations and leverages a novel ranking strategy to prioritize them. The evaluation results show that our ranking strategy can effectively identify valid API-device correlations and significantly outperform an existing technique. Based on the learned API-device correlations, we built an archive of FIC issues and further conducted a case study to show the usefulness of API-device correlations. *Our experiment results and other data are published at our project website [20].*

Currently, PIVOT requires human efforts to validate the learned API-device correlations. In future, we plan to study how to automate the validation process by combining program analysis and test synthesis techniques.


## Acknowledgements

This work is supported by the Hong Kong RGC/GRF grant 16202917, MSRA collaborative research fund, the National Natural Science Foundation of China (Grant No. 61802164), the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284), the Program for University Key Laboratory of Guangdong Province (Grant No. 2017KSYS008), Google PhD Fellowship, MSRA PhD Fellowship and the Nvidia academic program.

## REFERENCES


[1] Amazon Device Farm. https://aws.amazon.com/device-farm/, 2018.

[2] Android API Guide. https://developer.android.com/guide/index.html, 2018.

[3] Android Fragmentation Visualized (August 2015). http://opensignal.com/reports/2015/08/android-fragmentation/, 2018.

[4] Barcode Scanner. https://github.com/zxing/zxing, 2018.

[5] Calendula. https://github.com/citiususc/calendula, 2018.

[6] Camera.Parameters.setRecordingHint and aspect ratio - Stack Overflow. https://stackoverflow.com/a/22561537, 2018.

[7] Dashboards — Android Developers. http://developer.android.com/about/dashboards/index.html, 2018.

[8] Device List - Amazon Device Farm. https://aws.amazon.com/device-farm/device-list/, 2018.

[9] F-Droid. https://f-droid.org/, 2018.

[10] Face Slim. https://github.com/indywidualny/FaceSlim, 2018.

[11] GitHub. https://github.com/, 2018.

[12] Google. https://google.com, 2018.

[13] Google Play. https://play.google.com/store/apps, 2018.

[14] Image preview is very slow on Nexus4 - Stack Overflow. https://stackoverflow.com/questions/20252352/image-preview-is-very-slow-on-nexus4, 2018.

[15] jbool expressions. https://github.com/bpodgursky/jbool_expressions, 2018.

[16] LG Developer - NullPointerException in PhoneWindow with custom PanelView. http://developer.lge.com/community/forums/RetrieveForumContent.dev?detailContsId=FC29190703, 2018.

[17] NewsBlur. https://github.com/samuelclay/NewsBlur/, 2018.

[18] OctoDroid. https://github.com/slapperwan/gh4a, 2018.

[19] Open Manga. https://github.com/nv95/OpenManga, 2018.

[20] Pivot Homepage. https://ficissuepivot.github.io/Pivot/, 2018.

[21] Simple Camera. https://github.com/SimpleMobileTools/Simple-Camera, 2018.

[22] Simple Task. https://github.com/mpcjanssen/simpletask-android, 2018.

[23] Stack overflow. https://stackoverflow.com/, 2018.

[24] Stack Overflow - NullPointerException. https://stackoverflow.com/questions/26833242/nullpointerexception-phonewindowonkeyuppanel1002-main/27024610#27024610, 2018.

[25] There are now more than 24,000 different Android devices. https://qz.com/472767/there-are-now-more-than-24000-different-android-devices/, 2018.

[26] VLC – Android. https://code.videolan.org/videolan/vlc-android, 2018.

[27] Walleth. https://github.com/walleth/walleth, 2018.

[28] WeTest. http://wetest.qq.com, 2018.

[29] Xabber. https://github.com/redsolution/xabber-android, 2018.

[30] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *ESEC/FSE*, pages 25–34. ACM, 2007.

[31] Anne Chao and Tsung-Jen Shen. Nonparametric Estimation of Shannon's Index of Diversity When There are Unseen Species in Sample. *Environmental and Ecological Statistics*, 10(4):429–443, 2003.

[32] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *ICSE*, pages 175–186, 2014.

[33] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In *ICSE*, pages 459–472, 2018.

[34] Mattia Fazzini and Alessandro Orso. Automated Cross-Platform Inconsistency Detection for Mobile Apps. In *ASE*, pages 308–318, 2017.

[35] Jaroslav Fowkes and Charles Sutton. Parameter-Free Probabilistic API Mining across GitHub. In *FSE*, pages 254–265, 2016.

[36] John C Gower and Gavin JS Ross. Minimum Spanning Trees and Single Linkage Cluster Analysis. *Applied statistics*, pages 54–64, 1969.

[37] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. In *OOPSLA*, pages 108–124, 1997.

[38] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *WCRE*, pages 83–92, 2012.

[39] Edith Hemaspaandra and Gerd Wechsung. The Minimization Problem for Boolean Formulas. In *FOCS*, pages 575–584, 1997.

[40] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. A tale of two cities: How webview induces bugs to android applications. In *ASE*, pages 702–713, 2018.

[41] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. In *ASE*, pages 532–542, 2018.

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

[43] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot Framework for Java Program Analysis: A Retrospective. In *CETUS*, volume 15, page 35, 2011.

[44] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Felix Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. Characterizing Smartphone Usage Patterns from Millions of Android Users. In *IMC*, pages 459–472, 2015.

[45] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An Investigation into the Use of Common Libraries in Android Apps. In *SANER*, volume 1, pages 403–414, 2016.

[46] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps. In *ISSTA*, pages 153–163, 2018.

[47] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *ESEC/FSE*, pages 306–315, 2013.

[48] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*, pages 1013–1024, 2014.

[49] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *ESEC/FSE*, pages 296–305. ACM, 2013.

[50] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *CCS*, pages 229–240. ACM, 2012.

[51] Xuan Lu, Xuanzhe Liu, Huoran Li, Tao Xie, Qiaozhu Mei, Dan Hao, Gang Huang, and Feng Feng. PRADA: Prioritizing Android Devices for Apps by Mining Large-Scale Usage Data. In *ICSE*, pages 3–13, 2016.

[52] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. Mining Preconditions of APIs in Large-Scale Code Corpus. In *FSE*, pages 166–177, 2014.

[53] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[54] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static Specification Inference Using Predicate Mining. In *PLDI*, pages 123–134, 2007.

[55] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. *TSE*, 39(5):613–637, 2013.

[56] Claude E Shannon. Communication Theory of Secrecy Systems. *Bell Labs Technical Journal*, 28(4):656–715, 1949.

[57] Ian F Spellerberg and Peter J Fedor. A Tribute to Claude Shannon (1916–2001) and a Plea for More Rigorous Use of Species Richness, Species Diversity and the "Shannon–Wiener" Index. *Global Ecology and Biogeography*, 12(3):177–179, 2003.

[58] Sergiy Vilkomir and Brandi Amstutz. Using Combinatorial Approaches for Testing Mobile Applications. In *ICSTW*, pages 78–83, 2014.

[59] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining Succinct and High-Coverage API Usage Patterns from Source Code. In *MSR*, pages 319–328. IEEE Press, 2013.

[60] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *ASE*, pages 226–237, 2016.

[61] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. Understanding and detection fragmentation-induced compatibility issues in android apps. In *TSE*, 2018.

[62] Tao Xie and Jian Pei. MAPO: Mining API Usages from Open Source Repositories. In *MSR*, pages 54–57. ACM, 2006.

[63] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP*, pages 318–343. Springer, 2009.