How do Mobile Apps Violate the Behavioral Policy of Advertisement Libraries?

Feng Dong[1], Haoyu Wang[1,2], Li Li[3], Yao Guo[4,5], Guoai Xu[1], Shaodong Zhang[1]
[1] Beijing University of Posts and Telecommunications, Beijing, China
[2] Beijing Key Laboratory of Intelligent Telecommunication Software and Multimedia
[3] Faculty of Information Technology, Monash University
[4] Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
[5] School of Electronics Engineering and Computer Science, Peking University, Beijing, China
{dongfeng,haoyuwang,xga,zhangsd}@bupt.edu.cn,li.li@monash.edu,yaoguo@pku.edu.cn

ABSTRACT

Advertisement libraries are used in almost two-thirds of apps in Google Play. To increase economic revenue, some app developers tend to entice mobile users to unexpectedly click ad views during their interaction with the app, resulting in kinds of ad fraud. Despite some popular ad providers have published behavioral policies to prevent inappropriate behaviors/practices, no previous work has studied whether mobile apps comply with those policies. In this paper, we take Google Admob as the starting point to study policy-violation apps. We first analyze the behavioral policies of Admob and create a taxonomy of policy violations. Then we propose an automated approach to detect policy-violation apps, which takes advantage of two key artifacts: an automated model-based Android GUI testing technique and a set of heuristic rules summarized from the behavior policies of Google Admob. We have applied our approach to 3,631 popular apps that have used the Admob library, and we could achieve a precision of 86% in detecting policy-violation apps. The results further show that roughly 2.5% of apps violate the policies, suggesting that behavioral policy violation is indeed a real issue in the Android advertising ecosystem.

KEYWORDS

Ad library; Admob; behavior policy; Ad fraud; Android

ACM Reference Format:
Feng Dong[1], Haoyu Wang[1,2], Li Li[3], Yao Guo[4,5], Guoai Xu[1], Shaodong Zhang[1] . 2018. How do Mobile Apps Violate the Behavioral Policy of Advertisement Libraries?. In *HotMobile '18: 19th International Workshop on Mobile Computing Systems & Applications, February 12–13, 2018, Tempe , AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3177102.3177113

HotMobile '18, February 12–13, 2018, Tempe , AZ, USA

ACM ISBN 978-1-4503-5630-5/18/02…$15.00
https://doi.org/10.1145/3177102.3177113

1 INTRODUCTION

Mobile apps have seen widespread adoption in recent years, with more than 3.3 million apps available in Google Play [7]. More than 85% of the apps are free [25], and ad libraries (e.g., Admob) are commonly used by app developers to monetize their apps by showing ads to mobile users.

A significant number of studies focused on the topic of ad libraries in various directions. LibRadar [17] and WuKong [23] were proposed to identify third-party libraries used in Android apps. Meng et al. [18] focused on privacy leakage detection of ad libraries. Shekhar et al. [20] proposed to separate the privilege of ad libraries from the host apps to prevent permission escalation. Cho et al.[8] and Nath et al.[10] proposed to detect click fraud of advertisements in Android apps.

Although mobile ad libraries have been extensively studied, no previous work has been proposed to analyze ***whether Android apps comply with the usage and behavioral guidelines defined by ad networks***. Some popular ad libraries (e.g., Admob and DoubleClick) have released policies to regulate the behaviors of apps that use them [4, 13], including how the ads should be placed and how they interact with users. Some policies are mandatory and are meant to avoid kinds of ad fraud, which might affect user experiences significantly. Moreover, breaking the policies can also impact the reputation of ad networks and advertisers, as in the end, the displayed ads are from those ad companies.

In this paper, *we propose an exploratory study on how Android apps violate the behavioral policy of ad library.* To the best of our knowledge, this work is the first attempt in our community towards checking whether ad libraries are properly used by Android apps. We choose the Google Admob library [5] as the starting point because it is a quite popular ad network [21] provided by Google, the official maintainer of Android, and it has defined clear guidelines and policies for its users to follow, which eases implementing and checking for instances in which ads violate those guidelines and policies.

Practically, we summarize a taxonomy of policy violations based on the behavior policy provided by Google Admob [4]. Based on this taxonomy, we then propose an automated approach to detect policy-violated apps. We introduce an automated test input generation technique to traverse the user interface (UI) states (i.e., a running page of apps) and construct a UI state transition graph. Each state has been associated with a set of visual views (e.g., buttons), where

the metadata such as the position of each view is included. The ad-related transitions are then located based on the UI state graph and are consequently checked against a set of heuristic rules characterized based on the Admob policies.

We have implemented a prototype system to detect policy violations and applied it to 3,631 popular Android apps. The experimental results show that we could achieve a precision of 86% in violation detection, which counts for roughly 2.5% of apps violated Admob's behavior policies, demonstrating that policy violation is indeed a real issue in the Android ecosystem. We have released the dataset and experiment results to the mobile app research community at: *https://github.com/BUPT-privacy-research/ad-policy-violation*.

2 BACKGROUND & RELATED WORK

2.1 The Behavior Policy of Admob

Mobile ads are usually displayed in three common ways: 1) **Banner ad**, which is a rectangular image or text ad that occupies a spot within apps; 2) **Interstitial ad**, which is a square and locates in the center of the screen; 3) **Full Screen ad**, which fills the whole screen. These types of ads are integrated into an app by either specifying it in the layout XML file or embedding it in the source code[1].

Google Admob has released a series of policies (including content policies, behavioral policies, etc.) to guide and regulate the usage of Admob library [4]. Once an app developer fails to comply with these policies, Admob will disable ad serving or disable his/her Admob account, although Admob does not explicitly provide how they perform policy-violation detection. In this paper, *we only focus on behavioral policies*, because other policies are either non-mandatory or hard to detect due to the vague standard. For example, Admob does not permit monetization of dangerous content, while it is hard to define which content is dangerous. As a result, we consider in this work six Admob behavioral policies that are summarized, along with their key behaviors, in Table 1.

2.2 Related Work

Here, we discuss some closely related work relating to general ad library and automated Android UI testing.

Ad libraries. A significant number of studies focused on the topic of ad libraries in various directions such as on discovering ad libraries [14, 17, 22], on detecting privacy leaks within ad libraries [11, 16], on separating the privilege of ad libraries from host apps [20, 24], and on pinpointing click frauds [8, 10]. We believe all the aforementioned approaches can be leveraged to supplement our work towards providing a better characterization of violated ad policies.

Automated Android GUI Testing. Automated app testing has been recurrently adopted to address various challenges [6, 9]. The testing part of our approach is in line with those work but have a different focus. Because automated Android GUI testing is known to be time-consuming, we have conducted several customizations on our model-based approach to improve the overall efficiency in Section 3.2.

[1] https://developers.google.com/admob/android/quick-start

3 POLICY-VIOLATION DETECTION

3.1 Overview

The overall process of our approach is shown in Figure 1. We first use LibRadar [3, 17], an obfuscation-resilient tool to identify apps that use Admob library with simple static analysis and feature comparison. Then we propose an automated test input generation technique to run apps that embed the Admob library on smartphones. We preserve the attribute information of visual views (i.e., controls) and state (i.e., a running page of apps) transition information in the *UI state transition graph*. By leveraging properties such as resource strings, view types preserved in the attribute information, we can identify Admob ad views accurately. Finally, we apply a set of heuristic rules to detect policy violations.

3.2 UI State Transition Graph Generation

Automated Test Input Generation. Monkey [2] is the most popular and lightweight tool to perform Android GUI testing, but the inputs generated by Monkey are *completely random*, which is not effective for us to explore the ad-contained states and generate the UI state transition graph.

In our approach, we generate inputs based on the current UI state to simulate real user behaviors. As Android apps are event-driven, inputs are mostly in the form of events. In our implementation, we simulate both UI events (e.g., touch, click, etc.) and system events (e.g., BOOT_COMPLETED intent). Note that **we generate *UI-guided events* according to the position and type of UI elements instead of sending random events and clicks like Monkey [2] does**. We take advantage of Accessibility [12] to understand the layout of the UI state, and obtain exhaustive information from each view such as name, size and class name, which could be used to build a view tree that can accurately describe current state. Our automation technique gets the view list from the current state and chooses the event input for the next view based on a systematic exploration strategy.

Exploration Strategy. Previous work suggested that traversing all the UI states of an app takes several hours [15]. By manually labeling 1,963 UI states that generated from 180 apps [1], **we find that 89.3% of the UI states (1,752) do not contain any ad views and more than 90% of ad views are displayed in either the main UI state or exit state**. Thus we take a breadth first traversal exploration strategy to explore the states for more effective results in terms of ad coverage and efficiency. To achieve the balance between time efficiency and coverage, we explore each state with maximal 50 events. The initial experiments on 180 apps suggested that the time efficiency increases by 17 times on average (7 minutes vs 120 minutes per app), while we could cover 90% of the UI states that contain ad views.

Generating UI State Transition Graph. Our automation technique runs apps automatically to generate the UI state

Table 1: Analysis of Google Admob Behavioral Policies.

Policy #	Policy Detail	Key Behaviors
Policy #1	Ads should not be placed very close to or underneath buttons or any other object which users may accidentally click while interacting with your application	Ads are overlapped with or hidden behind other views
Policy #2	Ads should not be placed in a location that covers up or hides any area that users have interest in viewing during typical interaction	Displaying ads during users' interaction with the app
Policy #3	There must be a way to exit a screen without clicking the ad	Ads cannot be closed unless clicked
Policy #4	Ads should not be placed in applications that are running in the background of the device or outside of the app environment	Displaying ads outside the host apps
Policy #5	Ads should not be placed in a way that prevents viewing the app's core content. Example: an interstitial ad triggered every time a user clicks within the app	Poping up ads frequently
Policy #6	Publishers are not permitted to place ads on any non-content-based pages such as thank you, error, log in, or exit screens	Placing ads on start, exit, login, or thank you screens

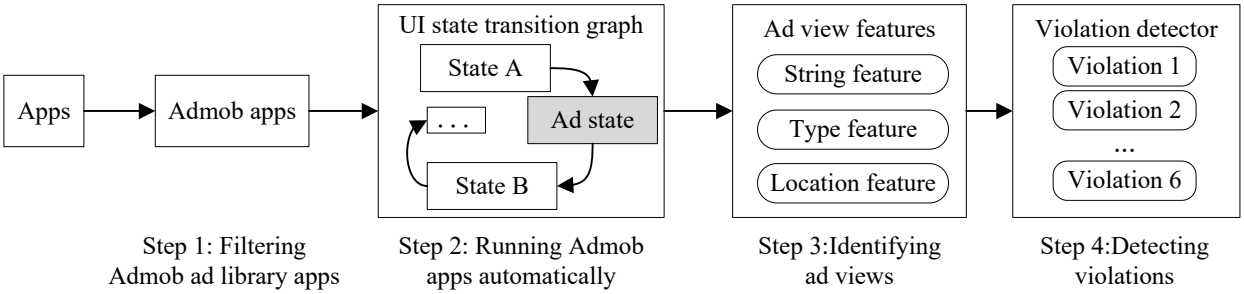


Figure 1: Approach Overview.

transition graph, which is basically a directed graph. Each node of the graph represents a state and each edge between two nodes represents the input event that triggers the state transition. Figure 2 shows an example of the UI state transition graph. The app (*com.rcplatform.fontphoto*, version 4.0.7) is launched by event 1 with an intent event "am start", triggering transitions from state A to state B that contains an ad view (with resource_id "ad_container"). State B is a launch screen (i.e., the first screen after starting the app), thus the ad view is placed on the non-content-based page, which violates the Policy #6.

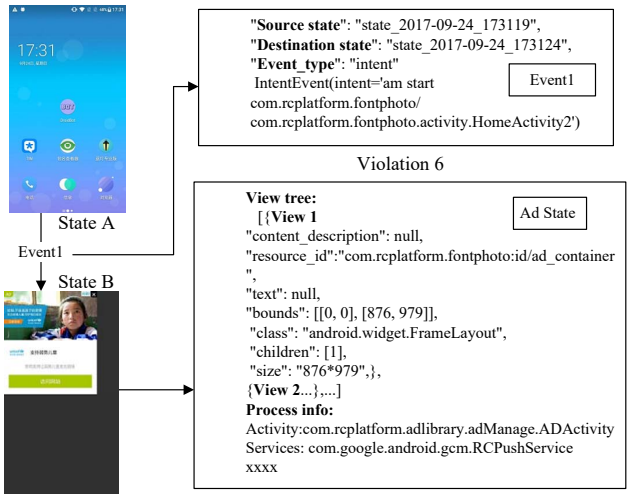


Figure 2: An example of UI state transition graph.

3.3 Identifying Ad Views

To differentiate the Admob ad views from a large number of normal views in a given UI state, we manually labeled many ad views and normal views, and compared them from various aspects (e.g., resource string, position, view type, etc.) to explore features that can distinguish them.

We randomly choose 180 apps from the 3,631 Admob apps and manually label 1,963 UI states generated from them. Overall, we obtain 1,752 ad-free states and 211 ad-contained states. Then we observe various features that could be used to identify ad views. As shown in Table 2, the features could be classified into three categories: string features, type features and location features.

For string features, we find that roughly 90% of Admob ad views (189 out of 211) in the labeled ad-contained states have special representative strings in "resource_id" field

Table 2: Features we used to identify Ad views.

Category (attribute)	Value
type (class)	ImageView, WebView
location (size)	620*800[Center],600*760[Center], 600*790[Center]
string (resource_id)	AdWebview, AdLayout, ad_container, fullscreenAdView, FullscreenAd, AdActivity, AppWallActivity, AppBrainActivity, OverlayActivity

of attribute information, such as *"AdWebview"*, *"AdLayout"* and *"ad_container"*, while other views in the 1,963 labeled UI states do not use these strings. Thus we have collected a list of 9 common strings to detect ad views. For example, view 1 in the ad state in Figure 2 has a "resource_id" named "id/ad_container", which indicates that it is an ad view.

For apps that have no explicit string features due to fully obfuscation (22 out of 211), we use a heuristic detection method based on the type and the position of the view. All the 211 Admob ad views in our labeled dataset are implemented by system views "ImageView" and "Webview". Furthermore, they usually have specific size and position features as shown in Table 2. For example, for "Webview" that locates in the center of the screen with the size of 620*800, we will identify it as an ad view.

3.4 Violation Detection

We now provide the practical rules that are, so far, implemented in our work to detect violations.

Violation 1: We consider that an ad view violates Policy #1 if and only if it has overlapped with or hidden behind other views. Note that we detect overlap and hiding by calculating the size, bounds and z-coordinates in the attribute information of the views.

Violation 2: Violation 2 can be found in either interstitial ads or full screen ads. Our violation detector first traverses all the UI states by checking the UI state transition graph to identify the states that contain interactive views (e.g., dialog and buttons). Then it will check the adjacent (previous or next) UI states to analyze whether an ad view exists. The UI state will be regarded as a violation if an ad view is placed on top of interactive views.

Violation 3: In general, the interstitial ads and full screen ads could be closed by either clicking the *close widget* or touching the back button. Since it is non-trivial to automatically identify the *close widget*, in this paper, we only consider the situation where interstitial or full screen ads cannot be closed by touching the back button.

Violation 4: Violation 4 is flagged when an ad view is displayed after its host app exits. Thus, for each state that contains ad views, we check the package name of the host app[2]. Note that the Android system default HOME app's name varies according to different system version, the package name is "com.cyanogenmod.trebuchet" in our experiment smartphones.

Violation 5: This violation is flagged if and only if an ad state is triggered more than three times. To avoid repeated visits, which may cause inaccurate counts, we only count it once if a UI state is visited, although multiple times, via the same path.

Violation 6: This violation is flagged if and only if interstitial or full-screen ads are placed on the app launching, exiting and log in pages. The app launching and exiting states could be identified when we start or close the app. The log in state can be heuristically identified based on the view information. For example, if a state contains two *editviews* and has string features such as "username" and "password", we will regard it as a log in state.

[2]Note that the host app of an ad view will be *system default HOME app* if the tested app exits.

Table 3: The experiment result of detection.

	Violation (prediction)	Normal (prediction)
Violation	TP(74)	FN(15)
Normal	FP(12)	TN(3530)

4 EVALUATION

We study the prevalence of policy violation on a large scale of apps that embed the Admob library. Apps are running on a Nexus 5 smartphone instead of emulators, because previous work [19] suggested that some apps refuse to display ads when it detects the app is running in an emulator. Note that our automation GUI testing tool is running on a laptop, and it fetches app information from the device and sends input events to the device through Android Debug Bridge (ADB). Experiment results are available on GitHub [1].

4.1 Data Collection

We collected 5,981 popular Android apps from Google Play in July 2017. Take advantage of LibRadar [3], we could identify 3,631 apps (60.71% of the collected apps) that embed the Admob library. We get 39,702 states from these apps, among which 3,872 states contain ad views. Through a manual analysis, we found that the accuracy of ad view identifier reached 95%, which means that most of the Admob ad views in our dataset are identified correctly.

4.2 Experiment Results

Overall Result. First of all, to label the ground truth data, two experienced master students in our group check the graphs manually for five days. As a result, we label 89 policy-violation apps, which covers about 2.5% of the Admob apps in our dataset. We then use these apps to evaluate our automated violation detector, the detection results are shown in Table 3. Our violation detector identifies 86 policy-violation apps. Among them, 12 apps are false positives. Besides, 15 apps are false negatives. Thus the corresponding precision and recall of our detector are 86.05% and 83.15%, respectively.

Table 4: The distribution of violation apps.

Violation	#1	#2	#3	#4	#5	#6	Total
Number of apps	5	6	0	2	18	87	89

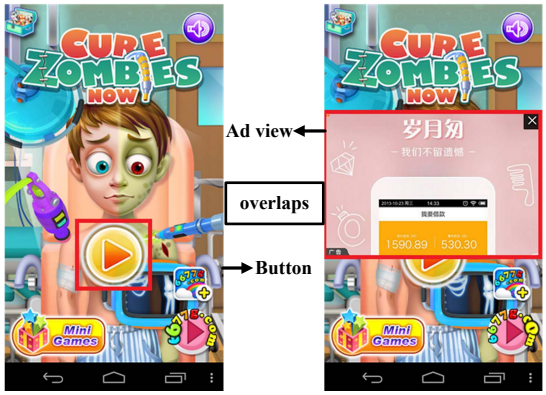


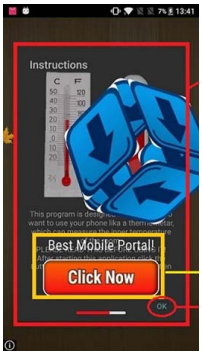
Figure 3: An example of Violation #1.

False Negatives. We inspected the 15 instances and found three reasons that could lead to false negatives. (1) **Lazy-loading of the ad views**. In some cases, it has a delay time to display the ad views, leading to the result that we cannot record it in the states. For example, the ad views in app *br.com.gtlsistemas.crosswords* (version 1.6.0) has a 5 seconds delay to be displayed before our automation technique records the state. (2) **Fully obfuscation**. We find some apps are fully obfuscated, making it hard to extract representative string features to identify ad views. For example, we extract no useful features to identify ad views in app *bubbleshooter.blaze.pop* (version 1.0.1). (3) **Some ad views are not displayed when the app is used for the first time**. For example, ad views in app *com.tp.android.waxspa* (version 1.0.1) appear only after the app is started at the second time.

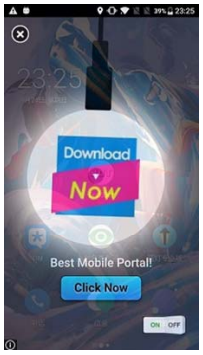
False Positives. Our violation detector caused false positives because some normal views had the same size and location features as ad views, which could mislead our detector. We will discuss the limitations in the Section 5.1.

Violation Distribution. The distribution of the 89 policy-violation apps is shown in Table 4. Note that some apps have violated more than one policy. For example, app *com.spiderapps.redhotfruits* (version 1.2) violates type 4, type 5 and type 6. More than 97% of the policy-violation apps break Policy #6, that is to say, these apps pop up ads upon launching or before potentially leaving the app. One possible reason is that placing ads on these pages could potentially increase the impression (i.e., the number of ads displayed) and clicks. Note that we do not identify apps that violate Policy #3. One possible reason is that Violation #3 could greatly decrease user experience, which will lead to the uninstallation of the app. Thus this type of violation is uncommon.

Case Study. Fig. 3 shows an example of Violation #1. The app (*com.tp.android.curezombie2nd*, version 1.0.0) has more



(1) Violation 2: Ad view pops up above the instruction dialog



(2) Violation 4: Ad view is placed in the home screen

Figure 4: Examples of Violation #2 and Violation #4.

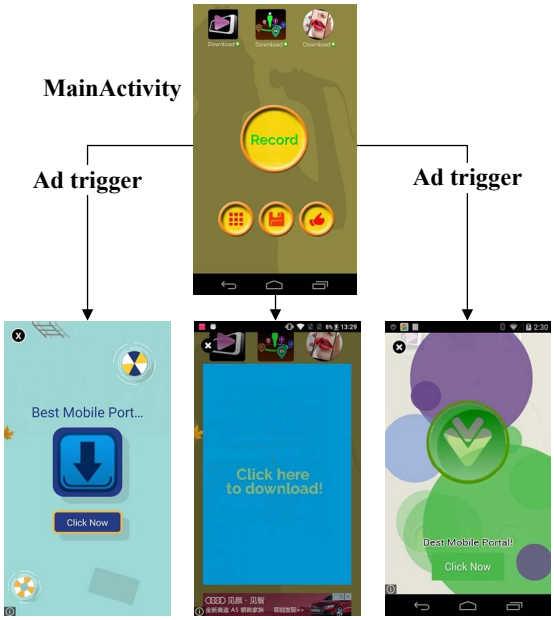


Figure 5: An example of Violation #5.

than 1 million downloads in Google Play. The *game button* is overlapped with the ad view, which could cause accidentally click. Fig. 4 (1) shows an example of Violation #2 (*com.heaven.thermo*, version 1.3). The ad view pops up above the instruction dialog, causing undesirable click on the ad instead of the *OK button*. Fig. 4 (2) shows an example of Violation #4 (*com.HomeCleaningGames*, version 5.0). The ad view is placed on the home screen after exiting the host app. The example of Violation #5 (*com.sink.apps.girl.voice.changer*, version 1.0.4) is shown in Fig. 5. The ad view triggers every time after the main activity receives an event input. The app has 1,000,000 - 5,000,000 downloads. The example of Violation #6 is shown in Fig. 2. The policy-violation apps with high downloads may have already produced a huge negative impact on a lot of mobile users.

5 DISCUSSION

5.1 Method Limitations

Ad Coverage. To improve the time efficiency, we use an optimized UI-guided automation technique with the BFS exploration strategy, which is able to traverse more than 90% of the ad-contained states. However, it is quite possible that we cannot traverse all the states that contain ad views, e.g., it has a delay time to display the ad views in some cases as discussed in Section 4.2. Note that many apps abuse the notification bar to display ads, we did not study this kind of ads in this paper.

Ad View Identification. We use a heuristic approach to detect ad views. However, our experiment results show that some ad views have no obvious features due to code obfuscation, which could cause false positives. To mitigate this, one possible solution is to apply advanced program analysis (e.g., mapping ad views to the decompiled code) or machine learning techniques to build a more accurate ad view identifier.

5.2 Implications

Our experiment results show that behavioral policy violation is indeed a real issue in the Android ecosystem. Profit-driven app developers may break the behavioral policies of ad networks in order to increase economic revenue. However, *inappropriate usage of ad libraries may impact the success of mobile apps as it gives a bad impression to app users*. Furthermore, app markets and ad networks should take efforts to identify/prevent policy violation apps. Ad networks should also consider how to prevent/check certain policy violations when designing ad libraries, e.g., requiring app developers to implement a *close function* for each ad view in the ad libraries to prevent Violation #3.

6 CONCLUSIONS

In this paper, we propose an exploratory study of ad behavioral policy violation in Android apps. We first design a set of violation rules based on the characteristics observed from the Admob behavioral policies. Then, we propose an automated approach (via automated GUI testing) to detect ad behavior policy violations. By applying our approach to 3,632 apps that have embedded with Admob library, our approach achieves a precision of 86.05% in detecting policy violations. The experiment results further show that 2.5% of apps have violated the Admob policies, suggesting that behavior policy violation is a real problem in Android ecosystem and hence is necessary to be highlighted and subsequently avoided.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No.61702045, No.61401038 and No.61772042), the 2016 Frontier and Key Technology Innovation Project of Guangdong Province Science and Technology Department (No.2016B010110002), the National Key Research and Development Program (No.2017YFB0801901), and the BUPT Youth Research and Innovation Program (No.2017RC40). *Haoyu Wang* is the corresponding author. We would like to thank our shepherd Dr. *Narseo Vallina-Rodriguez* and the anonymous reviewers for their helpful comments.

REFERENCES

[1] 2017. admob-behavioral-policy-violation. (2017). Retrieved October 16, 2017 from https://github.com/admob-behavioral-policy-violation/violation-detector.git
[2] 2017. MonkeyRunner. (2017). https://developer.android.com/studio/test/monkeyrunner/index.html
[3] 2018. LibRadar. (2018). Retrieved January 9, 2018 from https://github.com/pkumza/LibRadar
[4] Google Admob. 2017. AdMob & AdSense policies. (2017). Retrieved October 14, 2017 from https://support.google.com/admob/answer/6128543?hl=en&ref_topic=2745287
[5] Google Admob. 2017. AdMob by Google. (2017). Retrieved October 21, 2017 from http://www.google.cn/admob/
[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59.
[7] AppBrain. 2017. Number of Android applications. (2017). https://www.appbrain.com/stats/number-of-android-apps
[8] Geumhwan Cho, Junsung Cho, Youngbae Song, and Hyoungshick Kim. 2015. An empirical study of click fraud in mobile advertising networks. In *ARES*. IEEE, 382–388.
[9] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *ASE*. 429–440.
[10] Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. Madfraud: Investigating ad fraud in android applications. In *MobiSys*. ACM, 123–134.
[11] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android.. In *NDSS*.
[12] Android Developers. 2018. Accessibility Overview. (2018). Retrieved January 9, 2018 from https://developer.android.com/guide/topics/ui/accessibility/index.html
[13] DoubleClick. 2017. DoubleClick program policies. (2017). Retrieved October 21, 2017 from https://support.google.com/adxseller/topic/7316904?hl=en&ref_topic=6321576
[14] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *SANER 2016*. 403–414.
[15] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps.. In *NSDI*. 57–70.
[16] Minxing Liu, Haoyu Wang, Yao Guo, and Jason Hong. 2016. Identifying and Analyzing the Privacy of Apps for Kids. In *HotMobile '16*. 105–110.
[17] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. Libradar: Fast and accurate detection of third-party libraries in android apps. In *ICSE*. ACM, 653–656.
[18] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. 2016. The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads.. In *NDSS*.
[19] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *EuroSec '14*.
[20] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications.. In *USENIX Security Symposium*, Vol. 2012.
[21] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *IMC*. 221–233.
[22] Haoyu Wang and Yao Guo. 2017. Understanding Third-party Libraries in Mobile App Analysis. In *ICSE 2017*. 515–516.
[23] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: a scalable and accurate two-phase approach to Android app clone detection. In *ISSTA*. 71–82.
[24] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I. Hong. 2017. Understanding the Purpose of Permission Use in Mobile Apps. *ACM Trans. Inf. Syst.* 35, 4, Article 43 (July 2017), 43:1–43:40 pages.
[25] Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. 2017. An Explorative Study of the Mobile App Ecosystem from App Developers' Perspective. In *WWW 2017*. 163–172.