Automated Detection and Repair of Incompatible Uses of Runtime Permissions in Android Apps

Malinda Dilhara
University of Moratuwa
Sri Lanka
malinda.dilhara@gmail.com

Haipeng Cai
Washington State University
Pullman, WA, USA
hcai@eecs.wsu.edu

John Jenkins
Washington State University
Pullman, WA, USA
john.jenkins@wsu.edu

ABSTRACT

The runtime permission model of Android enhances security yet also constitutes a source of incompatibility issues that impedes the productivity of mobile developers. This paper presents a novel analysis that detects the incompatible permission uses in a given app and repairs them when found, hence automatically adapting the app to the runtime permission model. The key approach is to check and enforce the app's conformance to the runtime permission use protocol through static control flow analysis and bytecode transformation. We implemented our technique as an open-source tool, ARPDroid, and initially evaluated it on 20 incompatible and 3 compatible real-world apps, assisted by manual ground truth and verification. Our results show that ARPDroid achieved 100% detection accuracy, 90% repair success rate, and 91.3% overall adaptation success rate at an average time cost of about two minutes.

KEYWORDS

Android, runtime permission, incompatibility, detection, repair

1 INTRODUCTION

Among other means, the permission system plays a critical role in Android security. Prior to Android 6.0 (*Marshmallow*), the user of an app grants permissions that the app asks for at installation time. Once the app is installed, it will be able to access all permitted resources without further permission checking or request at runtime [11]. The user will thus lose control of permissions until removing or reinstalling the app. This model, referred to as *static permission mechanism*, has led to standing permission related security threats to Android users [17].

To enhance its security related to app permissions, Android has moved from the static permission mechanism to a runtime permission model since 6.0 (API level 23 and forward, noted as *new platforms*). With the new, runtime model, users are privileged to revoke previously granted permissions or grant permissions any time after app installation. Meanwhile, Android apps are required to check if permissions are still available for invoking an API that needs the permissions at the time of the API invocation [13] and, if not, request missing permissions before calling the API. Apps with targetSDKversion≥23 (noted as *new apps*) that do not implement these checks and requests properly will crash (at the first exercised callsite of the API that needs permissions not granted when the API is invoked). For Apps with targetSDKversion<23 (noted as *legacy apps*), new platforms allow them to be used in a degenerated manner (i.e., as in an older platform thus losing the benefits of the enhanced security) or crash them if the user revoked permissions required by an invoked API at runtime. As such, both legacy and new apps face compatibility issues that either cost security or usability, due to Android's move to the new permission mechanism.

Two potential solutions exist, which may mitigate these compatibility issues. The first is to drop the legacy apps themselves, and develop substitutes from scratch (or possibly on top of reusable parts from the corresponding legacy apps). Apparently, this is the most straightforward, yet also the most costly, approach which would cause large resource waste thus may not be practically acceptable. Alternatively, developers could manually go through the source code of legacy apps, check all callsites of each invoked method that is dependent on runtime permissions, and make changes if necessary to ensure their compliance with the new permission model (e.g., guarding each callsite with permission check and/or request [7, 10]). This approach, however, is subject to excessive human efforts, in addition to being error-prone.

Recently, a few tools/libraries [14–16] appeared to help app developers in dealing with the runtime permission model when *developing new apps*. For example, PermissionsDispatcher [15] facilitates the use of runtime permissions in apps by allowing developers to implement the permission checks and requests using simple annotations (instead of using the checking/requesting APIs directly). However, these tools do not address the problem of *migrating legacy apps* to new platforms. Also, they require developers to change the source code and rebuild the app, which largely impedes their use for legacy app migration since the source code may not be available. (In reality, the source code of Android apps is commonly unavailable, except to the original app developers.)

In this paper, we propose a fully automated solution that helps mobile developers deal with runtime permission related compatibility issues. Our technique detects such issues in a given app and fixes them where they are identified, hence automatically adapting the app to platforms featured with the runtime permission mechanism. Our approach realizes the adaptation through static bytecode analysis and transformation without accessing the source code of the app. Also, it does not rely on any code annotations, nor does it involve other manual efforts by developers during the adaptation. The proposed technique focuses on checking against and repairing incompatible uses of runtime permissions, without changing any other aspects of the app's semantics.

We have implemented our technique as an open-source tool ARPDroid [6] (Adaptation to Runtime Permission for anDROID). As a preliminary evaluation of our approach, we randomly selected 23 popular, real-world Android apps from Google Play and applied ARPDroid to each of them. Our results with respect to these benchmarks show promising effectiveness and practical efficiency of our technique. ARPDroid correctly detected all these benchmarks as incompatible or not, with 100% detection accuracy. Also, ARPDroid successfully transformed 18 out of 20 incompatible apps such that the resulting apps work normally on a new platform of Android (version 6.0) with respect to extensive manual inputs. The overall

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197255

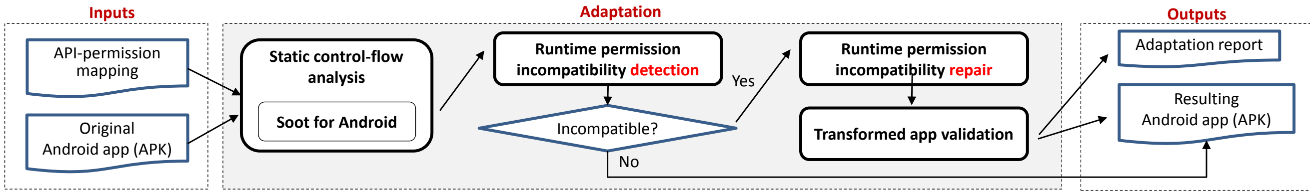

Figure 1: Overview of our approach to detecting and repairing Android app runtime permission incompatibilities.

adaptation success rate was 91.3%. The entire analysis by ARPDroid on these benchmarks took 133 seconds on average. These results suggest that our solution offers a promising automated support for mobile developers to avoid permission-induced compatibility issues, and hence help increase their app development productivity.

In sum, this paper makes the following main contributions.

- We proposed an automated solution to detecting and repairing incompatible runtime permission uses in Android apps without accessing or changing their source code.
- We developed and evaluated an open-source tool that implements our technique, and hence demonstrated the promising effectiveness and practical scalability of our approach.

2 TECHNIQUE

We first give an overview of the workflow of our approach, and then present the details of core technical components. We also discuss the limitations of our approach with respect to the current design.

2.1 Overview

Figure 1 depicts the overall process flow of our approach. The technique takes as inputs the app under analysis, as well as an API-permission mapping which gives the list of permission-dependent APIs along with the permissions each API depends on. The static control flow analysis identifies the program points where permission uses need to be checked (and potentially to be repaired). This analysis is built on the Soot framework for Android [4].

With the results of the control flow analysis, the technique determines if the given app is compatible with the runtime permission mechanism through a dedicated *detection* module. If the app is detected as compatible, the analysis will abort and simply output the original app. If the app is detected as incompatible (i.e., containing any incompatible permission uses with respect to the runtime permission model), the next step repairs all the incompatible permission uses found by the detection module through bytecode transformation. The transformed app is then validated as regards to whether it is compatible. If the validation succeeds, the transformed app is produced as the primary component of the technique's outputs. The other output is a brief adaptation report that indicates whether the adaptation was successful or not, along with relevant logs (e.g., error information or the list of transformed methods). In the cases of validation failure, the original app APK is outputted.

2.2 Static Control Flow Analysis

To enable detection and repair of incompatible permission uses, we start with a static control flow analysis which first constructs the call graph of the input app. Due to their framework-based and event-driven nature, call graph construction for Android apps needs to model the lifecycle of app components (e.g., `Activity` and `ContentProvider`) and analyze control flows induced by callbacks. To that end, we construct a dummy `main` method to emulate the lifecycle and perform an iterative callback analysis as in FlowDroid [4].

Based on this call graph and the (intraprocedural) control flow graph (CFG) of each method in the app, the static analysis continues with localizing incompatible permission uses, using the input API-permission mapping. To identify these locations, our analysis performs a forward traversal on the call graph (from the dummy `main` method). Whenever a permission-dependent API is encountered, the analysis starts a backward traversal on the call graph until the *permission-responsible caller* (PRC) of the API is reached. We define the PRC of a method as the closest caller of the method that is defined in the user code of the app but not in an inner class. The PRC of an API x is thus found through a backward depth-first search (starting from x) on the call graph. For better scalability, our analysis only deals with the incompatibility issues in user code (i.e., code written by the app developer), assuming these issues exist only in this code layer. As expected, each of the permission-dependent APIs invoked in the app may have more than one PRC. These PRCs essentially define the scope (search space) of the incompatibility detection and repair that follow the static control flow analysis.

2.3 Permission Incompatibility Detection

With the results of the static control flow analysis, our approach detects whether the given app has permission-induced incompatibility issues in two steps as follows. Only when it detects the app as possibly incompatible, will the technique proceed with repair. The input app can be either a *legacy* app or a *new* app.

Step 1. The detection algorithm begins with parsing the manifest file contained in the app APK [9]. If the app has a *targetSDKversion* < 23 as declared in the manifest, the algorithm immediately concludes with the decision that the app needs to be repaired (in order to run on new platforms): prior to API level 23, the static permission mechanism was enforced by Android. Otherwise, the algorithm further checks if the app declares to use any dangerous permissions [12] in the manifest. The rationale is two-fold: first, Android divides all permissions into two groups, *normal* and *dangerous*, with normal permissions automatically granted by the system; second, the app or its users can only possibly use, request, grant, or revoke dangerous permissions *that are declared in the manifest*. Thus, if an app does not declare any dangerous permissions in its manifest, it is unlikely to have incompatible permission uses and no further analysis (i.e., repair) is necessary.

Step 2. If the above step results in a positive decision (i.e., the app is likely to be incompatible), the algorithm continues with code-level checking against incompatible permission uses. Specifically, it iterates over all the PRCs found by the static control flow analysis and verifies (1) every permission-dependent API callsite in each PRC is *dominated* by the true branch of a permission check (by invoking the system method *ContextCompat.checkSelfPermission*), and (2) the false branch of the check is *post-dominated* by permission requests for all the permissions required by the API (by invoking the system method *ActivityCompat.requestPermissions*). The app is regarded as compatible only if both (1) and (2) are verified as true; otherwise, the app is detected as incompatible.

Moreover, a permission-dependent API callee of a PRC is considered incompatible if the API fails the verification of (1) and/or (2). A PRC is considered incompatible if it contains at least one permission-dependent API callee that is incompatible.

ALGORITHM 1: Runtime permission incompatibility repair

Input: P - Android app for repair, $\mathcal{M}$ - API-permission mapping
Output: P' - repaired app

1 let $\mathcal{L}$ be the list of incompatible PRCs in P
2 **foreach** *PRC* $m \in \mathcal{L}$ **do**
3   let G be the CFG of m
4   let A be the list of incompatible API callsites in m
5   **foreach** permission-dependent API callsite $cs \in A$ **do**
6         $Perms = \mathcal{M}[cs.callee]$
7   create an empty conditional C
8   **foreach** permission $perm$ *in* $Perms$ **do**
9   create a predicate b asserting $perm$ is granted
10             $C = C \wedge b$
11   **end**
12   **if** cs is not dominated by the true branch of C on G **then**
13   insert C $s.t.$ the true branch of C dominates cs
14   **end**
15   create a permission-requesting callsite *prcs*
16   **if** *prcs* is not post-dominated by C*'s* false branch on G **then**
17   insert *prcs* $s.t.$ it is post-dominated by C's false branch
18   **end**
19   **end**
20   create a permission response handler PRH $w.r.t$ $Perms$
21   **if** PRH is not a member of the enclosing class c of m **then**
22   insert PRH as a new member method of c
23   **end**
24 **end**
25 repackage the changed code to P' and sign it

2.4 Repairing Incompatible Permission Uses

For an app that is detected as incompatible, our technique proceeds with an attempt to automatically repair all the incompatible uses found during the detection, such that the resulting app can function properly while taking advantages (e.g., the enhanced permission security) of new platforms. The repair algorithm, as outlined in Algorithm 1, works by enforcing the two verification rules (1) and (2) above that our technique checks against during detection.

The algorithm uses the list of incompatible PRCs (line 1) and the callsites of incompatible APIs in each incompatible PRC (line 4) that both resulted from the detection algorithm. For each of such callsites (lines 5–19), the algorithm inserts a check against all the permissions on which the API called at the callsite depends (lines 6–14) if there was no such a check properly placed before. Each predicate (line 9) is of form *ActivityCompat.checkSelfPermission(...)==PackageManager.PERMISSION_GRANTED*. The code transformation ensures that the original callsite will fall in the true branch of the check (i.e., the API will be called when all of the permissions required are found already granted). If the check fails (conditional C evaluated as false), a call for requesting all the required permissions will be inserted if there was no such a call properly placed before (lines 15–18). After all incompatible APIs in a PRC are repaired, the algorithm ensures there is a permission request response handler (*onRequestPermissionsResult*, which will be invoked by the platform when the permission-request dialog is closed) included in the class that encloses the PRC (lines 20–23)—in the case of this class being an inner class, the higher level ancestor class will be used instead. At this stage, our technique would create such handlers that simply delegate the event handling to the superclass (i.e., invoking *super.onRequestPermissionsResult*). Finally, the transformed app P' is packaged and signed as the return result (line 25).

Repair validation. The transformed app is validated by rerunning the detection algorithm on it. The validation passes if it is detected as compatible. We are adding a further validation step through dynamic analysis: running the repaired app on a new platform against automatically generated inputs, and then analyzing the execution log to determine the runtime permission compatibility.

2.5 Implementation and Limitations

We implemented our technique as a tool, ARPDroid, based on Soot [3] while leveraging relevant analysis facilities (e.g., lifecycle modeling and callback analysis) in FlowDroid [4] to build the call graph. The API-permission mapping is generated using PScout [5]. ARPDroid provides flexible options allowing users to choose using the detection or repair feature only. The tool has been made available as an open-source project at

https://bitbucket.org/malindadoo/arpdroid

When required permissions are found not granted yet, currently ARPDroid inserts code to directly request those permissions without considering showing additional rationale to the user (by invoking *ActivityCompat.shouldShowRequestPermissionRationale*). Also, the strategy dealing with permission request response is currently simplified, without creating a response handler specific to the permissions being requested. Our current detection algorithm does not check such handlers either, and will miss incompatible permission uses with obfuscated APIs. A main implementation limitation lies in the dependence of our tool on the capability of Soot and FlowDroid: ARPDroid would not be able to handle apps that cannot be successfully processed by these underlying utilities (e.g., failure in bytecode parsing and manipulation or call graph construction).

3 EVALUATION

Methodology. We applied our tool to 23 real-world Android apps of varying sizes and functionality categories that are randomly chosen from Google Play, as listed in Table 1 (the first two columns). For each benchmark, we manually produced the ground-truth compatibility with respect to the runtime permission model (the third column). The benchmark selection rationale for this preliminary evaluation is that we intended to test our technique on 20 incompatible apps to assess its detection and repair capabilities; we also wanted to check if it correctly recognizes compatible apps, for which we intended to have 3 benchmarks for a sanity check. We then ran ARPDroid to detect whether the app is incompatible or not (the fourth column) and, if it was, proceeded with the incompatibility repair step. Each repaired app was initially validated by running the detection algorithm again (see § 2.4).

Next, for any app produced by ARPDroid, either repaired or simply carried over after being detected as compatible, we manually verified whether the app is indeed compatible with the runtime permission model. We ran the app on an Android emulator [8] that has Android 6.0 (API level 23) installed, and then extensively navigated the app with manual inputs (especially permission-dependent operations and permission granting/revoking) to check if the app functions normally on the new platform. If the resulting app passed the manual verification, we considered that the adaptation for this app *succeeded* and otherwise *failed* (the fifth column of Table 1). In the case of successful adaptation, the repair was considered successful if the original app was incompatible. Our experiments were performed on a machine with an Intel(R) Core (TM) i5 2.00GHz processor and 8GB DDR3 RAM. The emulator was given 4GB RAM.

Results and discussion. As shown in Table 1, for the 20 benchmarks that are *actually* incompatible, ARPDroid detected them all correctly as incompatible, and successfully repaired 18 of them. The repair success rate was 90%. We inspected the two unsuccessful cases and confirmed that the causes were that the underlying Soot

Table 1: Detection and Adaptation Effectiveness and Efficiency of Our Technique

Benchmark (package name)	Size (MB)	Ground truth	Detected as	Adaptation	Total time (s)
yogi.corporationapps.telescope.bigzoomhd	2.81	incompatible	incompatible	succeed	123.4
photo.album.galleryvault.photogallery	2.79	incompatible	incompatible	succeed	137.0
com.flashlight017.app	1.13	incompatible	incompatible	succeed	174.8
internet.signal.speed.booster	1.52	incompatible	incompatible	succeed	263.6
com.softwego.applock	3.18	incompatible	incompatible	failed	323.6
com.huawei.netinfo3d	41.53	incompatible	incompatible	succeed	178.8
com.sand.airdroid	21.92	incompatible	incompatible	succeed	220.1
com.google.android.launcher	14.56	compatible	compatible	succeed	143.4
com.bloketech.lockwatch	0.23	incompatible	incompatible	succeed	50.3
com.jvckenwood.ao2.kenwood.musicplay	0.39	incompatible	incompatible	succeed	45.7
bb.andry.hack	0.91	incompatible	incompatible	succeed	89.8
com.jvckenwood.ao2.jvc.musicplay	0.92	incompatible	incompatible	succeed	61.3
com.facebook.orca	47.00	compatible	compatible	succeed	63.7
ur.control.television.rimote.toolss	1.22	incompatible	incompatible	succeed	66.9
connection.stabilizer.powersignals	1.45	incompatible	incompatible	succeed	104.7
com.barcode.home.nga	3.18	incompatible	incomplete	failed	157.8
com.aliengod.zoom	1.50	incompatible	incompatible	succeed	87.8
com.latestnewappzone.autoflashoncallsms	1.82	incompatible	incompatible	succeed	138.8
com.miragestack.secret.voice.recorder	2.31	incompatible	incompatible	succeed	148.9
com.piggy.myfiles	2.35	incompatible	incompatible	succeed	148.9
com.geekslab.screenshot	2.59	incompatible	incompatible	succeed	154.5
com.skype.raider	36.00	compatible	compatible	succeed	52.6
com.mantishrimp.salienteyeremote	4.52	incompatible	incompatible	succeed	122.1
overall	**average=8.50**		**Accuracy=100%**	**success rate=91.3%**	**average=133.0**

failed to process (parse/repackage) the original app code or crashed during call graph construction. The detection, however, still (trivially) succeeded on these two apps based on their manifest files (detection **step 1**). The three compatible apps were all correctly detected (as compatible) as well. Thus, the detection algorithm worked perfectly well on these benchmarks, for a 100% precision and 100% recall (hence 100% accuracy). In total, 21 of the 23 benchmarks were successfully *adapted* to (i.e., these 21 analyzed apps normally ran on) the new platform, for an adaptation success rate of 91.3%.

The last column of the table lists the total analysis time of ARPDroid for each benchmark. As shown, the cost ranged from around one minute to over five minutes, for an average of 133 seconds. There does not seem to be a consistent correlation between the app sizes and time costs, as expected (since the size is not a necessary indicator of app complexity). For the successfully repaired apps, the code transformation led to a 8.29% increase in app size on average. Given that these numbers were obtained with our prototype implementation without any performance optimization/tuning, our approach is expected to be well scalable to real-world Android apps.

In all, our preliminary results reveal good potential of our technique for practical use in terms of both effectiveness and efficiency. On the other hand, however, considering the limited scale of our evaluation, we could not claim that the results will surely be generalized. More extensively evaluating our tool with a much larger and diverse set of benchmarks is a major step of future work.

4 RELATED WORK

In [18], various Android app compatibility issues are studied, with a focus on those issues due to the Android fragmentation problem yet without the consideration of permission-induced incompatibilities. A user study has been conducted to understand how Android users react and adapt to the runtime permission model, and revealed that users prefer the new model over the static permission mechanism [2]. In an extended study [1], researchers confirmed similar preferences of end users for the new permission model driven by their security and privacy concerns. Unlike our approach, these studies do not address the need of mobile developers for more productively using the runtime permission mechanism during app development and maintenance.

Libraries and tools are available to facilitate mobile developers in the transition to the runtime permission model of Android, including annotation-based APIs [15] and wrappers [16]. These utilities are helpful for developers of new apps by making it easier to write permission checking/requesting code. In comparison, our technique addresses both the need for migrating legacy apps to new platforms and the need for ensuring compatible permission uses in new apps, by detecting and repairing incompatibility issues induced by runtime permissions in both legacy and new apps.

5 CONCLUSION

We presented the technical design and implementation of ARPDroid, an automated solution that assists developers with correct adoption of the runtime permission model of Android. Given an app, originally targeting new or older platforms, our analysis detects and repairs incompatible permission uses, adapting the app to the new permission model. Our preliminary evaluation of ARPDroid suggests that our solution is highly accurate in detection with promising repair capability and practical scalability. The tool has been made publicly accessible online. Both technical expansion and empirical extension are immediate next steps. In particular, automating dynamic validation of repair apps and handling permission request responses more thoroughly are part of future work.

REFERENCES

[1] Panagiotis Andriotis, Shancang Li, Theodoros Spyridopoulos, and Gianluca Stringhini. 2017. A comparative study of android usersąŕ privacy preferences under the runtime permission model. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*. Springer, 604–622.

[2] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. 2016. Permissions snapshots: Assessing users' adaptation to the Android runtime permission model. In *IEEE International Workshop on Information Forensics and Security (WIFS)*. 1–6.

[3] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Instrumenting Android and Java applications as easy as abc. In *International Conference on Runtime Verification*. Springer, 364–381.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 259–269.

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 217–228.

[6] Malinda Dilhara. 2018. ARPDroid. (2018). Retrieved January 10, 2018 from https://bitbucket.org/malindadoo/arpdroid

[7] Google. 2014. ActivityCompat. (2014). Retrieved January 10, 2018 from https://developer.android.com/reference/android/support/v4/app/ActivityCompat.html

[8] Google. 2014. Android emulator. (2014). Retrieved January 10, 2018 from https://developer.android.com/studio/run/emulator.html

[9] Google. 2014. App Manifest. (2014). Retrieved January 10, 2018 from https://developer.android.com/guide/topics/manifest/manifest-intro.html

[10] Google. 2014. ContextCompat. (2014). Retrieved January 10, 2018 from https://developer.android.com/reference/android/support/v4/content/ContextCompat.html

[11] Google. 2014. Requesting Permissions. (2014). Retrieved January 10, 2018 from https://developer.android.com/guide/topics/permissions/requesting.html

[12] Google. 2015. Normal Permissions for Android Applications. (2015). Retrieved January 10, 2018 from https://developer.android.com/guide/topics/security/normal-permissions.html

[13] Google. 2015. Requesting Permissions at Run Time. (2015). Retrieved January 10, 2018 from https://developer.android.com/training/permissions/requesting.html

[14] Karumi/Dexter. 2018. Android library that simplifies the process of requesting permissions at runtime. (2018). Retrieved January 15, 2018 from https://github.com/Karumi/Dexter

[15] S. Katafuchi. 2016. Hotchemi's Permission Dispatcher. (2016). Retrieved January 10, 2018 from https://github.com/permissions-dispatcher/PermissionsDispatcher

[16] Daniel Lew. 2015. Android permissions library proliferation. (2015). Retrieved January 10, 2018 from https://gist.github.com/dlew/2a21b06ee8715e0f7338

[17] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 76.

[18] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.