MAdLens: Investigating into Android In-App Ad Practice at API Granularity

Ling Jin, *Student Member, IEEE,* Boyuan He, *Student Member, IEEE,* Guangyao Weng, *Student Member, IEEE,* Haitao Xu, *Member, IEEE,* Yan Chen, *Fellow, IEEE,* and Guanyu Guo, *Student Member, IEEE*

Abstract—In-app advertising has served as the major revenue source for millions of app developers in the mobile Internet ecosystem. Ad networks play an important role in app monetization by providing third-party libraries for developers to choose and embed into their apps. Various ad mediations help developers manage all of the ad libraries used in apps to show the best available ad among received ads from different ad network servers. However, developers lack guidelines on how to choose from hundreds of ad networks or ad mediations and various ad features to maximize their revenues without hurting the user experience of their apps. Our work aims to provide app developers guidelines on the selection of ad networks, ad mediations and ad placement by observing current common practices.

To this end, we investigate 838 unique APIs from 207 ad networks which are extracted from 277,616 Android apps, develop a methodology of ad type classification based on UI interaction and behavior, and perform a large scale measurement study of in-app ads with static analysis techniques at the API granularity. We found that developers have more choices about ad networks than several years before. Most developers are conservative about ad placement and about 77% of the apps contain at most one ad library. Besides, the likeliness of an app containing ads depends on the app category to which it belongs. Furthermore, we propose a terminology and classify mobile ads into five ad types: Embedded, Popup, Notification, Offerwall, and Floating. Also, our research shows that it is a better solution for developers to integrate ad libraries with ad mediation feature in their apps because it may avoid bad ratings and improve user experience. And in our findings, more than 95% of Embedded, Popup, Notification and Offer ads locate in the zero activity (main activity), the first activity and the second activity of Android apps. More interestingly, developers tend to put high aggressive ads on activities which need deeper user interaction. Our research is the first to reveal the preference of both developers and users for ad networks, ad mediation feature and ad types.

Index Terms—Android app, In-app advertising, Ad network, Ad mediation, Static analysis.

1 Introduction

Android has increasingly become the dominant operating system for mobile devices today and its world wide market share has hit 86.1% in the first quarter of 2017 [1]. According to [2], the total number of apps in Google Play has reached 2.9 million in the Feb 2017, 92.5% of which are free apps. In-app advertising has become one of the major sources of income for free app developers, VisionMobile predicts that the in-app advertising market will be worth 62 billion US dollars by 2017 [3].

Most of free apps leverage third-party in-app advertising for monetization. To achieve this, app developers need to connect their own apps to an ad network, a intermediate platform used for ad delivery. This process usually requires code integration: merging a pre-compiled library (a.k.a. ad library) provided by a specific ad network with the original app. In practice, there are hundreds of ad networks available on the market, and even for a specific ad network, developers still have great flexibility in this integration process. As described in documentation, many ad networks provide different ad types, such as banner, interstitial, native, video and etc, varying in aggressiveness to users. Furthermore, there is no unified classification system available for in-app ads, making it easier for developers to get confused. Consequently, for a specific category of app, following decisions must be made by developers before integration:

- What is the trend in ad networks used by apps in recent years?
- Can in-app advertising jeopardize user growth and app ratings?
- Are there any patterns in different co-existed ad types within the same app?
- How many ad networks within an app can be considered as acceptable to users?
- What are the popular ad types and what are the ad types that users are inclined to tolerate?
- Why is it a better solution to choose ad networks with ad mediation feature and how ad mediation affects user experience?

- *Corresponding Author: Yan Chen*
- *Ling Jin, Boyuan He, Guangyao Weng and Guanyu Guo are with the Institute of Cyber Security Research, Zhejiang University, Hangzhou, China, 310027.*
  E-mail: {ljin1995, heboyuan, gyweng, guanyuguo}@zju.edu.cn
- *Haitao Xu is with School of Mathematical and Natural Sciences, Arizona State University, Glendale, AZ, USA, 85306.*
  E-mail: hxu@asu.edu
- *Yan Chen is with the Department of Electrical Engineering and Computer Science, Robert R. McCormick School of Engineering and Applied Science, Northwestern University, Evanston, IL, USA, 60208.*
  E-mail: ychen@northwestern.edu

- Where do different ad types locate in the view hierarchy of Android apps and how location influences user engagement from the perspective of user interaction?

All the questions above must be carefully considered by developers because excessive ads impression or improper ad type may ruin an app's user experience, thus causing user loss. Given the fact that there are hundreds of ad networks with different features on the market, it poses a great challenge for developers to choose the best ad networks and ad types for their own apps with the balance between revenue and apps' user experience. Although mobile advertising ecosystem has been a target of many recent research, most of them focus on security and privacy of ad networks [4] [5] [6] [7] [8], mobile ad fraud [9] [10], and ad targeting [11] [12] [13]. None of existing works can address this challenge.

To provide guidelines of choosing best ad networks and ad types, in this paper we study Android in-app advertising from a developer's perspective. According to the principle of statistics, it is of great significance to mastering the general trend for guiding the holistic direction of development. Popular apps are favorable platforms for mobile advertising, because they have a large number of users. The more users an app has, the more likely the mobile ads embedded are seen by its users, and thus improving the developers' revenue. Therefore, we statistically surveyed how the developers of those popular apps leverage mobile ads to gain their profits. People naturally seek profit and circumvent loss. In the environment of market economy, the developers of popular apps will take advantage of their vast number of users to gain revenue by embedding ads. At the same time, they will certainly avoid ads affecting the user experience, which may make their users unsatisfactory and cause a user loss. Therefore, we think that by observing how the developers of popular apps use in-app advertising, we can give guidance to those developers who want to make money from mobile ads. In particular, we develop a system called `MAdLens`, a static analysis framework for Android apps, which extracts libraries of different ad networks from a large set of apps, map each of ad relevant APIs inside a SDK to a specific ad type and generates summary information (e.g. number of ad API calls, number of instructions, number of Android component and etc). Leveraging on `MAdLens`, we further perform a large scale measurement across the Android market and uncover the current trend in usage of mobile ad networks, aggressiveness difference of ad types and its impact on various properties of apps.

To summarize, this paper makes the following contributions:

- To the best of our knowledge, we are the first to provide practical guidelines for mobile developers to monetize their apps through third-party in-app advertising.
- We are the first to map APIs of ad network to specific ad types and measure third-party in-app advertising in Android apps at API granularity.
- We are the first to provide a unified classification system for mobile in-app ads, and measure the impact of different ad types on various properties of apps.
- In order to explore how ad mediation, a new feature of ad networks, influences our measurement of the impact of different ad types on various properties of apps, we improve our mapping methodology by using function call graph instead of simply scanning the smali code. To the best of our knowledge, we are the first to detect ad networks with ad mediation feature and measure their impact on mobile advertising in Android apps.
- We are the first to measure the impact of ad types on mobile advertising from the perspective of user interaction. More specifically, we target at the relationship between ad types and the view hierarchy of Android apps, which implies one of the most important evaluation indexes for Android apps, the user engagement.

The article extends our conference version [14] in the following important aspects. We detect ad networks with ad mediation feature based on function call graph and greatly improve the accuracy of the percentages of ad types in Android apps. This demonstrates how ad mediation, a new popular feature of ad networks, influences our measurement of the impact of ad types on a variety of properties of apps. We also provide additional analysis on the relationship between ad types and the view hierarchy of Android apps from the perspective of user interaction which implies the user engagement, one of the most significant evaluation indexes for Android apps.

Overall, we successfully extract 838 unique APIs from 207 ad networks, which are identified in a dataset of 277,616 apps. Our results reveal many implications for developers and here we list some major ones:

1) The number of apps using in-app ads has increased in recent years but 77% of the apps contain at most one ad network, indicating that a conservative strategy is widely accepted.

2) Developers are inclined to use two lowly aggressive ad types: *Embbedded* and *Popup* simultaneously.

3) Too many ad networks that are placed in the same app will dissatisfy the users and therefore lead to bad rating. Empirically, it's better to integrate no more than 6 ad networks into a single app.

4) A better solution for developers is to use ad libraries with ad mediation feature when managing multiple SDKs. Ad mediation decreases the number of ad libraries in apps which may avoid bad ratings and improve user experience.

5) Developers may put more aggressive ads on deeper layouts for that users who explore more activities could have higher user engagement and may be more tolerant to aggressive ads.

The remainder of this paper proceeds as follows: Section 2 provides relevant background in Android third-party ads and Section 3 provides the system design of `MAdLens` as well as detailed discussion of methodology we apply in `MAdLens`. Section 4 demonstrates the result of our large scale measurement study and gives explanations as well as implications based on the result. We describe our improved methodology and give a comparison of results between "with ad mediation" and "without ad mediation" in Section 5. In Section 6, we show our analysis on the relationship

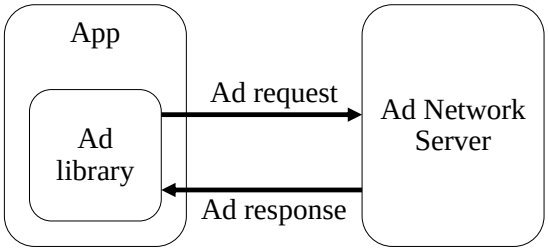


Fig. 1: Overview of mobile ad network.

between ad types and the view hierarchy of Android apps. Section 7 discusses the limitation of our works. Section 8 presents related work and Section 9 concludes the paper.

2 Background

In this section, we give a brief introduction to the ecosystem of mobile advertising and explain how third-party ad networks currently work. We also discuss details of mobile ad classification as well as ad aggressiveness.

2.1 Overview of Mobile Ad Network

Generally, the current ecosystem of mobile advertising involves three major participants: a publisher who shows ads in her app to make revenue, an advertiser who pays to get impression of her own ads and an ad network that serves as a intermediate platform between a publisher and an advertiser. In practice, an ad network connects the supply side platform (SSP) to the demand side platform (DSP), exchanging advertising impression inventory and revenue across the ecosystem.

As a publisher, most developers monetize their apps with third-party ad networks. This is usually implemented by importing an ad library provided by an ad network into an app, registering on the ad network's website to set up an ad account for payment, setting preferences in the ad library to specify credentials, suitable ad contents, desired ad frequency, layout, format and etc, and finally building and publishing the app. Unless explicitly stated, the two terms *ad network* and *ad library* are used interchangeably in this paper since all ad networks that we discussed have their correspondent ad libraries for integration. As shown in Figure 1, an ad request is triggered by ad library when a user browses specific parts of the app. The ad network server then receives the request, authenticates the developer's account, checks the parameters and responds with a desired ad. Any impression or click on this ad will be counted by ad network for revenue share.

2.2 Mobile Ad Network Integration in Android Apps

Ad networks often provide developers with both documentation and SDK for easy integration. The SDK usually consists of a pre-compiled ad library and its necessary dependencies. For Android apps, ad libraries are generally implemented in Java and provided in compiled jar files. Developers are required to import the ad library into the project of their own app and interact with the ad library with specific APIs.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  [...]
3  <com.google.android.gms.ads.AdView
4      xmlns:ads="http://schemas.android.com/apk/
           res-auto"
5      android:id="@+id/adView"
6      android:layout_width="wrap_content"
7      android:layout_height="wrap_content"
8      android:layout_centerHorizontal="true"
9      android:layout_alignParentBottom="true"
10     ads:adSize="BANNER"
11     ads:adUnitId="MY-UNIT-ID">
12 </com.google.android.gms.ads.AdView>
13 [...]
```

Listing 1: Banner ad implementation in XML

Most of ad networks provide a rich API surface, which allows the developer considerable latitude in manipulating the ad impression. However, this interaction is done either by changing layout files or calling specific Java API method. Listing 1 and Listing 2 give two examples of a banner ad implementation with Google Admob [15]. Both examples place an `AdView` to the app's GUI for the banner ad by changing app's XML layout files and using Java code respectively. Then a banner ad can be loaded where the developer wants by calling `loadAd()` method of the `AdView` class.

```
1  import com.google.android.gms.ads.AdView;
2  [...]
3  AdView adView = new AdView(this);
4  adView.setAdSize(AdSize.BANNER);
5  adView.setAdUnitId("MY-UNIT-ID");
6  [...]
7  }
```

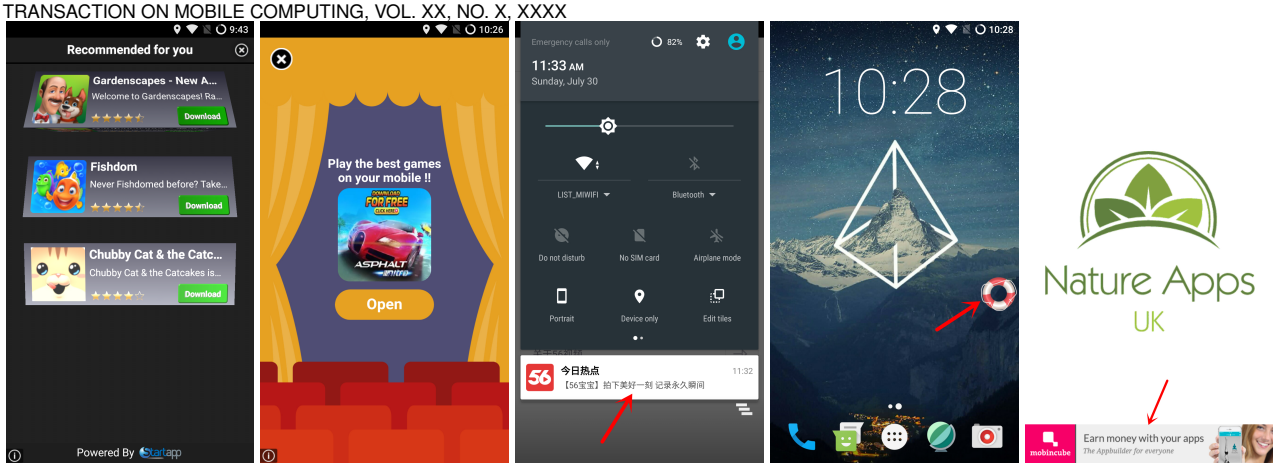
Listing 2: Banner ad implementation in Java

2.3 Mobile Ad Classification and Aggressiveness

An ad network usually supports more than one ad format and developers are expected to choose those that fit best with the design and user flow of their app. By randomly picking up 10 popular mobile ad networks (Table 1), we can easily draw conclusion that there is no unified classification system for mobile ads. For research purpose, specifically, in order to keep a consistent criterion for analyzed results, we need to carry out a unified classification of current advertising presentation forms in the market. We did not arbitrarily classify them. We referred to the classification methodologies of popular ad networks, such as Google AdMob [16], Airpush [17] and Chartboost [18]. In this paper, we propose a methodology of mobile ad classification based on the behavior, UI layout and UI interaction of an ad. Our classification methodology covers all known forms of mobile advertising, such as, *Embedded* (including banner, text, and static picture), *Popup* (including interstitial, and full-screen ads), *Offerwall* (including offer wall and app wall).

Generally, we classify mobile ad into 5 different types that are described below:

1) **Offerwall** is the type of ad that uses a page appearing within the app to offer users rewards (e.g. Unlocking new features) in exchange for completing some specified actions (e.g. downloading other apps). An *Offerwall* ad often completely interrupts



(a) Offerwall (b) Popup (c) Notification (d) Floating (e) Embedded

Fig. 2: Examples of different mobile ad types.

app's behaviour and cannot be ignored by users. Figure 2(a) is a typical *Offerwall* ad that we identified in an Android app.

2) **Popup** is the type of ad that uses a new pop-up windows in front of current app's GUI to display its contents. Some popup ads are full screen like interstitial ad, while others are not. While *Popup* ads interrupt app's behaviours, most of them can be closed by users by clicking on the close button. A typical *Popup* ad is shown in Figure 2(b).
3) **Notification** is the type of ad that uses the notification mechanism to display its contents (Shown in Figure 2(c)). It is displayed in the system's notification area rather than inside the app. Technically, an *Notification* ad are still active after the app is closed since it uses notification pushing mechanism. Like any other daily used notifications (e.g. SMS), by default *Notification* ads are displayed while the same ringtone is played. Besides, users often have no idea from which app a *Notification* ad is pushed, therefore have troubles to turn it off.
4) **Floating** is the type of ad that appears in front of the GUI of current app with a floating window (Shown in Figure 2(d)). It usually just occupies a small part of the screen, but can still be seen outside the app.
5) **Embedded** (Shown in Figure 2(e)) stands for a type of ad which embeds its contents into current window of the app. It covers a variety of different ad types listed in Table 1, such as banner, video and etc. In general, embedded ads are user friendly because they rarely interrupt app's user flow or only interrupt for a short time and usually can be ignored, skipped or closed.

By taking all the factors mentioned above into consideration, we intuitively give different levels of aggressiveness to each type of mobile ad based on its user experience (Table 2). For those types of ad that can be easily ignored, skipped or closed by user, we consider them as user tolerable and their aggressiveness as "Low". Otherwise, we give "High" as its aggressiveness level. The only exception is the notification ad. Although it can be closed or even disabled by user, it still cause interference to user reading other notifications

TABLE 1: Supported ad format of 10 popular mobile ad network.

Ad Network	Ad Type Supported
Admob	Native, Video, Interstitial.
AdColony	Rich Media, Video.
Airpush	App Icon, Messaging, Notifications, Offerwall.
Chartboost	Content Lock, Interstitial, OfferWall, Video.
Fyber	Banner, Interstitial, Native, Video.
Inmobi	Banner, Native, Video, Interstitial, Rich Media.
Leadbolt	Native, Video, Interstitial.
RevMob	Interstitial, Video, Pop-up, Rich Media.
Startapp	App Icon, Full Page Ads, InApp Ads, Interstitial, Video.
Tapjoy	Content Lock, Interstitial, Offerwall, Rewards.

because most of notification ads will eventually turn into a notification flood. Another factor in classification is user privacy which has always been a hot research topic. Since the launch of GDPR (General Data Protection Regulation) by the European Union in 2018, people have paid more attention to personal privacy. We registered with several popular ad networks and found that developers could configure on the admin page to collect user information when showing ads. Different ad types require different user information, such as IMEI, GPS and WiFi. We found that low-aggressiveness ads collect less information about users. In general, they only collect IMEI and resolution to adjust the placement of ads. High-aggressiveness ads may determine whether to show videos based on WiFi connection. Some ads also fetch a list of installed apps. For example, some *Offerwall* ads are integrated to recommend other apps, and the list of installed apps is needed to avoid a redundant recommendation. Other ad types, such as *Floating*, may even require additional permission to display over other layouts. Our ad classification methodology takes user privacy into consideration, as high-aggressiveness ads tend to acquire more private information and require more permissions. We will demonstrate the impact of different ad types on various categories of Android applications in Section 4.

3 METHODOLOGY

In this section, we present our methodology that we apply in `MAdLens`.

TABLE 2: Aggressiveness of ad type.

Ad Type	Possible to Ignore, Skip or Close	Aggressiveness
Embedded	Yes[1]	Low
Popup	Yes	Low
Float	Yes	Low
Notification	Yes	High[2]
Offerwall	No	High

[1] Banner ad only takes a small part of the screen, so it can be considered as ignorable. Other embedded ads like video ad usually can be skipped or closed.
[2] Even though notification ad can be closed or disabled in Android notification bar, we still consider its aggressiveness as "High" because it often affects users reading other notifications.

3.1 System Overview

As mentioned earlier, we design and implement a system called `MAdLens`, which identifies internal third-party ad network modules from a large Android app dataset, maps APIs of ad networks to specific ad types, and finally generates detailed report of all the ad information and summaries for each app in the dataset using static analysis techniques. Our overall approach is summarized in Figure 3. Specifically, `MAdLens` works in 3 steps: module analysis, API mapping and static analysis, which are discussed in detail in the following subsections.

3.2 Data Collection

Before `MAdLens` could get to work, here we introduce our approach for dataset collection first. For fairness, instead of directly crawling from the web pages of Android market, we take another approach by using Android package name enumeration techniques based on a customized pre-defined dictionary. The crawler randomly picks up words from the dictionary, concatenates them into any possible package name of an app. If the name corresponds to an existing app in the market and the installed amount is above 10k, we will download its APK file. In this way, our gathered apps are all popular installations. And our *Android App Dataset* covers all 48 categories of apps in Google Play. Along with an app's apk file, we also collect its meta info including description, user rating and reviews. Since paid apps rarely have third-party ads, it is unnecessary to include any of them in the dataset. Overall, we collected 277,616 free Android apps from Google Play market in March, 2017. We upload an Excel file containing all app names and their meta info to Google Drive, the shareable link is https://drive.google.com/open?id=13F7p0UKS_E7skoxiPCTsSU4h5HywQV7B.

3.3 Ad Network Identification

The first goal of `MAdLens` is to automatically identify all existing ad networks from each app in the dataset. Since most of ad networks are widely used in Android apps, we can assume common ad libraries are shared by many applications and thus code clones can be detected in these applications. Besides, an ad library, which is provided by a certain ad network, is a relatively independent piece of code in apps because there are usually only a few API calls across the boundaries between an ad library and other parts of the app. Based on the two important observations, we leverage the technique described in [19] [20] to detect these frequently used but relatively independent modules in apps. Clustering techniques are employed when measuring the coupling of different modules and finally, clusters are mapped to ad libraries, which are associated with ad networks. We refer the process described above as module analysis in our system.

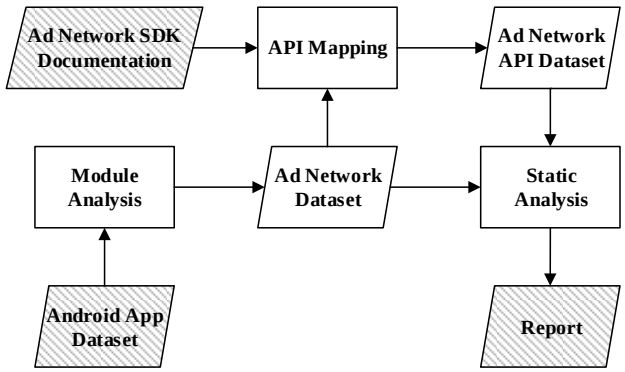


Fig. 3: Overview of `MAdLens`.

TABLE 3: Feature statistics in ad network API dataset.

	Embedded Ad	Popup Ad	Floating Ad	Notification Ad	Offerwall Ad	Total
Number of APIs	407[1]	200	10	29	51	697

[1] APIs of embedded ad are found in both in Java code (299 APIs) and XML (108 APIs) layout resource files in apps (Listing 1 and Listing 2).

In total, we identify over 200 unique ad networks from our dataset, however, only 164 of them have documentation available. Our goal is to perform a comprehensive third-party in-app ads analysis which requires semantic information at API level. In addition, the documentations can help developers place ads in a recommended way. Those SDKs without documentation also increase the difficulty for developers to configure ad networks. So we only add the 164 unique ad networks into our *Ad Network Dataset*.

3.4 Ad Network API Mapping

Since there is no unified ad classification system available at present, we propose methodology of mobile ad classification based on the behavior, UI layout and UI interaction of an ad. To investigate various ad types adopted by apps at a large scale, it is necessary to have the "API mapping" information which includes all the correspondent ad network APIs to specific ad types. Obviously, it is impossible to automate this task since ad APIs are not always well documented, so we have to manually test all relevant APIs from 164 ad networks and map them to the 5 ad types we defined in Section 2.3. This is a non-trivial task that requires huge effort. For those ad libraries with detailed documentation, we write test cases to verify each APIs to make sure that it maps to the a correct ad type. For those undocumented ad libraries we identified from apps, we do API mapping by setting hooks to suspicious APIs with reverse engineering techniques and interacting with the host app to observe its behavior. Overall we get 697 unique ad APIs (Shown in Table 3) and add them into our *Ad Network API Dataset*. On average, each ad network has 4.25 ad APIs.

3.5 Ad Detection

With the two datasets that generated from last two steps, now it is possible to detect different types of ads from different ad networks in Android apps. Recall that there are two approaches for ad APIs to integrated into an app, either in resource file (e.g. xml) or in Java code (See Section 2.2). To get a complete result, both resource files and Java code need to be analyzed thus we leverage on Androguard [21], an lightweight opensource reverse engineering tool for Android in static analysis. For resources files, we first decompile the apk file with Androidguard to get all the Android layout files, then parse all the XML nodes and match them with our 2 datasets. Similarly, for java code, smali code is generated by Androguard after decompiling the apk file and we again match the smali code with the 2 datasets. Note that the code of the ad library itself must be filtered before the code matching process because self-reference to ad APIs may exist inside the code of ad libraries.

In comparison with other Android reverse engineering tools, one of Androguard's major advantages is its resistance to apk obfuscation and hardening. Androguard successfully decompiles 98% of Android apps in our measurement. For those apps which try to hide their code behaviours by adopting app hardening techniques, we characterize them with summary information extracted from the apk file (e.g. numbers of Android activities in code and mainifest file, numbers of Java class and instructions, etc), and eliminate them from our dataset.

4 Data Analysis and Results

To understand the status quo of how in-app ads are involved in the real-world apps, we conducted a large-scale measurement study of in-app ad libraries among the apps from Google Play. Specifically, we collected a total number of 277,616 apps during March 2017. We performed both manual and static analysis on those real-world apps across tens of app categories to better understand the preference of developers in in-app ad selection from various angles.

4.1 The Trend in the Number of Ad Libraries Embedded in an App

We estimate the trend in the number of ad libraries embedded in an app. The last update time of an app indicates the year in which the latest version of the app comes onto the market. We examine whether there exists a positive correlation between the number of ad libraries hosted by an app and its last update year. In Figure 4, we use boxplots to demonstrate the correlation. For each box, its bottom corresponds to the number of ad libraries per app on the 25th percentile, its top corresponds to the ad library number on the 75th percentile, and the line across the box corresponds to the ad library number in the median. These boxplots show that the apps in the years 2008 and 2009 barely have ads, 25% of the apps released in the years from 2010 to 2014 host at least one ad library. Besides, 50% and 25% of the apps released in the recent years from 2015 to 2017 host at least one or two ad libraries, respectively. Thus the figure presents a clear trend that the number of ad libraries embedded in an app increases with the year.

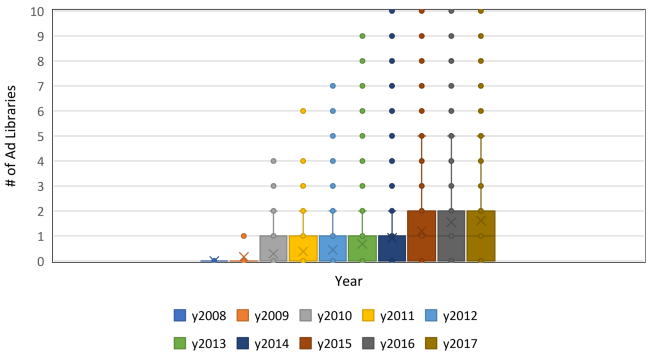


Fig. 4: The trend of the number of ad libraries along with year.

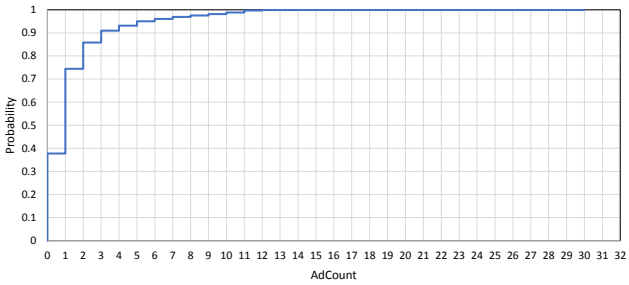


Fig. 5: CDF of the number of ad libraries embedded in an app.

One interesting question is to see how many ad libraries a developer usually places in her app nowadays. Figure 5 shows the cumulative distribution function (CDF) of the number of ad libraries embedded in an app. Up to 37.8% of the apps do not contain any ads, which is surprising given the fact that developers profit from their apps mainly by accommodating ads in their apps. The percentages of apps partnering with one ad library, two ad libraries, and three libraries are about 33%, 10%, and 5%, respectively. This indicates that most apps only involve one or two ad libraries. One possible explanation is that developers avoid to place too many ads[1] in apps since displaying ads may affect user experience, hurdle the promotion of the app, and finally decrease the revenue of the developers.

Implication 1: Developers have more choices about ad networks than several years before. Most developers are conservative about ad placement in their apps given the observation that about 71% of the apps contain at most one ad library.

4.2 Prevalence of Ad Types in an App

We also examine the popularity of each of the five mobile ad types among 277,616 apps. Figure 6 shows a breakdown of the apps by the ad type involved. We can see that 62.2% of the apps in our dataset contain at least one ad type, which is reasonable since after all advertising is one of the most important monetization ways for ad developers. An app could contain multiple types of ads, and 8.5% of the apps accommodate more than two out of the five types of ads.

1. Note that the number of ad libraries included in an app is proportional to the number of ads displaying on the app.

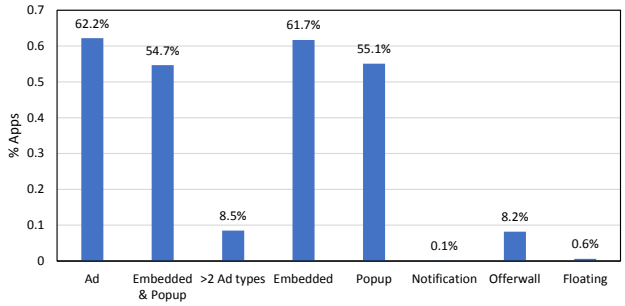


Fig. 6: Breakdown of apps by the ad type.

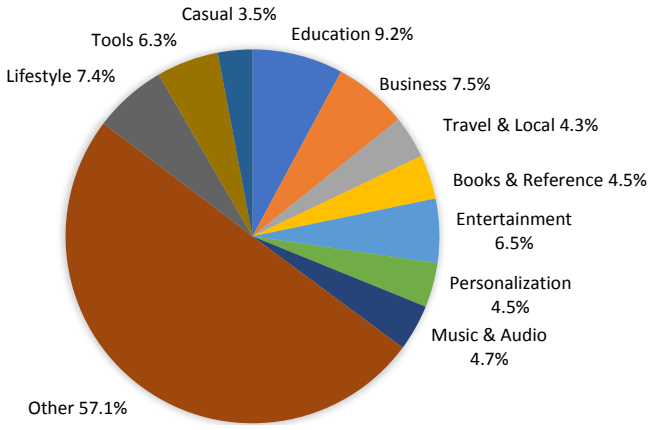


Fig. 7: Top 10 categories with the most apps.

Embedded and *Popup* are among the most popular ad types, and up to 61.7% and 55.1% of the apps contain those two types of ads, respectively. Furthermore, these two ad types usually appear in one app simultaneously, and more than a half (54.7%) of the apps are found to contain both ad types. *Offerwall* is the third popular ad type, which is observed in 8.2% of the apps. *Notification* and *Floating* ad types are the two least popular, with less than 1% of the apps containing them.

Implication 2: Statistically, developers include two lowly aggressive ad types, *Embedded* and *Popup*, simultaneously in more than a half of their apps. In contrast, the two highly aggressive ad types including *Offerwall* and *Notification* are not popular.

4.3 Prevalence of Ad Types in the Apps of the Top 10 Categories

Based on the category tag associated with each app, the apps we collected fall into 48 categories. The top 10 categories with the most apps are shown in Figure 7. It can be seen that apps are quite dispersed across categories. The number of apps in the top 3 three categories, `Education`, `Business`, and `Lifestyle`, only occupy less than 10% each, respectively. The rest popular categories include `Entertainment`, `Tools`, `Music & Audio`, `Personalization`, `Books & Reference`, `Travel & Local`, and `Casual`. Up to 57.1% of the apps fall under the categories out of the top 10 ones.

We further study the prevalence of the five ad types in the apps of the top 10 categories, which is helpful for developers to learn popular practices for placement of different kinds of ads in the corresponding popular categories. Figure 8 provides the details about the ad placement in the top categories. Note that for each app category, the figure provides the percentages of apps in this category which host: 1) ads, 2) ads of more than two different ad types, 3) both *Embedded* and *Popup* ads, 4) *Embedded* ads, 5) *Popup* ads, 6) *Notification* ads, 7) *Offerwall* ads, and 8) *Floating* ads. Please check the legend of the figure for the meaning of each column.

Percentage of apps with ads. Among the top 10 categories, `Business`, `Education` and `Tools` apps are the least likely to contain ads, with 39.6%, 48.0%, and 48.5% having ads, respectively. Section 4.2 shows that 62.2% of the apps contain ads in overall. One reason why these three categories of apps have the below average percentages is that those apps are mainly utility tools and too many ads may distract or even annoy users. In contrast, about 74% to 87% of the apps in the four categories, `Entertainment`, `Personalization`, `Music & Audio`, and `Casual`, contain ads.

Implication 3: The likeliness of an app containing ads depends on the app category to which it belongs, to some extent. The business or utility apps are much less likely to contain ads than the entertainment or casual apps. The app categories featuring young audience usually contain the most ad libraries maybe because of the ad-tolerance characteristic of young people. Thus, developers may decide the ad placement issue based on the category of their apps.

Percentage of apps accommodating at least three types of ads. App developers could place multiple types of ads in their apps for maximizing the profit. The figure shows that the percentage of such apps is not high. Only 1.1% of `Business` apps are embedded with at least three types of ads, while the `Casual` apps have the largest likeliness, with 23.6% accommodating at least three ad types.

Embedded ads and Popup ads. *Embedded* ads turn out to be the most popular ad types, which is true across all categories. 39.6% to 85.7% of the apps in the top 10 categories contain *Embedded* ads. *Popup* ads are quite popular too. Interestingly, the figure shows that a significant proportion (from 36.8% to 71.5%) of the apps across all the top categories contain both *Embedded* ads and *Popup* ads.

Notification, Offerwall, and Floating ads. Comparatively, these three kinds of ad types are much less popular than *Embedded* and *Popup* ad types. *Offerwall* ad type is observed across all top categories, and up to 23.8% of `Casual` apps contain *Offerwall* ad type. In contrast, the *Notification* and *Floating* ads are trivial. `Casual` apps have the largest percentage to contain *Notification* ads, with a value of 0.4%, *Floating* ads mostly appear in `Lifestyle` apps, occupying only 2.9%.

Implication 4: The two lowly aggressive ad types, *Embedded* and *Popup*, are popular with apps in nearly all categories. The highly aggressive ad type, *Offerwall*, is somewhat popular. Developers could consider placing these three types of ads on their apps.

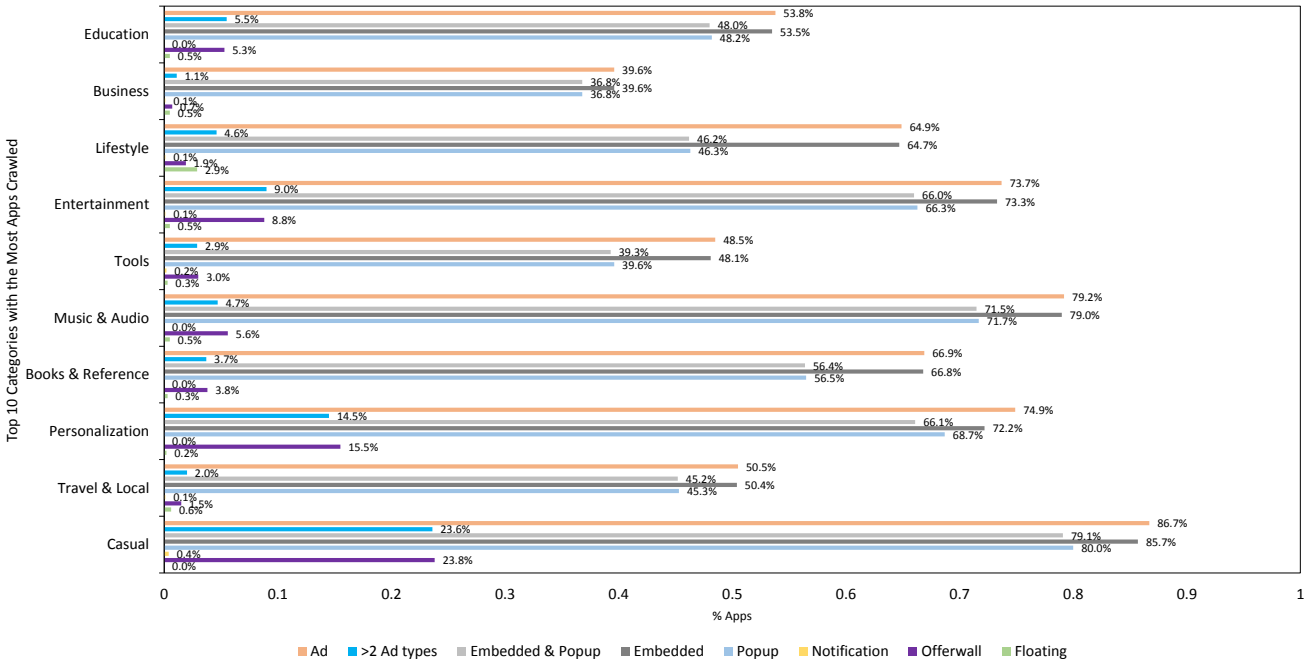


Fig. 8: Breakdown of the apps in the top 10 categories by the ad type.

4.4 Correlation between in-app Ads and User App Ratings

Intuitively, a user could be annoyed by an app embedded with too many ads. Thus it is interesting to examine whether there indeed exist the cause-effect relationships between in-app ads and user ratings. We explore the question by studying the correlation between the number of ad libraries contained in an app and the ratio of users who give bad ratings for the app. Users can rate apps in the Google Play by giving stars. A score with five stars means that a user is very satisfied, while a score with only one star means that a user is extremely dissatisfied. We calculated the percentage of the users who give one star as the ratio of bad ratings. Figure 9 depicts such a correlation. It clearly shows that the ratio of bad user ratings increases along with the number of ad libraries. Bad ratings received on apps with 7 ad libraries or more increase significantly compared to those received by apps with 6 ad libraries or fewer.

Implication 5: Developers are expected to avoid embedding too many ad libraries into an app, empirically no more than 6, which otherwise would dissatisfy the app users and result in bad ratings.

4.5 Ad Network Choice for Apps in the Different Stages of Their Lifecycles

To maximize the revenue, apps in the different stages of their lifecycles may consider different ad networks. Among the metadata information associated with an app, the *downloads* of an app could best indicate the stage of the lifecycle in which the app currently is, at least to some extent. A newly released app typically has few downloads and an app that has been on the app market for a while usually has more downloads.

We examined the correlation between downloads of an app and the number of ad networks that the app is partnering with. Figure 10 depicts such a correlation for all 277,616 apps in our dataset. The x axis denotes the logarithm of the downloads of per app, and the y axis denotes the number of ad networks used per app on average. The curve in the figure shows that statistically the average number of ad networks used by an app has a positive correlation with the downloads of the app. Specifically, an app with 10 downloads uses 0.84 ad networks on average and an app with 1 million downloads adopts 2.79 ad networks on average. That is, a new app usually uses less ad networks and a relatively old app tends to use more ad networks. Since the number of ad networks is reasonably proportional to the number of ads displayed on the app, we could say that a new app tends to display less ads than an old app.

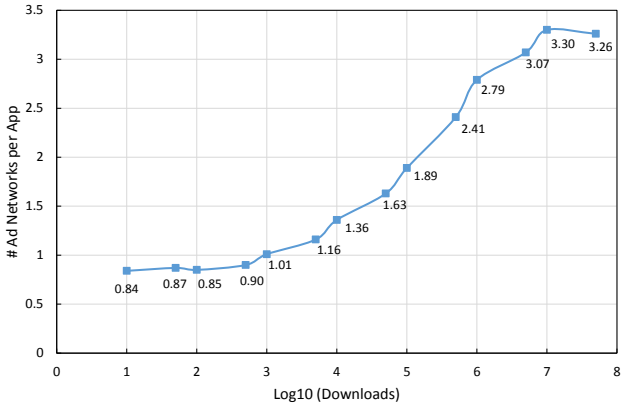


Fig. 10: Ad network choice for apps in different lifecycle stages.

Implication 6: A developer is suggested to use at most one ad network when her app is still at the initial stage and could start using more (2 or 3) ad networks when the app becomes popular, reflected by its downloads.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2953609, IEEE Transactions on Mobile Computing

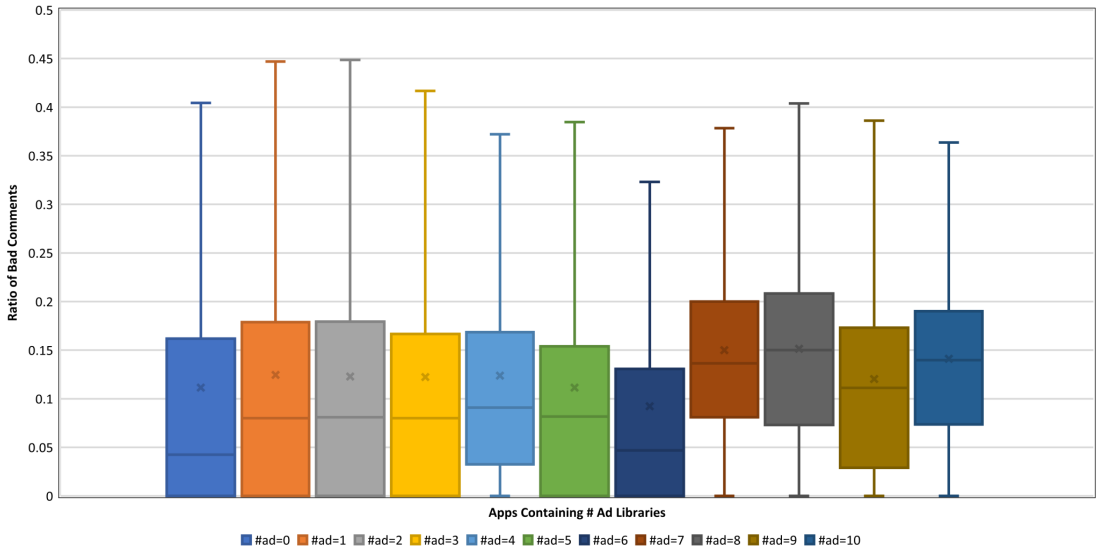


Fig. 9: Correlation between the ad library number in an app and the ratio of bad ratings for the app.

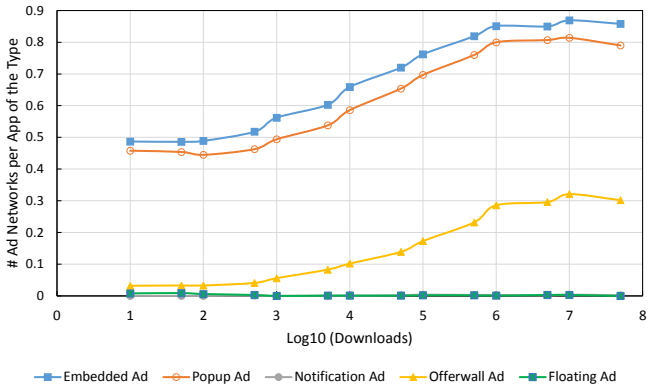


Fig. 11: Ad type choice for apps in different lifecycle stages.

4.6 Ad Type Choice for Apps in the Different Stages of Their Lifecycles

Similarly, we examined the correlation between downloads of an app and the number of ad networks of each ad type that the app is partnering with. Figure 11 depicts such a correlation for each of the 5 ad types. One observation is that statistically *Embedded* and *Popup* are the two most popular ad types for apps with any number of downloads; in contrast, *Notification* and *Floating* are the two least popular ad types, with negligible percentages; *Offerwall* is only popular with the apps with tremendous downloads. Overall, the average number of ad networks of each ad type used per app increases with the downloads of apps.

Implication 7: As the statistical results of the large dataset suggest, we recommend that a developer mainly places *Embedded* and *Popup* ads on her app in whatever lifecycle stage, and could start to place *Offerwall* ads when her app has many enough downloads.

4.7 Impact Analysis of the Aggressiveness of the Ads Hosted by an App

As mentioned before, ad types are classified into two categories: low-aggressiveness and high-aggressiveness. It would be interesting to evaluate the impact of different levels of aggressiveness of the ads on other performance metrics of apps. We choose Pearson correlation coefficient to perform correlation analysis.

As a measure of the linear correlation between two variables, the value of Pearson correlation coefficient is between +1 and -1, where 1 is total positive linear correlation, 0 is no linear correlation, and -1 is total negative linear correlation [22]. Table 4 lists the computed Pearson correlation coefficients.

The table shows that the number of low-aggressiveness ad types and the number of high-aggressiveness ad types included in an app have extremely strong positive correlation between each other, which suggests that an app usually includes these two aggressiveness-levels of ads simultaneously. In addition, both of them have relatively strong positive correlation with the number of ad networks used of the app, which is reasonable given that the two ad types of ad networks combined make the total number of ad networks. Surprisingly, the levels of aggressiveness have no correlations with other metrics of the apps including downloads and ratings. One possible reason is that apps in the different stages could host both aggressiveness-level ads for maximum profit, and app ratings could be affected by many factors and not only by the aggressiveness of the ads in the app.

Implication 8: Developers are recommended to use both low-aggressiveness ads and high-aggressiveness ads at the same time for maximizing their revenues.

5 Ad Mediation

Recently, developers tend to use a new feature called ad mediation which simplifies the integration of multiple ad libraries in Android apps. This new feature utilizes a mediation engine which selects a third-party ad library by invoking its ad API based on some preconditions during an ad request. To use ad mediation, developers first need to embed all the ad libraries they need to make sure that all API invocations from the mediation engine are successful. Our analysis in previous sections do not take the ad mediation feature into consideration, therefore all the API calls

TABLE 4: Pearson correlation coefficients

	# ad networks	# low-aggressiveness ad types	# high-aggressiveness ad types	Downloads	Ratings
# low-aggressiveness ad types	0.568934	1	0.999974	0.004088	0.038284
# high-aggressiveness ad types	0.686385	0.999974	1	0.004583	0.03201

invoked by ad mediation are added into the result set. However, there are some cases where developers do not need all the third-party ad libraries supported by ad mediation or they just forget to embed a few ad libraries, resulting in invalid ad API invocations. Such API invocations never request ads so they should not be counted in our analysis.

To eliminate the inaccuracy caused by ad mediation, in this section, we first introduce the overview of mobile ad mediation showing how it works, then propose a method for ad mediation libraries detection. After successfully identifying ad mediation libraries, we also update our ad network API dataset used in previous sections. Finally, we repeat our analysis on 277,616 apps with the updated ad network API dataset and compare with the results in Section 4.

5.1 Overview of Mobile Ad Mediation

Ad mediation is popular in mobile in-app advertising toady. An ad mediation library allows a developer to coordinate various ad libraries in an application and display ads from one of the ad networks based on factors such as ad availability and value. Ad mediation also reduces SDK bloat. Having a lot of SDKs integrated can slow down apps and affect performance. The more SDKs in an app, the more unpredictable and inconsistent the app's user experience will be. Instead of manually integrating multiple ad networks, a mediation solution requires just one SDK, aggregating all those ad networks inside it. This saves developers coding time and minimizes the SDK bloat. According to the introduction of Google AdMob Mediation [23], AdMob mediation platform can manage up to 33 ad networks.

Figure 12 describes how mobile ad mediation works when adding mediation feature into ad networks. First the user launches the app and triggers an ad request. The mediation platform then receives the user's parameters. Later the mediation platform's optimization engine sequentially sends the parameters to its mediated ad libraries by invoking the corresponding ad APIs in the order of revenue from high to low. Once an ad is returned, the sending process ends and the returned ad must be the highest-paying one which is served to the user finally. Ad mediation increases fill rates, maximizes ad revenue, and simplifies the SDK integration process, therefore making it a better solution for developers to monetize their apps. The main advantages [24] of using ad mediation is to centralize access to ad networks with just one SDK, instead of managing various different SDK integrations directly.

5.2 Ad Mediation Library Detection

Previous analysis scan smali code for all the ad API calls tagged with ①②③ as shown in Figure 12. However, a successful ad request requires developers to integrate ad

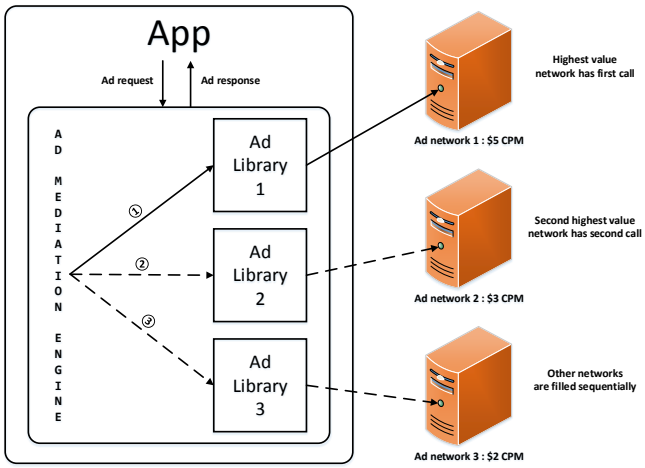


Fig. 12: Overview of mobile ad mediation.

library SDK 1,2,3 in the app. If any of them doesn't exist, an "SDK not found" error will occur indicating the API call is invalid. In order to filter these invalid ad API calls from previous result set, we need to detect ad libraries with ad mediation feature and validate the integration of ad library SDKs that used by mediated ad networks. Next we describe how we improve our approach for more accurate results.

We improve our methodology for detecting in-app ads in Android apps by using function call graph instead of simply scanning smali code. As mentioned in Section 3.5, to detect ads we scan the generated smali code line by line and match the specific statements of API calls. Such approach suffers from an accuracy problem because ad libraries may have self-call and mediation behaviors, and some mediated ad API may lead to an invalid call into an empty function body, resulting in false positives for in-app ad detection. To eliminate such false positives in our previous results, we take a callgraph-based approach. The call relationship of the target ad API function can be clearly captured in the function call graph, thus improving the accuracy of ad detection. Although developers are likely to use various common obfuscation techniques like Identifier renaming, Reflection, and Encryption (IRE) to protect their source code [25], however, class names of most third-party ad libraries do no support obfuscation due to compatibility reasons. As for the dynamically loaded third-party ad libraries, developers embed them in two ways. In the first situation, developers actually use the reflection mechanism. We can detect such ad libraries through function call graph. In the second situation, third-party ad libraries download the jar packages from the remote server and embed the SDKs. We cannot handle this case by static analysis. There is a solution that we dynamically run the apps and randomly

click on the screen to trigger the ads. We can use the UI-based methodology proposed in [26] to detect the triggered ads. In the meanwhile, we capture the network flow and analyze which ad network the requested ad belongs to. We leave this dynamic detection for future work. Based on call graph of Android apps and the extracted layout files in APK files, we redesign `MAdLens` to detect Android in-app ads by taking ad mediation into consideration. Our improved approach refines the results in Section 4 and it includes the following steps:

Step 1: **Package name extraction for ad mediation.**

Based on our observation, we find that when an ad API is invoked by an ad mediation, the node representing this API in the call-graph always has no child node. That is because the library code of the ad API is not actually imported. So we utilize the pseudo-code in Algorithm 1 to extract the package names of the functions that invoke such APIs. Specifically, if the package name of the invoking function is not same as the package name of the invoked API (coming from different ad libraries), we record the package name and mark it as ad mediation. Finally, we put all the package names of ad mediations into an ad mediation list, and update our *Ad Network API Dataset* (See Section 3).

Algorithm 1 Get Ad Mediation Package Name

Input: Apk files of Android apps $|A|$
Output: Ad mediation name list $|M|$

```
procedure GetAdMediationName(|A|)
    initialize ad mediation name list |M| with φ;
    for all apk files A_i in |A| do
        for all ad APIs I_i in A_i do
            N_i ← function call graph node of I_i
            if N_i has no child function node then
                M_i ← ad library name which invokes I_i
                S_i ← ad library name of I_i
                if M_i ≠ S_i then
                    |M| ← M_i
                end if
            end if
        end for
    end for
    return |M|
end procedure
```

Step 2: **Call graph scanning.**

In this step, we search the function call graph to find the third-party ad libraries embedded in Android apps by matching the package names in our *Ad Network API Dataset*. Given a function node in the call graph of an app, we first check whether this node has any sub function node. An empty child node set indicates that the corresponding API of this node is actually mediated by an ad adapter and its ad information results depend on the ad mediation. So we ignore these ad APIs in our result. On the contrary, if the child node set is not empty, we next check its parent function node set. 1) If parent node doesn't belong to any ad libraries, indicating the corresponding ad API is directly invoked by an Android app in a regular way as depicted in Figure 13, we then add the name of the ad library and the ad types of the ad API to *Regular Result Set*. 2) If the parent node belongs to the same ad library with the ad API, again we simply ignore the ad API in our result. 3) If the parent node belongs to a different ad library from the ad API, indicating this API is called by an ad mediation, we then add the ad mediation name and the ad types of the ad API to the *Mediation Result Set*.

Step 3: **Merging the results.** After we obtain the *Regular Result Set* and the *Mediation Result Set*, we take the intersection of the ad types in these two result sets. Then we respectively append the ad library name in the *Regular Result Set* and the ad mediation name in the *Mediation Result Set* with the merged ad types to generate our *Final Result Set*. Our research reveals the fact that developers may choose only a few of the ad types when integrating ad mediation into their apps although the mediation adapter class often supports nearly all types of APIs that provided by other ad networks. As shown in Figure 14, in the *Mediation Result Set*, the ad library A supports the embedded and the popup ad types while the ad library B just supports the embedded ad type. Because the ad library A is mediated by the ad library B, there exists only one ad type (embedded) in the application.

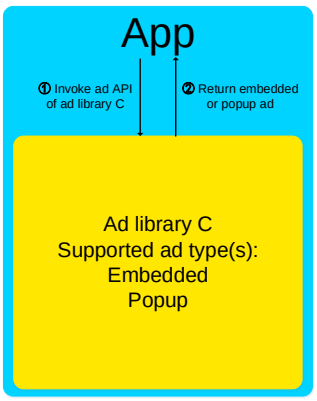


Fig. 13: Invoke ad API in a regular manner.

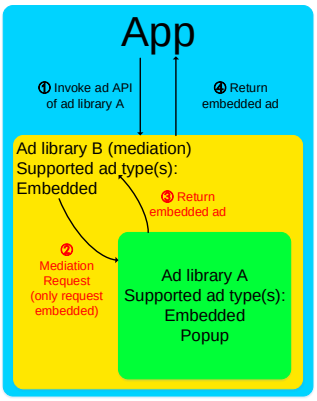


Fig. 14: Invoke ad API in a intermediate manner.

Step 4: **Ads in the layout file.** As a final step, we decompile Android apks to get Android layout files and parse all the XML nodes with our *Ad Network API Dataset*. Later, we extract the advertisements inserted in layout files and append these ad information to the *final result set*.

5.3 Updating Ad Network API Dataset

As described in Section 5.2, in order to find API calls from ad mediation to third-party ad library in the function call graph, we need the package names of ad libraries with ad mediation feature and the smali code of these API calls. Previous *Ad Network API Dataset* only has the package names of the third-party ad libraries and the smali code of their ad APIs. In this section, we discuss about updating our *Ad Network API Dataset* by adding information related to ad mediation.

We collect all the ad APIs corresponding to the nodes in function call graphs which have no subfunction node. These APIs are expressed in smali code that contain a class name and a method name. Then we leverage Androguard again to search all the relationship of the function calls with

the class name and the method name in the smali code as the parameters. Note that we filter an invoking relationship when the invoked and the invoking package name are the same. Generally a package name of an ad mediation includes keywords "mediation" or "adapter", for example, *com.google.ads.mediation* and *com.facebook.ads.internal.adapters*. But we find some particular cases where the API code is dynamically fetched from ad servers when the app is running. In this situation, we cannot infer which ad types are integrated in the app. However, we only find 142 of our 277,616 Android apps with dynamic code loading feature, so we ignore these apps in our result. For the newly discovered ad libraries, we search their SDK documentations and add the supported APIs into our *Ad Network API Dataset*.

Previous dataset contains 164 ad networks and 697 ad APIs, and we update it to 207 ad networks and 838 ad APIs (Shown in Table 5). Of 207 unique ad networks, we find 64 ad networks with mediation feature. On average, each ad network has 4.05 ad APIs. With the updated *Ad Network API Dataset*, we perform a large scale ad detection on our previous Android App Dataset by following the above-mentioned four steps.

5.4 Comparison of Results

This section gives a comparison of the results between with ad mediation and no ad mediation introduced. We estimate the impact of ad mediation feature on ad libraries and ad types.

Figure 15 depicts the CDF of the number of ad libraries embedded in an app when using our methodology to detect ad mediations. Statistics show that the percentage of apps which do not contain any ad increases from 37.8% to 44.77%. This reflects that it is inaccurate to simply scan the smali code of apps. In some cases, we may only find the integrated third-party SDK embedded by the ad mediation, but the ad APIs are not invoked indeed. In our improved methodology, these situations correspond to empty function bodies so we can skip adding these ad APIs to the result set. However, the percentages of apps partnering with one ad library and three libraries decrease to 32.45% and 3.47% respectively. Recall that implication 5 suggests developers not to embed too many ad libraries into an app, which otherwise would dissatisfy the app users and lead to bad ratings. Ad mediation can centralize access to ad networks with just one SDK, instead of integrating various different SDKs directly, thus decrease the number of ad libraries contained in android apps.

Implication 9: It is better for developers to use ad libraries with ad mediation feature when managing multiple SDKs. Ad mediation not only reduces developers' burden in embedding multiple ad libraries, but also decreases the number of ad libraries in apps which may avoid bad ratings and improve user experience.

Table 6 shows the decrease of the percentage of apps containing different ad types. As demonstrated in the table, the percentage of apps having more than 2 ad types decreases from 8.5% to 6.15%. Meanwhile, the percentages of apps containing *Embedded*, *Popup* and *Offerwall* ads go down by 10.35%, 20.11% and 1.27% respectively. This also proves the accuracy of our methodology which can eliminate the ad APIs that are not invoked indeed. Among them, the percentage of apps containing *Popup* drops the most. The reason could be that while the mediated third-party ad libraries support multiple ad types such as *Embedded* and *Popup*, the ad mediation only provides *Embedded* ads. In this situation, there is only one ad type, *Embedded*, in the result set.

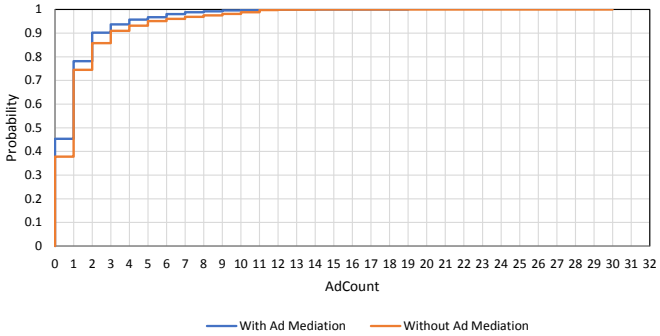


Fig. 15: Comparison of CDF of the number of ad libraries embedded in an app.

TABLE 6: Comparison of percentages of apps containing ad types.

Ad Type	Percentage with no ad mediation	Percentage with ad mediation
Embedded	61.7%	51.35%
Popup	55.1%	34.99%
Offerwall	8.2%	6.93%
More than 2 ad types	8.5%	6.15%

Implication 10: When selecting ad libraries with ad mediation feature, it's better for developers to compare the supported ad types between different ad mediations, which helps developers meet their needs for multiple ad types to maximize their benefits.

6 App View Hierarchy and Ad Types

In previous sections, we have analyzed the ad types from several aspects such as percentages and bad ratings. We want to explore more with the ad types from the perspective of user interaction which implies one of the most important evaluation indexes for Android apps, the user engagement. In this section, we target at the relationship between the view hierarchy of Android apps and the ad types.

6.1 User Engagement of Android Apps

Android apps are mainly written in Java. The Java code is compiled to compressed bytecode in a .dex file. The bytecode runs in the Dalvik virtual machine, a virtual machine similar to the Java Runtime Environment (JRE). Apart from Java, some applications are also allowed to be written in native code. Android apps are composed of components containing four types: *activity*, *service*, *broadcast receiver*, and *content provider*. Content providers manage the access to data while services are running in background. Broadcast receivers receive system events (such as reboot completed or an SMS received, and so on) from registered system services.

TABLE 5: Comparison of feature statistics in ad network API dataset with and without ad mediation.

Number of APIs / Ad Type / Ad Mediation	Embedded Ad	Popup Ad	Floating Ad	Notification Ad	Offerwall Ad	Total
With Ad Mediation	481[1]	260	10	30	57	838
Without Ad Mediation	407	200	10	29	51	697

[1] APIs of embedded ad are found in both in Java code (356 APIs) and XML (125 APIs) layout resource files in apps (Listing 1 and Listing 2).

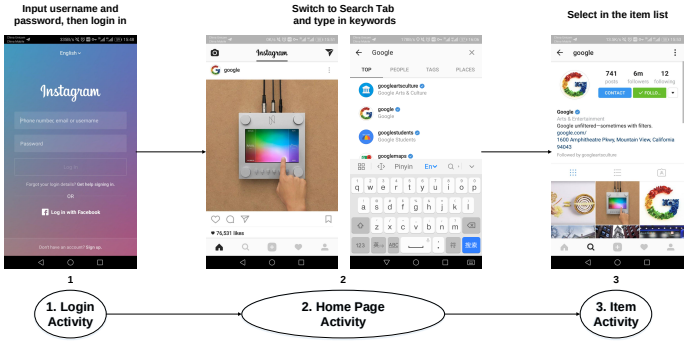


Fig. 16: Activity transition diagram showing the interaction with the popular Android app, Instagram.

Whenever a system event is triggered, except some special cases for instance broadcast abortion, the code specified in the broadcast receiver is run to respond to the system service. As Android apps are GUI-centric, a typical Android app consists of separate screens corresponding to activity components, which is functionally equivalent to windows in a conventional desktop GUI [27].

When interacting with an app, users transit between different activities by navigating typical GUI elements, such as toast (popups), text boxes, text view objects, list items, and progress bars. In Figure 16, we describe how users interact with a popular Android app, Instagram. On the top we give the textual explanation of users' actions, in the middle we have a real screenshot, and on the bottom we draw the transitions between the activities. Initially the app is in the Login Activity. When the user inputs her username and password and login in, the app transits to the Home Page Activity (another different screen). Then the user switches to the Search Tab (the same screen) and searches for items by typing in item keywords. After the user presses down on the screen, the layout changes to show the details of the selected item as the app transits to the Item Activity.

Measuring the mobile KPIs [28] helps developers gain the best insights on their apps, optimize effectively, and generate more revenue. One of the best mobile KPIs is to measure how many screens the users visit each time they use an Android app, and how they interact with those screens. It's usually a good indication when the users engage with a significant number of screens per session, as this shows that an app is interactive and useful to people. As illustrated in the last paragraph, the activity transitions naturally form a graph. From the perspective of graph search, if we regard the Login Activity as the start node, users who explore the Item Activity go more deeply in the activity transition graph than those who just explore the Login Activity. The deeper the users navigate in an Android app, the more screens the users visit (users who explore the Item Activity also explore the Login Activity and the Search Tab of the Home Page Activity). In conclusion, users with high engagement and interaction usually explore more deeply in the activity transition graph.

6.2 Depth of Ads in App View Hierarchy

In this section we present the two elements of our methodology: Static Activity Transition Graph, which displays the view hierarchy of an Android app, and Ad API Call Locating, whose main goal is to trace which activity an ad API call occurs. Combining the Static Activity Transition Graph and the generated activity name of the Ad API Call Locating, we use graph search algorithm to map the location of the activity, where the ad API call occurs, in the view hierarchy of an Android app. We first give the definition of the Static Activity Transition Graph, and then introduce how the Static Activity Transition Graph is constructed. Next we show how to trace the activity name where the ad API call is located and finally how we map the two elements to get the depth of the occurrence of the ad API call.

Static Activity Transition Graph. The Static Activity Transition Graph (SATG) [29] is a graph $G_S = (V_S, E_S)$ where the set of vertices, V_S, represents the app activities, while the set of edges, E_S, represents possible activity transitions. SATG helps understanding programs as it provides a view hierarchy of the high-level application work flow. SATG is automatically extracted from Android apps using static analysis (to be specific, taint analysis). The programming model of Android apps is based on callbacks in response to user interactions or background services invoked by the Android framework, which differs from the traditional model based on main function. If the transition from activity A to activity B is caused by user interaction, the methods in call graph associated with A indirectly invoke B. The transition is implemented in a generic intent passing logic. Consequently, to get a SATG, the construction process can be achieved by data-flow analysis, or specifically, taint tracking. Take the transition between the A and B activities as an example, we taint B when setting up taint analysis. If the taint analysis can find an actual invocation path starting from A, it means that activity A reaches B, so an A to B edge is drawn in the SATG. Activity transition is realized based on *Intent* objects. As described in the official Android documentation [30], an Intent is an abstract description of an operation to be performed. It can be used with *startActivity* method or other similar methods like *startActivityForResult* and *startActivityIfNeeded* to launch an activity by passing the intent as a parameter, or with *startService* and *bindService* method to communicate with a background service by sending broadcast messages. Intent can be thought of as the glue between activities. We now illustrate how to use

taint tracking to construct a SATG. Initially, the data-flow analysis tags Intent object declarations as taint sources. Then the taint tracking looks for the tagged sinks: *startActivity*, *startActivityForResult*, and *startActivityIfNeeded*. With the tagged sources and sinks, the taint tracking checks whether tainted sinks are reachable from sources. For all the detected (source, sink) pairs by the taint tracking, an edge is added in the SATG. Generally speaking, the principle for constructing the SATG is to test the reachability from Intent declaration points as sources to activity launch requests as sinks.

Ad API Call Locating. To trace which activity an ad API is invoked or embedded, we first decompile all the layout files, extract all the activity names and record the main activity of an app. Then for each invoked ad API, we search all the methods which call the ad API. After we get the call relations, we obtain the package names of the calling methods and match all the activity names to check if any package name equals to an activity name. If an activity name is found, we record the activity name and the ad API information. Next for each ad API defined in XML layout files, we match all the string values in layout files to check if the package name of an ad API equals to any string value, that is, the ad API is embedded in a layout file. If an ad API is defined in a layout file, we read its hexadecimal resource ID and scan the smali code of the app to look for which method uses this resource ID. Same as the operations with invoked ad APIs, after we get the package name of the method and obtain the activity name, we record the activity name and the ad API information. Finally we gather all the results of both invoked and XML-defined ad APIs.

Breadth-First Mapping. We now show how we use graph search algorithm to map the SATG and get the depth of the occurrence of the ad API call. To compute the depth, we actually want to find the shortest path between the start node and the target node. So we employ breadth-first search algorithm in this step. From the interactive point of view, there are two kinds of start nodes in our methodology, the first is the main activity defined in AndroidManifest.xml, and the second is the registered broadcast receiver.

Algorithm 2 precisely describe the breadth-first mapping methodology. First we construct the SATG of an Android app (line 4) and extract the nodes representing the broadcast receivers. But we only record those nodes which only have outward edges as *startBroadcastReceivers* (line 5). Then we fetch the main activity M_i from the AndroidManifest.xml file in the app's APK (line 6). For each ad API call C_i, we get the activity O_i where C_i occurs (line 8). Next we compute the length of the shortest path between the start nodes and other activities which contain ad API calls. We compute between M_i and O_i. Also we compute between each *startBroadcastReceiver* and O_i in the SATG. For each O_i, we start from M_i or a *startBroadcastReceiver* in the SATG using breadth-first search. Once we've found that the activity name of the current node equals to O_i's, we return the depth of the current node. Note that we choose the minimum length among the computed results of M_i and *startBroadcastReceivers* as the depth D_i of O_i (line 9-15). Finally, we write ad type of C_i with the depth information D_i to the result set (line 16).

Algorithm 2 Breadth-First Mapping

Input: Apk files of Android apps $|A|$
Output: Depths of ad API calls $|D|$

```
1:  procedure BFM(|A|)
2:      initialize depths of ad API calls |D| with φ;
3:      for all apk files A_i in |A| do
4:          SATG_i ← SATG of A_i
5:          |SBR|_i ← Start Broadcast Receivers of SATG_i
6:          M_i ← main activity of A_i
7:          for all ad APIs calls C_i in A_i do
8:              O_i ← activity where C_i occurs
9:              D_i ← BFS(SATG_i, M_i, O_i)
10:             for each broadcast receiver BR in |SBR|_i do
11:                 D_br ← BFS(SATG_i, BR, O_i)
12:                 if D_br < D_i then
13:                     D_i = D_br
14:                 end if
15:             end for
16:             |D| ← {ad type of C_i, D_i}
17:         end for
18:     end for
19:     return |D|
20: end procedure
21:
22: procedure BFS(S_i, Node, O_i)
23:     initialize a queue Q with φ;
24:     initialize a visited set V with φ;
25:     initialize a depth set Depth with φ;
26:     Depth(Node) = 0;
27:     if Node equals to O_i then
28:         return Depth(Node)
29:     end if
30:     V ← {Node}
31:     put Node on Q;
32:     while Q ≠ φ do
33:         v ← head(Q)    // head(Q) is the first item on Q
34:         for all neighbors w_i of v in S_i do
35:             if w_i ∉ V then
36:                 Depth(w_i) = Depth(v) + 1;
37:                 if w_i equals to O_i then
38:                     return Depth(w_i)
39:                 end if
40:                 V ← V ∪ {w_i}
41:                 put w_i on Q;
42:             end if
43:         end for
44:         Delete v from Q;
45:     end while
46: end procedure
```

6.3 Mapping Activity Transition Graph

We now proceed to presenting the implementation of our methodology for computing the depths of the occurrences of ad API call.

SATG. A^{3}E [31] defines SATG and proposes an automatic Android app exploration tool which contains a SATG constructor. A^{3}E is a static analysis tool based on Dalvik bytecode, and it tags intent object declarations as sources and activity life-cycle methods as sinks for the construction of SATG. Given that the highest Android version A^{3}E supports is 2.3.7 (Gingerbread), it is obviously out of date for analyzing current apps as the newest Android version has updated to 8.x (Oreo). So we choose to write our own code to construct the SATG instead of using their designed constructor.

We leverage Androguard 3.1.0 (released on Mar 16, 2018) to implement our code. First we get the call graphs of all activities and broadcast receivers. We determine whether a class is a broadcast receiver according to the onReceive method. Then for each call graph of an activity class or a broadcast receiver class, we find all the nodes labelled with start-like invocations, for example, startActivity. Next for such a start-like invocation, we analyze the call graph to see which method of the class contains the occurrence of the invocation. Finally, we scan the smali code of the method and extract the data stored in the intent parameter register which shows the information of the destination activity or broadcast receiver in an activity transition relation. In this way, we tag the activities or the broadcast receivers which contain occurrences of start-like invocations as sources and the activities or the broadcast receivers whose information are passed to intent parameters as sinks for our SATG construction.

Breadth-First Search. As mentioned before, we see the main activity defined in AndroidManifest.xml or the broadcast receivers which only have outward edges as the start node. In the beginning, we maintain a set of all broadcast receivers. Each time we find an intent which transits to a broadcast receiver, we delete this destination broadcast receiver from the set. After iterating all intents, the remaining elements in the set are all start broadcast receivers. Then we can compute the length of the path (depth) from the start nodes to the target activity which integrates ad APIs.

With the constructed SATG using Androguard and the activity names of ad API calls, we implemented Breadth-first mapping using a standard breadth-first strategy: if the current node is not our target node, we sequentially check its neighboring nodes. This process continues until we find our target node or there is no more node to explore.

6.4 Data Analysis and Results

Figure 17 shows the CDF of layer depths of activities, which integrate five types of ad APIs, from main activities. In our dataset of 277,616 apps, we only get one app embeds a *Floating* ad API and we can't find a path from the activity which integrates this *Floating* ad API to the main activity. So the percentage of layer depth for the *Floating* ad remains zero in the CDF. For this reason, we just talk about other four ad types: *Embedded*, *Popup*, *Notification* and *Offerwall*.

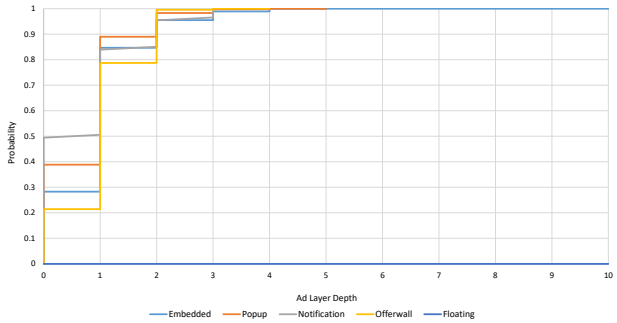


Fig. 17: CDF of layer depths of activities integrating five ad types from main activities.

As we can see, more than 95% of *Embedded* (95.5%), *Popup* (98.3%), *Notification* (95.4%) and *Offerwall* (99.58%) ads locate in the zero activity (main activity), the first activity and the second activity of Android apps, which needs at least one and two user interaction(s) respectively. The percentages of four ad types all reach nearly 100% at the fifth layer depth which needs at least five activity transition operations from main activity to start the fifth activity integrating ad APIs.

Implication 11: Most ads locate in activities of first three layer depths and almost all the layer depths of ad APIs are no more than five. Most apps may not be designed with deep activity transitions. However, developers had better not put ads in activities which need more than 5 user interactions or activity transition operations. It gains little benefit to put ads in deeper activities for that most users may never reach such activities with very deep layers, but these unnecessary ads will definitely make apps become bloatwares.

We observe a big step from layer 1 to layer 2 in the CDF curve of *Offerwall*, which depicts an increase by 20.9% of *Offerwall* ads located in the second activity layer depth. Although not obvious, *Notification* rises by 11.5%, which is also greater than *Embedded* (10.8%) and *Popup* (9.3%). Compared to low aggressiveness ad types, developers usually choose to put a more proportion of high aggressiveness ad types in activities needing more user interactions.

Implication 12: Developers may put more aggressive ads on deeper layouts for that users who explore more activities could have higher user engagement and may be more tolerant to aggressive ads.

7 Limitation

We acknowledge the following limitations of our methodology.

Ad Number Approximation. Our system `MAdLens` takes a static approach to analyze Android apps, so it has inherited limitations of static analysis. `MAdLens` measures the number of ads inside a certain app by counting the occurrence of ad APIs. However, this is an approximation due to code control flows and not every ad API can be necessarily triggered at runtime. While dynamic approaches are capable of capturing an app's runtime behaviors, it is still very challenging to analyze apps at a large scale. Besides, our experiment shows that most ad networks have encrypted their network traffic with servers, making it even more

difficult for dynamic analysis to monitor ad behaviors from network level. One possible dynamic solution is to apply image recognition techniques in identifying ad activities on UI, and we leave it for future work.

Android App Hardening. Although we have apply some countermeasures to app hardening techniques by characterizing these apps with code summary information (Section 3.5), in practice this approach could not cover all the cases and thus may lead to false negatives, which affects the accuracy of our measurement.

8 Related works

8.1 Mobile advertising

Mobile advertising has become a hot topic for research recently. The privacy issues related to ad networks and associated ad libraries have been the focus of many existing works. Grace et al. [7] study the potential risks (e.g. privacy leakage) posed by embedded or in-app ad libraries. Demetriou et al. [6] and Meng et al. [32] estimate the risk associated with user data exposure to advertising libraries in Android apps and show that malicious ad libraries can infer sensitive information. To address the privacy concerns, many solutions are proposed to isolate advertising from application. Pearce et al. [4] use privilege separation to identify advertising-related over-privilege. Shekhar et al. [5] implement privilege separation by extracting ad services from recompiled apps. Zhang et al. [33] provide a general approach to isolate third-party ad libraries into a separate process and implement privilege, display and input isolation. Liu et al. [34] first use machine-learning to detect ad libraries and then use code instrumentation to de-escalate their privileges. Besides, to balance user privacy and mobile advertising, Leontiadis et al. [8] propose a privacy protection framework to "achieve an equilibrium" between the developer's revenue and the user's privacy based on the establishment of a feedback control loop that adjusts the level of privacy protection.

In addition to user privacy, Crussell et al. [9] study mobile ad fraud perpetrated by Android apps and identify two fraudulent ad behaviors in apps. One is requesting ads while the app is in the background and the other is clicking on ads without user interaction. Liu et al. [10] study a kind of mobile ad fraud called "placement fraud" and design a system for automated detection. Nath [11] characterize user targeting strategies of top ad networks and measure their effectiveness by developing a tool called MAdScope. Ad targeting has also been discussed in [12] and [13].

Besides, some of the recent works study the impact of ad networks on an app's rating. Ruiz et al. [35] find that integrating multiple ad networks can lead to negative impact on user experience. Fu et al. [36] collect user feedback to explain why people dislike a given app.

8.2 Online advertising

Online advertising has been studied for decades from many perspectives with focus on security and privacy. Stone-Gross et al. [37] describe how online advertising ecosystem works, study the associated security issues from network level and introduce known types of fraud, including impression spam, click spam, competitor clicking, conversion (action) spam and misrepresentation. Similarly, Xu et al. [38] study click fraud in online advertising and provide a novel approach for advertiser to detect and evaluate click frauds against their campaigns.

Apart from ad fraud, other various topics have also been discussed. Li et al. [39] study malicious activities behind online advertising and provide mitigation using prominent features they identify from malicious advertising nodes and their related content delivery paths. Apostolis et al. [40] analyze to what extent users are exposed to malicious content through online advertisements. Malicious advertising known as malvertising, exhibit different behaviors: drive-by downloads, deceptive downloads and link hijacking. Phillipa et al. [41] characterizing the value of user information and privacy to advertising revenue by measuring network traffic.

8.3 Ad library detection

Kuhnel et al. [25] collect publicly available ad APIs and scan smali code of Android apps to detect ad libraries. But their heuristics for ad libraries only recognize embedded ad type. Liu et al. [34] propose a method to find the usage of ad libraries in the obfuscated code and develop a system called PEDAL to de-escalate privileges of ad libraries. Zhang et al. [33] design a separated activity called AFrame to isolate advertisements in Android apps. AFrame achieves process, permission, display and input isolation for privilege restriction of third-party ad libraries. However, none of them take the ad mediation feature into consideration.

Ruiz et al. [42] analyze different versions of Android apps collected over 12 months and explore the various expenses involved in updating ad libraries. In their findings of ad library updates, they find some ad libraries, for example Mobclix, which add mediation capability to coordinate other ad networks. They just know the prevalence of adding mediation feature, while we implement an algorithm to extract package names of ad libraries with mediation feature.

8.4 Android apps exploration

Existing works have conducted both static and dynamic exploration of Android apps for app analysis and testing tasks. Rastogi et al. [27] propose a framework which can automatically and dynamically explore the app GUI using fuzz testing and intelligent black-box execution by feeding a stream of random inputs. Azim et al. [29] develop a tool to extract *Static Activity Transition Graph* (SATG) and adopt a systematic exploration strategy to perform static analysis on Android apps. Their works focus on exploring the apps for security issues such as privacy leaks and malicious functionality. While our work, leveraging Azim's SATG which shows the app view hierarchy, studies the implication of ad types and app structure.

Overall, none of existing works cited in this section analyzes the potential impact of certain ad networks or ad types on app popularity at API granularity. To the best of our knowledge, we are the first to provide a unified classification of mobile in-app ads and practical guidelines for mobile developers to monetize their apps.

9 Conclusion

App monetization could be the ultimate goal of most app developers. However, app developers lack the guidelines on how to maximize their app revenues. In this work, we aim to provide insights from developers about how to optimize their app monetization with optimal ad placement choices. To this end, we study the in-app advertising ecosystem from a developer's perspective.

We collected 277,616 Android apps and developed a static analysis framework to extract ad libraries of different ad networks from those apps. Besides, by utilizing function call graph, we distinguish ad networks with ad mediation feature from those without this feature which simplifies the management of multiple ad libraries. We also abstract ad relevant APIs inside a SDK to ad types. With the extracted information by both manual labeling and static analysis, we further perform a large scale measurement study and uncover the current practice about ad placement.

We found that most developers are conservative about ad placement and about 77% of the apps contain at most one ad library. In addition, the likeliness of an app containing ads depends on the app category to which it belongs. Furthermore, embedded and popup ad types are found to be quite popular with apps in nearly all categories. Our results also suggest that it's better for developers to embed at most 6 ad libraries to avoid affecting user experience. A better solution for developers is to use ad libraries with ad mediation feature when managing multiple SDKs. Ad mediation decreases the number of ad libraries in apps which may avoid bad ratings and improve user experience. Also, developers can learn from popular practices that they may use at most one ad network at the initial stage of their apps and use 2 or 3 ad networks later. From the perspective of user interaction, developers are suggested to put more aggressive ads on deeper layouts for that users who explore more activities could have higher user engagement and may be more tolerant to aggressive ads. Our research is the first to reveal the preference of both developers and users for ad networks with ad mediation feature and ad types.

References

[1] "Gartner Says Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017." http://www.gartner.com/newsroom/id/3725117.

[2] AppBrain, "Free vs. paid Android apps." http://www.appbrain.com/stats/free-and-paid-android-applications, 2017.

[3] VisionMobile, "App Economy Forecasts 2014-2017." December 2014.

[4] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proc. of the 7th ASIACCS*. Acm, 2012, pp. 71–72.

[5] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications." in *USENIX Security Symposium*, vol. 2012, 2012.

[6] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," in *Isoc Network and Distributed System Security Symposium*, 2016.

[7] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. of the WiSec'12*. ACM, 2012, pp. 101–112.

[8] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: Balancing privacy in an ad-supported mobile application market," in *Proc. of the 12th HotMobile*. ACM, 2012, p. 2.

[9] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proc. of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 123–134.

[10] B. Liu, S. Nath, R. Govindan, and J. Liu, "Decaf: Detecting and characterizing ad fraud in mobile apps." in *NSDI*, 2014, pp. 57–70.

[11] S. Nath, "Madscope: Characterizing mobile in-app targeted ads," in *MobiSys*, 2015, pp. 59–73.

[12] J. Lee and D. H. Shin, "Targeting potential active users for mobile app install advertising: An exploratory study," *International Journal of Human-Computer Interaction*, 2016.

[13] T. Book and S. W. Dan, "An empirical study of mobile ad targeting," *Computer Science*, 2015.

[14] B. He, H. Xu, L. Jin, G. Guo, Y. Chen, and G. Weng, "An investigation into android in-app ad practice: Implications for app developers," 2018.

[15] "Mobile Ads SDK for AdMob," https://developers.google.com/admob/.

[16] "Google AdMob Ad Formats," http://admob.com/home/admob-advantage/#engaging_ad_formats.

[17] "Airpush Ad Formats," https://airpush.com/developers/.

[18] "Chartboost Ad Formats," https://www.chartboost.com/advertisers/#all-ad-formats.

[19] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces." in *NDSS*, 2016.

[20] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. of the 3rd CODASPY*. ACM, 2013, pp. 185–196.

[21] "Androguard," https://github.com/androguard/androguard.

[22] "Pearson correlation coefficient," https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[23] Google, "Google AdMob Mediation Platform Introduction." https://developers.google.com/admob/android/mediation.

[24] Jasmine Cohen, "What is Mobile Ad Mediation and How Does it Help App Developers?" https://www.ironsrc.com/news/what-is-mobile-ad-mediation/, November 2017.

[25] M. Kühnel, M. Smieschek, and U. Meyer, "Fast identification of obfuscation and mobile advertising in mobile malware," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1. IEEE, 2015, pp. 214–221.

[26] R. Shao, V. Rastogi, Y. Chen, X. Pan, G. Guo, S. Zou, and R. Riley, "Understanding in-app ads and detecting hidden attacks through the mobile app-web interface," *IEEE Transactions on Mobile Computing*, vol. 17, no. 11, pp. 2675–2688, 2018.

[27] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 209–220.

[28] Appsee, "15 Mobile KPIs Every Developer Needs to Measure Right Now," https://android.jlelse.eu/15-mobile-kpis-every-developer-needs-to-measure-right-now-b1cf1cc8b707, November 2016.

[29] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[30] Android Developers, "Android Intents," https://developer.android.com/reference/android/content/Intent.html/, March 2018.

[31] Tanzirul Azim, "A3E index page," http://spruce.cs.ucr.edu/a3e/index.html/, March 2018.

[32] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-apps ads." in *NDSS*, 2016.

[33] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 9–18.

[34] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 89–103.

[35] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "Impact of ad libraries on ratings of android mobile apps," *IEEE Software*, vol. 31, no. 6, pp. 86–92, 2014.

[36] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, "Why people hate your app: Making sense of user feedback in a mobile

app store," in *Proc. of the 19th SIGKDD*. ACM, 2013, pp. 1276–1284.

[37] B. Stone-Gross, R. Stevens, A. Zarras, R. Kemmerer, C. Kruegel, and G. Vigna, "Understanding fraudulent activities in online ad exchanges," in *Proc. of the Internet Measurement Conference(IMC)*. ACM, 2011.

[38] H. Xu, D. Liu, A. Koehl, H. Wang, and A. Stavrou, "Click fraud detection on the advertiser side," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 419–438.

[39] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, "Knowing your enemy: understanding and detecting malicious web advertising," in *Proc. of the 2012 ACM CCS*. ACM, 2012, pp. 674–686.

[40] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The dark alleys of madison avenue: Understanding malicious advertisements," in *Proc. of the Internet Measurement Conference(IMC)*. ACM, 2014, pp. 373–380.

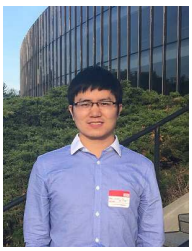
[41] P. Gill, V. Erramilli, A. Chaintreau, B. Krishnamurthy, K. Papagiannaki, and P. Rodriguez, "Follow the money: understanding economics of online aggregation and advertising," in *Proc. of the Internet Measurement Conference(IMC)*. ACM, 2013, pp. 141–148.

[42] I. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan, "On ad library updates in android apps," *IEEE Software*, 2017.

Ling Jin received her B.Eng. degree in computer science and technology from Xidian University, Xi'an, Shaanxi, China, in 2016. She is currently pursuing the Ph.D. degree with the college of computer science and technology, Zhejiang University, Hangzhou, Zhejiang, China. Her research interests include mobile security, blockchain security and natural language processing.

Boyuan He is a postdoctoral research associate at Department of Electrical Engineering and Computer Science, Northwestern University. He earned his a Ph.D. in computer science from Zhejiang University. His research interests lie in cybersecurity with the special focus on: logic vulnerability detection, Android app security, blockchain security, IoT device security, malware detection and forensic analysis.

Guangyao Weng received his B.Eng. degree in software engineering from Northeastern University, Shenyang, China, in 2015. He received his Master degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2018. He currently serves as a software engineer at Netease. His research interests include Android application analysis and Android application security.



Haitao Xu is an Assistant Professor in the School of Mathematical and Natural Sciences at Arizona State University, AZ, USA. His research focuses on the intersection of Cyber Security, Privacy, and Data Analytics. He received his Ph.D. degree in Computer Science from the College of William and Mary, VA, USA in December 2015.

Yan Chen received the Ph.D. degree in computer science from the University of California at Berkeley, USA, in 2003. He is currently a professor with the Department of Electrical Engineering and Computer Science, Northwestern University, USA and a distinguished professor with the College of Computer Science and Technology, Zhejiang University, China. Based on Google Scholar, his papers have been cited over 10,000 times and his h-index is 49. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. He received the Department of Energy Early CAREER Award in 2005, the Department of Defense Young Investigator Award in 2007, the Best Paper nomination in ACM SIGCOMM 2010, and the Most Influential Paper Award in ASPLOS 2018.

Guanyu Guo received his B.Eng. degree in computer science from Chongqing University, Chongqing, China, in 2014. He received his Master degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2017. He currently serves as a software engineer at Tencent. His research interests include Android application analysis and mobile security.