

Article

Using Ad-Related Network Behavior to Distinguish Ad Libraries

Ming-Yang Su *, Hong-Siou Wei, Xin-Yu Chen, Po-Wei Lin and Ding-You Qiu

Department of Computer Science and Information Engineering, Ming Chuan University, Taoyuan 333, Taiwan; kanzaki0421aria@gmail.com (H.-S.W.); best01010193@gmail.com (X.-Y.C.); vongolax343@gmail.com (P.-W.L.); eggegg234567@gmail.com (D.-Y.Q.)

* Correspondence: minysu@mail.mcu.edu.tw

Received: 24 August 2018; Accepted: 1 October 2018; Published: 9 October 2018



Abstract: Mobile app ads pose a far greater security threat to users than adverts on computer browsers. This is because app developers must embed a Software Development Kit (SDK), called an ad library or ad lib for short, provided by ad networks (i.e., ad companies) into their app program, and then merge and compile it into an Android Package (APK) execution file. The ad lib thus becomes a part of the entire app, and shares the whole permissions granted to the app. Unfortunately, this also resulted in many security issues, such as ad libs abusing the permissions to collect and leak private data, ad servers redirecting ad requests to download malicious JavaScript from unknown servers to execute it in the background of the mobile operating system without the user's consent. The more well-known an embedded ad lib, the safer the app may be, and vice versa. Importantly, while decompiling an APK to inspect its source code may not identify the ad lib(s), executing the app on a simulator can reveal the network behavior of the embedded ad lib(s). Ad libs exhibit different behavior patterns when communicating with ad servers. This study uses a dynamic analysis method to inspect an executing app, and plots the ad lib behavior patterns related to the advertisement into a graph. It is then determined whether or not the ad lib is from a trusted ad network using comparisons of graph similarities.

Keywords: mobile ads; software development kit (SDK), android package (APK), ad lib; ad libraries; ad networks; graph; graph similarity

1. Introduction

According to a report released by the Interactive Advertising Bureau (IAB) of the United States on 26 April 2017 [1], the trend of digital advertising has transferred from personal computers to mobile devices. In 2016, the annual revenue of digital advertising in the United States was USD 72.5 billion, of which revenue from mobile ads exceeded 50% for the first time, reaching USD 36.6 billion. The Mobile Application Industry Report for 2015 [2] revealed more about the popularity and importance of mobile advertising: 82% of app developers made a profit by advertising, and 91% were still using banner ads. Obviously, consumers were not willing to pay to download apps, so app developers turned to free apps and used ads to make profit.

The report on malicious mobile software evolution released by Kaspersky in February 2017 [3] listed 8,526,221 detected malicious apps in 2016, which was three times as many as that in 2015. Increasingly more information security reports related to mobile ads have since been conducted. A report by Trend Micro in June 2017 [4] showed that a Trojan Android ad program called Xavier could steal users' personal information and transmit it to somewhere without user permission whenever users downloaded the embedded app. According to the Trend Micro data, over 800 Google Play Android apps contained the Trojan ad lib, which had been downloaded millions of times. These apps included

utility apps such as photo editing apps, desktop and ringtone change apps. Xavier had a self-protection mechanism to avoid detection, and also downloaded and executed other malicious codes.

Doctor Web, an anti-malware company, indicated that in June 2016 a Trojan called Android.Spy.305, had been embedded in 155 Google Play apps, and estimated that more than 2.8 million people had downloaded and installed them [5]. The new Trojan, Android.Spy.305.origin, originally put into an ad lib, was embedded in apps when some developers used this ad lib to generate advertising revenue. It was known that 155 kinds of apps made by 8 app development companies had been infected. Once mobile device users had installed the embedded Android.Spy.305.origin module with the ad lib, it then connected to a Command and Control server to download the additional Android.Spy.306.origin module. The additional module would then begin to steal personal data, including, Google account E-mail logins and passwords, installed app lists, system languages, mobile brands, device names, IMEI numbers, OS versions, screen resolution, telecom operators, etc. In addition, third party apps would be installed during the app installation, which would then display various malicious advertisements from time to time. Many researchers have noticed the security issues caused by ad libs and a lot of efforts have been made in recent years to address this. Some of them are introduced below.

Athanasopoulos et al. [6] estimated that more than half of the apps available on Google Play contained ad libs linked to third party advertisers, posing a significant security risk to mobile app users. They therefore proposed the Native Code Isolation for Android Applications (NaCIDroid) architecture to separate the program code of an ad lib from that of an app, thus preventing permission sharing. Kumar et al. [7] noted that many ad libs required too many privileges or used privileges for which they did not have authority. Some observed apps could also sniff network traffic to obtain package content across the ad requests of multiple ad networks, making a user's personal information more easily accessible. They also discussed how a few notorious ad libs used online third parties to stealthily transfer personal information to an unknown server. Gao et al. [8] noted that because ad libs and apps were compiled after their merging, it was impossible to prevent the ad lib from using unauthorized permissions that exceeded the ad lib instructions. They therefore designed the Permission Supervision for Android Advertising (PmDroid) system to block ad libs' unauthorized use of permissions to transfer information. PmDroid employed a graphical interface to present the seriousness of any unauthorized usage. To understand the actual actions of these SDKs, Gao et al. wrote 53 different apps, each with a different ad lib embedded. The apps did not do anything, but announced all the privileges of the Android system to which they had access. The packet traffic of the apps was then recorded in order to understand how the ad libs abused permissions. Because the apps themselves did nothing, all network traffic was the result of ad lib activity. The authors concluded that unauthorized use of permissions by ad libs was very serious.

Narayanan et al. [9] observed that it was difficult to judge ad lib behavior using only the ad lib program code due to the widespread use of modern obfuscation tools. They used 26 different ads in their experimental dataset in order to test such obfuscation tools. They then proposed the AdDetect framework to assist in detecting ad libs and their behaviors in apps. AdDetect used semantic analysis to check ad libs, and used a support vector machine (SVM) to make classification judgments. Liu et al. [10] proposed their system, called PEDAL, to de-escalate privileges for ad libs in mobile apps. The study reported that, even if ad libs used obfuscation tools, PEDAL had a 98% accuracy in detecting them. Yan et al. [11] designed a new Android model, RTDroid, which basically modified the internal components of the Android operating system, and made use of a real-time Virtual Machine (VM) instead of the original Android Dalvik VM. This ensured that the execution of any app and its ad lib had greater predictability.

Book and Wallach [12] noted that, while ad libs could use the privileges of the host app to secretly transmit data, the host app could also use the privileges of the ad lib to engage in extra, unauthorized actions. That is, app developers and ad networks were colluding to carry out aggressive activities. The authors collected 114,000 apps, and collected statistics for the 20 most frequently used advertisers, identifying a total of 64,000 apps using those 20 ad libs. By observing the behavior of the 64,000 apps,

they concluded that app developers often actively collected too much personal user information to supply to ad networks in pursuit of high advertising profits. In addition, they found that the greater the popularity of an app, the easier it was to engage in such behavior, since as the number of users of an app increased, so did the motivation for advertisers to engage in such profit-seeking actions. Ruiz et al. [13] discussed the problems caused by ad lib updates. According to their experiment data, over 90% of apps were free, and advertising was the only income for these app developers, so it was very important to ensure that ad libs embedded in apps could bring the expected profit. If ad libs didn't achieve the expected profit, they were replaced or updated. The authors collected 13,983 versions of 5937 apps, and found that nearly 50% of these apps had changed their ad libs within 12 months by increase, removal or update. Ad lib maintenance was thus a burden on app developers. Su et al. [14] developed a data exploration method for HTTP dataflow. The features adopted were quantitative, timing and semantic. The authors claimed that their traffic identification of malicious ad libs could achieve an accuracy of 95% in their experiments. Kuzuno and Magata [15] used the difference of HTTP online traffic to identify ad libs. They adopted 1000 known advertising pictures to identify others. The experiment results exhibited a 76% detection rate for known advertisement maps and 96% for manual sorting advertisement maps.

Kajiwara et al. [16] observed that ad libs periodically used ad request packets to transfer personal information to ad servers, and received ad reply packets from ad servers. These reply packets were mainly advertisement pictures which appeared on the apps, changing the window screen. It was thus possible to estimate whether an app had an embedded ad lib by mathematically processing the HTTP frequencies online and screen changes. Crussell et al. [17] focused on the issue of MAdFraud, wherein app developers used background processing to connect to ad servers and ask for advertisements for profit, without users' knowledge, or have the program automatically click ads, thereby deceiving the ad networks. The PrivacyGuard system proposed by Song and Hengartner [18] had a number of functions which could not only track the flow of sensitive information, but also handled sensitive information to protect against illegal access. Backes et al. [19] noted the trend of ads being embedded in free apps, but those released apps often using an old version of an ad lib, thus hiding security weaknesses. The authors designed a system to help users check whether the ad libs contained in downloaded apps had security concerns, including whether there were malicious behavior instructions for obfuscation. Lee et al. [20] proposed the use of Contextual and Semantic perspectives to distinguish between app behavior and ad behavior. Tang et al. [21] carried out a static analysis of 10,710 apps, and found that 76.08% of them had obvious unauthorized use problems, and of those, 424 apps' sensitive permissions were only used by ad libs, instead of the host apps. This study also deals with the abuse of permissions by ad libs in a semantic way. Liu et al. [22] discussed the possibility that analytics libraries were more likely to leak users' personal information than ad libs. Analytics libraries are the mechanisms for tracking ad presentation and ad clicking on mobile phones.

Stevens et al. [23] evaluated 13 well-known ad networks and found that some ad libs had significant problems. For example, Mobclix used some permissions unrelated to displayed ads, such as Send SMS and Read Calendar. Through testing, this ad lib adopted 7 undeclared privileges, including four very aggressive permissions: Read Calendar, Write Calendar, Read Contacts and Write Contacts. In addition, 7 of 13 ad libs analyzed included JavaScript interface, which indicated that these modules could perform external JavaScript. The external malicious JavaScript could be embedded into four of seven ad libs (Mobclix, Greystripe, Mocean, InMobi) and executed; the behavior of which was as follows. Mobclix modified the user's calendar, contacts, message and image files, and opened or closed the camera. Greystripe obtained or set cookies, which could include account passwords or credit card numbers. Mocean sent newsletters and e-mails, made calls, added calendar items, and obtained users' locations and any network requests. InMobi sent short messages (SMS) and e-mails, made phone calls and modified users' calendars.

Today, what users most want to know in this context is what ad libs are embedded in their downloaded apps. If an ad lib is well known, it may be relatively safe, and vice versa. This study runs

apps on an emulator, and analyzes their network behaviors related to advertising. Most ad libs exhibit different behavior patterns, which are plotted into graphs to determine whether an ad lib comes from a trusted advertising company, using similarities between the graphs. The remainder of this paper is arranged as follows. Section 2 describes the operation of ad libs and related knowledge. Section 3 describes the proposed method of graph drawing according to an ad lib's network behavior patterns. Section 4 presents experimental results, and Section 5 offers conclusions.

2. The Operation of Ad Libs and Their Security Issues

Since 2010, mobile networks have undergone rapid growth and development. The boom included the adoption of mobile devices to quickly and reliably send messages. This sudden ubiquity of mobile devices resulted in a new mobile advertising market worth thousands of millions of U.S. dollars each year. The revenue of Internet advertising took 23 years to catch up with that of TV advertising, but the income of mobile advertising took only 6 years to surpass that of computer ads in 2016 [24]. However, as shown in a Purdue University and Microsoft report [25], the cost of using these free apps, which depended on ads for income, was the power consumption of the mobile phone and leakage of users' personal information. The surprise finding was that up to 75% of mobile device electrical power was used for advertising services, or tracking and uploading the relevant information of the user. There were already a variety of proper solutions for the advertising problems caused by websites, which could be addressed by computers via browsers. However, the information security problems caused by the mobile app ads had not yet been completely solved.

The reason these mobile advertising security-related problems were so difficult to work out was that ad libs were embedded in host apps and compiled together into APK execution files. That is, the ad lib had become a part of the entire app and could use all the permissions granted to the app. For example, an ad lib could claim to only use permissions P1 and P2, while the host app claims permissions P3, P4 and P5. Once merged and compiled, the ad lib would be able to use all permissions, i.e., P1, P2, P3, P4 and P5. When installing this app, the Android system only informs the user that it will use P1, P2, P3, P4 and P5, and once the app is installed, the system does not distinguish between the host app and ad lib permission use, and does not prevent the ad lib from using all privileges belonging to permissions P3, P4 and P5.

Ad libs are SDKs (Software Development Kits). In order for ad networks to target users with appropriate ads, the embedded ad libs collect user information such as age, gender, income, etc. However, a number of ad libs exceed their permissions, and collect the sensitive user data, web use habits or track a user's position, etc. The ad libs connect to an ad server autonomously and employed ad request messages to send out the collected personal information without the awareness of the user. Meanwhile, the ad reply messages from the ad server are received, which may include JavaScript. Thus, ad libs carry out unsanctioned, malicious actions such as collecting contact information, audio and image files, sending messages and emails, or stealing mobile users' cookies to obtain user accounts and passwords. The ad replies may also require the app to connect to a third-party server to download malware and automatically install and execute it in the background. This kind of app itself does not contain malicious codes, and all security risks occur in the advertising behavior during execution, so that general anti-virus software is unable to effectively detect them. Figure 1 shows the flowchart of app advertising processing.

The AppBrain website [26] listed the top five hundred ad networks around the world, of which the Google's AdMob was the most popular. According to the latest information released by this website in December 2017, 61.52% of all apps installed had the AdMob ad lib embedded. The second most popular network was Unity, with 18.73% of all apps installed having this ad lib embedded. Third was Chartboost, with 14.00% of installed apps using this ad lib. The less well-known ad networks may offer higher advertising profits to attract app developers, but their security risk is higher. Ad networks provided documentation on their official websites, but some collect more personal information than their permissions allow, and app developers are not aware of it, or do not mind, because of their

desire for profit. This is because, if an ad lib cannot collect enough personal information to include in ad request messages, ad servers determine that it is unable to provide effective advertising images to potential customers, and will thus not reply to the ad request messages. This means that app developers lose financially on the app, as free app developers rely on ads being sent to the users' phones for income.

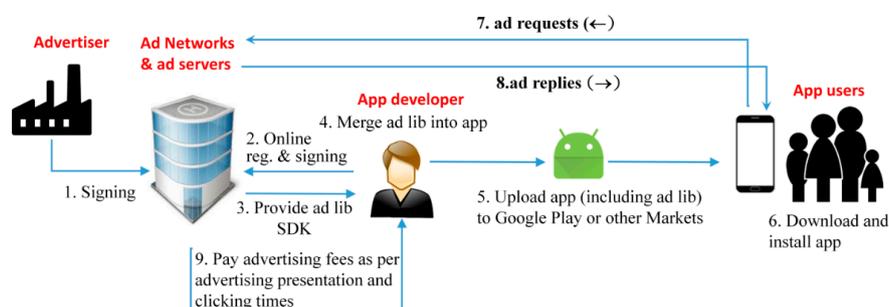


Figure 1. Ad lib operation flowchart.

According to Ruiz et al. [27], as a result of a large increase in free apps, the reply rate to ad requests of the Top 40 ad networks is lower than 18%. Therefore, free app developers must turn to the less known ad networks and (or) embed multiple ad libs from different companies in an app at the same time so as to increase the possibility of getting advertising pictures. The authors also collected more than 625,000 apps on these ad networks. After analysis, it was found that 34.88% of those apps had two or more ad libs embedded. A small number of apps had as many as 28 ad libs embedded.

Wei et al. [28] found that even apps with a good reputation and flagged as normal by anti-virus software were likely to be connected to malicious websites during their operation. They combined static (decompiling and checking program code) with dynamic (running apps for two hours and clicking as many links as possible through the tools) methods to observe who an app would communicate with. They collected 13,500 normal and popular apps, and found that in the course of their execution, these apps were connected to 254,022 URLs. In addition, 1,260 known malicious apps were collected, and it was found that they were connected to 19,510 URLs in their execution processes. According to the check returns based on Web-Of-Trust (WOT) [29] and VirusTotal [30], the authors divided all the above URLs into four categories: good websites, low-reputation websites, bad websites and malicious websites. Of the normal and popular apps, 8.8% of them were connected to malicious websites, 15% were connected to bad websites, and 73% to low-reputation websites. A total of 74% were connected to websites unsuitable for children. Of the known malicious apps, the situation can be expected to be worse. But otherwise, the authors found that the online URL distribution was similar to that of normal apps. This paper revealed an important point: even thorough and effective anti-virus software cannot guarantee that a certain app is safe, because the problem may not lie in the app itself, but with the website associated in the execution process. If connected to malicious or bad websites, a normal app could cause unimaginable damage.

The above authors [28] also found that only static decompilation of apps was not sufficient to achieve an effective full check by examining all possible online URLs, because the website could reconnect to other URLs through HTTP redirect mechanisms. Such problems are more difficult to predict because of the embedded ad libs of the apps. In fact, online advertising companies could resell ad slots to other ad networks (usually less known) through Ad exchange [31] so as to maximize advertising profits. This increases the advertising security risk, as the website could connect to multiple URLs when, for example, a free online game app is executed. Aside from the game server(s), the site could connect to ad server(s), redirect or unnamed server(s) by an ad resale mechanism.

3. The Proposed Method

Unlike static analysis, dynamic analysis focuses on the behavior of program execution, by analyzing the behavior of an app in an emulator. In some cases, better results may be obtained by dynamic analysis because it is resistant to obfuscation tools. Some researchers have emphasized the importance of dynamic analysis [32–36] for this reason. Since ad messages are carried out through HTTP packets, an understanding of HTTP is necessary to study the behavior pattern analysis of an ad lib, including the meaning of each field, and the information contained in it, so that the required data used in this research can be obtained.

In this study, an app was executed in an emulator and the packets of all network behavior were recorded, from which the packets related to the advertisement were picked using the proposed method. The tools used in this study were BlueStacks, TCP DUMP, ADB and self-created software. The emulator, called BlueStacks, used TCP DUMP to record network traffic from the virtual network adapter. The Android debug bridge (ADB) tool could directly access the Android emulator. The “logcat” instructions therein produced the required record files and the “pull” instructions exported the packet files (PCAP format) in the virtual machine. Because the captured packets were extremely large and messy, a program was designed to filter the packets related to the advertisement. Figure 2a shows a part of the proposed program, Figure 2b shows an ad request message, and Figure 2c shows an ad reply message.

```

int Reponse = 0;
int Request = 0;

for (; n<ip_len; n++)
{
//-----http get or post request-----
if (!find_http && ((n + 3 < ip_len && strcmp(ip_pkt_data + n, "GET", strlen("GET")) = 0)
|| (n + 4 < ip_len && strcmp(ip_pkt_data + n, "POST", strlen("POST")) = 0)))
    find_http = true, Request = 1;
//-----http get or post request-----
//-----http response-----
if (!find_http && n + 8 < ip_len && strcmp(ip_pkt_data + n, "HTTP/1.1", strlen("HTTP/1.1")) = 0)
    find_http = true, Reponse = 1;
//-----http response-----
//-----if Host is found-----
if (!find_host && n + 4 < ip_len && strcmp(ip_pkt_data + n, "Host", strlen("Host")) = 0)
{
    find_host = true;
}
//-----if Host is found-----
//-----find point-----
if (!find_point && n + 3 < ip_len && strcmp(ip_pkt_data + n, "png", strlen("PNG")) = 0)
    find_point = true;
}

```

(a)

```

MCU_Network_Security_Lab
AD_Analysis
No. 446 Frame      HTTP Get or Post Request
src address: 10.0.2.15/n  dest address: 54.192.212.146The Host is come from Host: 10.0.2.2:2861  10.0.2.15
10.0.2.2
or Host: d1e5xacotudrwg.cloudfront.net  10.0.2.15  54.192.212.146

GET /production/builds/2017-01/48793/c14d56f0-1e22-476c-b7f3-9e39f42a61f4/index.html HTTP/1.1
Host: d1e5xacotudrwg.cloudfront.net
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Linux; Android 4.4.2; SAMSUNG-SM-N900A Build/KOT49H) AppleWebKit/537.36
(KHTML, like Gecko) Version/4.0 Chrome/30.0.0.0 Safari/537.36
Accept-Encoding: gzip,deflate
Accept-Language: zh-TW,en-US;q=0.8
X-Requested-With: com.kiloo.subwaysurf
D:\Users\user\Desktop\Research\Data\20170827\F\subway.pcap

```

(b)

Figure 2. Cont.

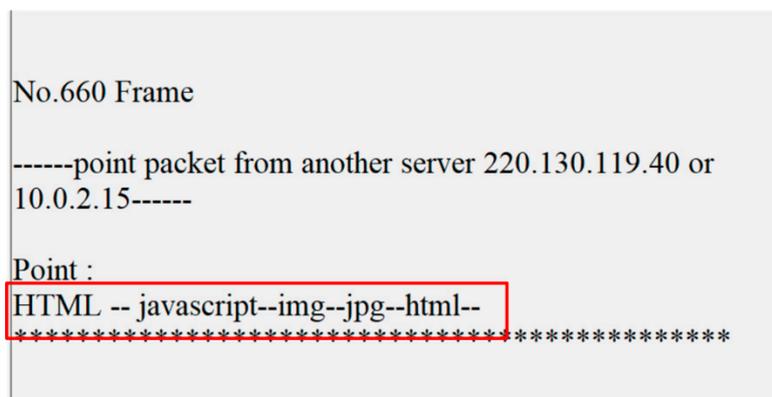


(c)

Figure 2. The proposed tool for obtaining advertising behavior: (a) Partial program; (b) Ad request; (c) Ad reply.

In this study, the interaction between ad lib and ad server was presented by a graph according to their HTTP connections. A series of contents of ad replies from the server were observed, which were driven by the ad request from the ad lib. There were basically 3 kinds of content types sent back to the app by the ad server: HTML, JavaScript, and IMG, of which IMG could have different picture formats, such as PNG, JPG and gif.

The proposed graph-based method first identified the main behaviors of the ad lib, each of which was expressed by one vertex in the graph. All vertices were connected according to the proposed algorithm, and then an undirected graph was constructed to represent the network behavior of the ad lib. The PChome [37] ad lib was taken as an example to illustrate as follows. Figure 3a shows the main behavior related to ads extracted from the packet traffic of the emulator by the proposed program. In order, HTML -> JavaScript (JS) -> IMG -> IMG -> html, where the upper-case HTML represented ad requests, and the lowercase html indicated ad replies. This array of ad behavior pattern [HTML, JavaScript (JS), IMG, IMG, html] was taken as input, and an undirected graph, as in Figure 3b, was constructed using the algorithms, as shown in Figure 4.



(a)

Figure 3. Cont.

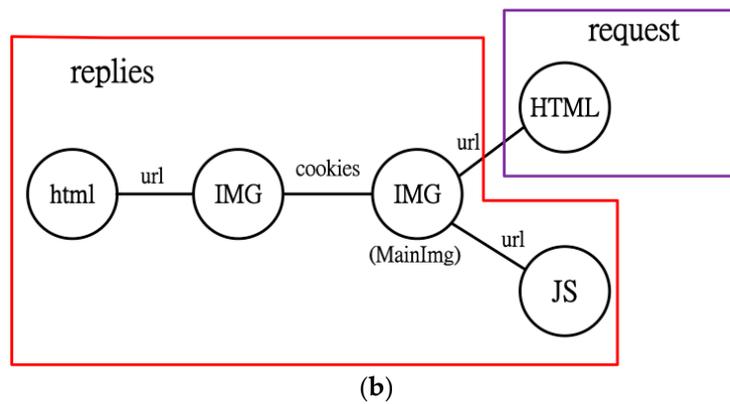


Figure 3. Graph construction of PChome ad lib: (a) online advertising behavior and (b) graph representation.

```

INPUT: An array of Actions //for example: [HTML, JavaScript (JS), IMG, IMG, html]
//types: HTML, IMG, Javascript(JS)
//HTML means coming from ad request, html means coming from ad replies

OUTPUT: a graph

Algorithm Graph_Construction()
{
  Scan the input array from left to right to locate the first IMG, also called MainImg.
  If there is no IMG, then exit;
  //supposedly at least an ad picture has been delivered in the ad activities
  else Create a vertex for the MainImg in the graph; //this is the first vertex created in the graph//

  for every element on the leftside of the MainImg in the input array
  {
    Create a vertex for the element and put the vertex on the right-hand of the MainImg in the graph;
    //the current element is either html(HTML) or JS//
    Link the current vertex to the MainImg and label the edge as "url";
  }

  for every element on the rightside of the MainImg in the input array
  {
    if it is IMG
    {
      Create a vertex for the IMG; //current vertex
      Put the current vertex on the left-hand of the MainImg in the graph;
      Link the current vertex to the MainImg and label the edge as "cookies";
      //cookies used to link two image vertices
    }
    else //either JS or html
    {
      Create a vertex for the element; //current vertex
      Find the closest IMG on the leftside in the input array; //related IMG. It could be MainImg or regular IMG
      Put the current vertex on the left-hand of the related IMG vertex in the graph;
      Link the current vertex to the related IMG vertex and label the edge as "url";
    }
  }
}

```

Figure 4. Algorithm of graph construction by ad lib network behavior.

According to the algorithm in Figure 4, the input array of actions is first checked from left and right so as to find the first IMG, which was taken as the first IMG point (vertex) in the graph, also called the MainImg in the algorithm. Each element of the input array on the left of the first IMG formed its

own vertex, which was drawn on the right side of the MainImg and linked to the MainImg by the edge marked “url”. Then the element on the right of the first IMG in the input array was processed. The element on the right of the first IMG was checked in the input array, and each IMG on the right formed its own vertex, which was drawn on the left side of the MainImg and linked to the MainImg by the edge marked “cookies”. If there was no IMG (either HTML or JS), one vertex was formed, which was connected to the vertex formed by the nearest IMG on the left side of the element in the input array with the edge marked “url”.

4. Experiment Results

This section gives more examples to demonstrate the effectiveness of the proposed approach. In Figure 5, on the left of each subgraph is the main ad behavior of an ad lib obtained from the network traffic of the emulator, and on the right side of each subgraph is the undirected graph based on the algorithm, as shown in Figure 4.

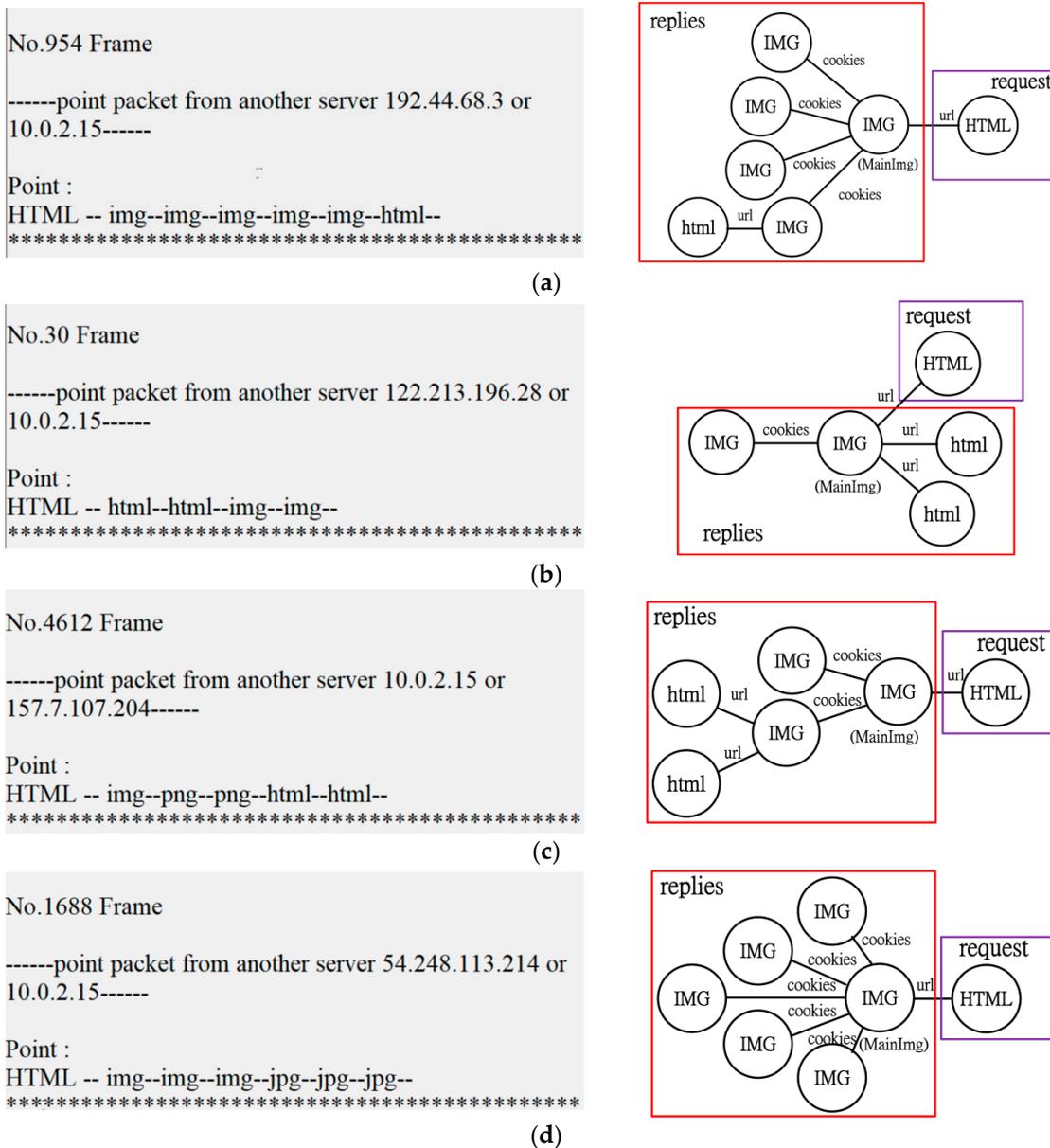


Figure 5. Cont.

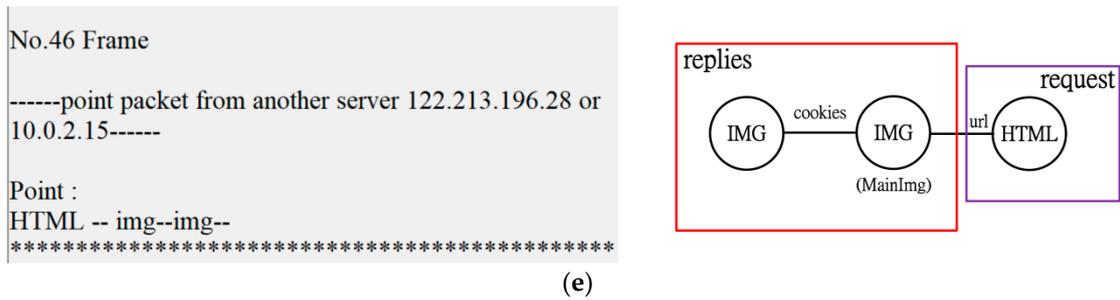


Figure 5. Ad libs’ network behaviors and their graph representations (a) MoPub, (b) Mmate, (c) Hap-game, (d) Millennial Media, (e) AppLovin.

Some of the advertising companies listed on the AppBrain website [26] and their ad libs were chosen and processed according to the above process, with the results presented in Table 1. For some ad libs, only one type of advertising graph was observed in the experiment, but most ad libs exhibited different advertising behavior patterns because of different advertising types, such as banner or full page, or version difference. Most of the javascripts found in this study were used to deal with the ad pictures, for instance adjusting the size of picture presented on the screen.

Table 1. Advertising behavior graphs of ad libs (with ad lib as index).

adecosystems	
ads.mopub.com	
app.mmate.jp	
AppLovin	

Table 1. Cont.

appserverstar		
campaign.ad-brix		
cdn.unityads.unity3d.com		
cloudfront.herocraft.com		
hap-game.sub.jp		
JScount		
hap-app.net		
magic.cmcm.com		
mydass		
rayjump		

Table 1. Cont.

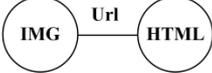
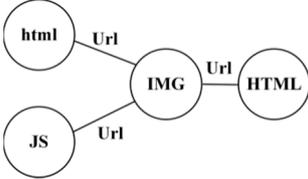
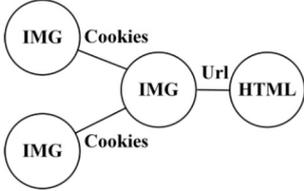
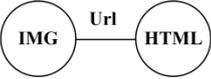
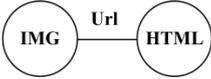
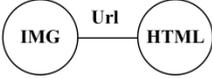
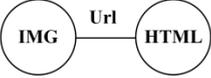
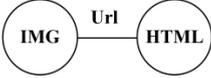
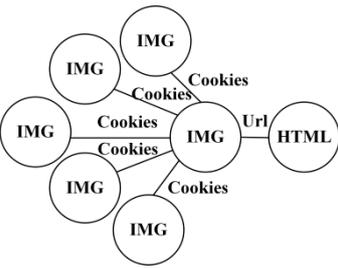
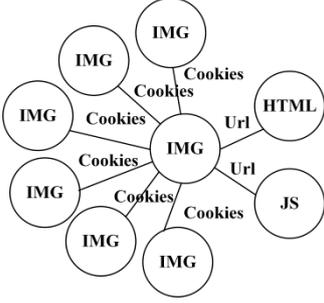
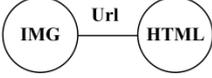
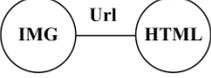
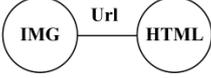
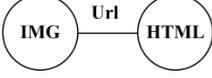
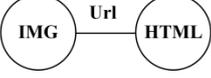
yahoo		
		
Startapp		
		
supersonicads-a.akamaihd.net	surpax	wpc.32DF9.rhocdn.net
		
ying	mysearch-online	omax.admarvel.com
		
Millennial Media	moreadexchange	AdMob
		
alog.umeng	app.hap.ne.jp	b.scorecardresearch.com
		
cdnicons.pluginmanagerconfig1.info	entity3.com	gamepromote

Table 1 uses ad lib as the index, while Table 2 uses the graph as the index. In Table 2, all graphs are categorized into different types, from A to P.

Table 2. Graph types and their corresponding ad libs (with graph as index).

A		<p>Adecosystems, ads.mopub.com, AppLovin, appservestar, campaign.ad-brix, cdn.unityads.unity3d.com, cloudfront.herocraft.com, hap-game.sub.jp, JScout, hap-app.net, magic.cmcm.com, mydass, rayjump, Startapp, surpax, wpc.32DF9.rhocdn.net, ying, mysearch-online, omax.admarvel.com, alog.umeng, app.hap.ne.jp, b.scorecardresearch.com, entity3.com, cdnicons.pluginmanagerconfig1.info,</p>
B		<p>Adecosystems, app.mmate.jp, AppLovin, AdMob, cdn.unityads.unity3d.com</p>
C		<p>ads.mopub.com</p>
D		<p>ads.mopub.com, yahoo, PChome</p>
E		<p>app.mmate.jp, appservestar, cdn.unityads.unity3d.com</p>
F		<p>Hap-game</p>
G		<p>campaign.ad-brix, rayjump, yahoo</p>
H		<p>cdn.unityads.unity3d.com, cloudfront.herocraft.com, gamepromote</p>
I		<p>mydass</p>
J		<p>mydass</p>

Table 2. Cont.

K		mydass
L		Startapp
M		supersonicads-a.akamaihd.net
N		Millennial Media
O		moreadexchange
P		app.mmate.jp

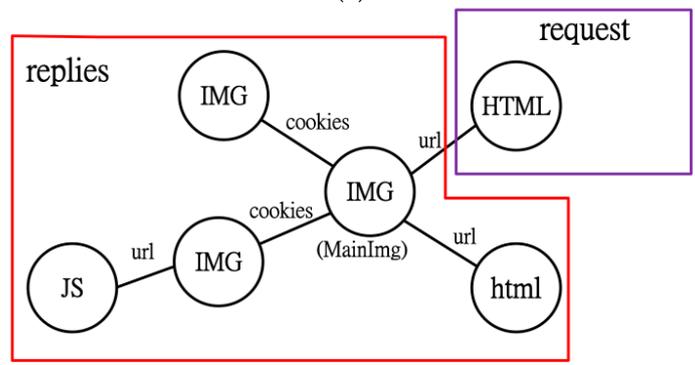
Because of the variety of graphs, a formula was designed to quantize the figures in order to make the graph classification easier later. This formula is mainly based on the vertices. The larger the IMG number, the lower its value; on the other hand, the larger the number of JavaScript and HTML, the lower the value. However, the differentia was not big. That is, the impact caused by JavaScript and HTML was relatively small compared to the number of IMG.

$$\frac{100}{2\#IMG-1} - 2\#JavaScript - 3\#HTML + 5$$

Using this formula, the corresponding values of different graphs could be obtained, and the corresponding values of some graphs are shown in Table 3. If an unknown pattern of ad lib produced the advertising behavior shown in Figure 6a, for example, i.e., the vertices were HTML, HTML, IMG, IMG, IMG and JavaScript, according to the algorithm in Figure 4, the graph was generated as Figure 6b. The value obtained by the suggested formula is 22, according to the numbers of different types of vertices. However, there was no matched graph value in Table 3, which indicated that it was a newly found advertising behavior model. Therefore, the content of the ad packet needed to be further analyzed, and it was found that this was the behavior of Mydas ad lib, shown in Figure 6c. Finally, the newly acquired information was added to Tables 1–3, in order to expand the content of known advertising patterns. The method proposed in this paper made it possible to more quickly classify the ad lib in an app. Some advertisers or app developers may deliberately hide the Host name. In this situation, the ad lib could still be classified by checking the ad behavior graph. If two or more ad libs shared the same graphs, the range of candidates was significantly reduced because of the classification.

```
No.468 Frame
-----point packet from another server 216.157.12.18 or
192.168.137.70-----
Point :
HTML -- html--img--img--img--javascript--
*****
```

(a)



(b)

```
No.468 Frame      HTTP Response
src address: 216.157.12.18/n dest address: 192.168.137.70The Host is come
from Host: androidsdk.ads.mp.mydas.mobi 192.168.137.70 216.157.12.18
or Host: androidsdk.ads.mp.mydas.mobi 192.168.137.70 216.157.12.18
```

(c)

Figure 6. An ad lib discovery, for example (a) network patterns (b) corresponding graph (c) packet details.

Table 3. Values of the different types of graphs.

Graph Type	#IMG	#html	#JS	Value
A	1	1	0	102.00
B	2	1	0	52.00
C	5	2	0	5.25
D	2	2	1	47.00
E	2	2	0	49.00
F	3	3	0	21.00
G	1	1	1	100.00
H	1	2	0	99.00
I	3	4	1	16.00
J	2	3	1	44.00
K	3	1	1	25.00
L	1	2	1	97.00
M	3	1	0	27.00
N	6	1	0	5.13
O	7	1	1	1.56
P	2	3	0	46

5. Conclusions

Mobile security issues today are not limited to malicious apps. This is because some security risks do not lie in an app itself, but in the advertising network behavior carried out in the course of the app execution. Well known advertising networks pose fewer security risks in this regard, but as their popularity increases and more instances of an app are installed, the number of ad requests sent to ad servers increases, resulting in fewer replies to ad requests by individual apps. App developers therefore often embed several ad libs into an app in order to increase advertising profits. Less well-known advertisers sometimes may offer better profits in order to attract app developers, and developers may increase profits at a higher risk to user security without users' knowledge. When advertisers hide their identity by removing their brand names or specific symbols, decompiling an app will still not identify which ad libs are embedded in the app. Therefore, the purpose of this study was to transform the network behaviors of ad libs into graphs, and then identify ad libs through the comparison of the graph similarities. Letting users know about the embedded ad(s) in an app is a further step to protect users' smartphones. If an ad lib cannot be identified, possibly from unknown ad networks, the user would be informed.

Author Contributions: M.-Y.S. conceived and organized the research work; H.-S.W., X.-Y.C., P.-W.L. and D.-Y.Q. conducted the experiments and analyzed data; M.-Y.S. wrote the paper; H.-S.W. and X.-Y.C. checked and verified the paper. All authors reviewed the paper.

Funding: This research was funded by the Ministry of Science and Technology, Taiwan, grant numbers MOST 106-2221-E-130-002 and MOST 107-2221-E-130-003. The APC was funded by the latter.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. IAB Internet Advertising Revenue Report—2016 Full Year Results. Available online: https://www.iab.com/wp-content/uploads/2016/04/IAB_Internet_Advertising_Revenue_Report_FY_2016.pdf (accessed on 7 October 2018).
2. Millennial Media: State of the Apps Industry Snapshot 2015. Available online: <http://visionmediainteractive.com/millennial-medias-state-of-the-apps-industry-snapshot-2015/> (accessed on 7 October 2018).
3. Mobile Malware Evolution 2016. Available online: <https://securelist.com/analysis/kaspersky-security-bulletin/77681/mobile-malware-evolution-2016/> (accessed on 7 October 2018).

4. Trend Micro Detects Xavier Android Malware in 800 Mobile Apps Downloaded from Google Play Store. Available online: <https://cio.economictimes.indiatimes.com/news/digital-security/trend-micro-detects-xavier-android-malware-in-800-mobile-apps-downloaded-from-google-play-store/59284441> (accessed on 7 October 2018).
5. Dr. WEB Anti-Virus, New Trojan Found in 155 Apps on Google Play: 2.8 Million Mobile Devices Already Infected. Available online: <https://news.drweb.com/show/?i=10115&lng=en> (accessed on 7 October 2018).
6. Athanasopoulos, E.; Kemerlis, V.P.; Portokalidis, G.; Keromytis, A.D. NaCIDroid: Native Code Isolation for Android Applications. *LNCS* **2016**, *9878*, 422–439.
7. Kumar, P.; Singh, M. Mobile Applications: Analyzing Private Data Leakage Using Third Party Connections. In Proceedings of the IEEE Proceedings of International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, 10–13 August 2015; pp. 57–62.
8. Gao, X.; Liu, D.; Wang, H.; Sun, K. PmDroid: Permission Supervision for Android Advertising. In Proceedings of the 34th Symposium on Reliable Distributed Systems, Montreal, QC, Canada, 28 September–1 October 2015; pp. 120–129.
9. Narayanan, A.; Chen, L.; Chan, C.-K. AdDetect: Automated Detection of Android Ad Libraries using Semantic Analysis. In Proceedings of the IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, 21–24 April 2014; pp. 1–6.
10. Liu, B.; Liuy, B.; Jin, H.; Govindan, R. Efficient Privilege De-Escalation for Ad Libraries in Mobile App. In Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, Florence, Italy, 18–22 May 2015; pp. 89–103.
11. Yan, Y.; Cosgrove, S.; Anand, V.; Kulkarni, A.; Konduri, S.H.; Ko, S.Y.; Ziarek, L. RTDroid: A Design for Real-Time Android. *IEEE Trans. Mob. Comput.* **2016**, *15*, 2564–2584. [[CrossRef](#)]
12. Book, T.; Wallach, D.S. A Case of Collusion: A Study of the Interface between Ad Libraries and their Apps. In Proceedings of the International Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Berlin, Germany, 8 November 2013; pp. 79–85.
13. Ruiz, I.J.M.; Nagappan, M.; Adams, B.; Berger, T.; Dienst, S.; Hassan, A.E. On Ad Library Updates in Android Apps. *IEEE Softw.* **2017**. [[CrossRef](#)]
14. Su, X.; Liu, X.; Lin, J.; He, S.; Fu, Z.; Li, W. De-cloaking Malicious Activities in Smartphones Using HTTP Flow Mining. *KSII Trans. Internet Inf. Syst.* **2017**, *11*, 3230–3253.
15. Kuzuno, H.; Magata, K. Detecting Advertisement Module Network Behavior with Graph Modeling. In Proceedings of the Ninth Asia Joint Conference on Information Security, Wuhan, China, 3–5 September 2014; pp. 1–10.
16. Kajiwara, N.; Kawamoto, J.; Matsumoto, S.; Hori, Y.; Sakurai, K. Detection of Android Ad Library Focusing on HTTP Connections and View Object Redraw Behaviors. In Proceedings of the IEEE International Conference on Information Networks (ICOIN), Cambodia, 12–14 January 2015; pp. 104–109.
17. Crussell, J.; Stevens, R.; Chen, H. MAdFraud: Investigating Ad Fraud in Android Applications. In Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services, Bretton Woods, NH, USA, 16–19 June 2014; pp. 123–134.
18. Song, Y.; Hengartner, U. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Denver, CO, USA, 12 October 2015; pp. 15–26.
19. Backes, M.; Bugiel, S.; Derr, E. Reliable Third-Party Library Detection in Android and its Security Applications. In Proceedings of the 23rd ACM Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 356–367.
20. Lee, J.-H.; Jun, S.-Y.; Park, S.-J.; Kim, K.-M.; Lee, S.-K. Demo: Mobile Contextual Advertising Platform based on Tiny Text Intelligence. In Proceedings of the 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys), Niagara Falls, NY, USA, 19–23 June 2017; p. 181.
21. Tang, J.; Li, R.; Han, H.; Zhang, H.; Gu, X. Detecting Permission Over-claim of Android Applications with Static and Semantic Analysis Approach. In Proceedings of the IEEE Trustcom/BigDataSE/ICCESS, Sydney, NSW, Australia, 1–4 August 2017.
22. Liu, X.; Zhu, S.; Wang, W.; Liu, J. *Alde: Privacy Risk Analysis of Analytics Libraries in the Android Ecosystem*; Springer: Cham, Switzerland, 2017; pp. 655–672.

23. Stevens, R.; Gibler, C.; Crussell, J.; Ericksonand, J.; Chen, H. Investigating User Privacy in Android Ad Libraries. In Proceedings of the IEEE Mobile Security Technologies (MoST), San Francisco, CA, USA, 24 May 2012.
24. Mobile Web Usage Overtakes Desktop for First Time. Available online: <https://www.telegraph.co.uk/technology/2016/11/01/mobile-web-usage-overtakes-desktop-for-first-time/> (accessed on 8 October 2018).
25. Without Ads, Android Apps Could Be More Than Twice as Power-Efficient. Available online: <http://www.theverge.com/2012/3/19/2884902/android-apps-battery-efficiency-study> (accessed on 7 October 2018).
26. AppBrain, Android Ad Networks. Available online: <http://www.appbrain.com/stats/libraries/ad> (accessed on 7 October 2018).
27. Ruiz, I.J.; Nagappan, M.; Adams, B.; Berger, T.; Dienst, S.; Hassan, A.E. Impact of Ad Libraries on Ratings of Android Mobile Apps. *IEEE Softw.* **2014**, *31*, 86–92. [CrossRef]
28. Wei, X.; Neamtiu, I.; Faloutsos, M. Whom Does Your Android App Talk To? In Proceedings of the IEEE Global Communications Conference (GLOBECOM), San Diego, CA, USA, 6–10 December 2015.
29. Web of Trust. December 2017. Available online: <http://www.mywot.com/> (accessed on 7 October 2018).
30. VirusTotal. December 2017. Available online: <https://www.virustotal.com/en/#url> (accessed on 7 October 2018).
31. Ad Exchange. Available online: https://en.wikipedia.org/wiki/Ad_exchange (accessed on 7 October 2018).
32. Enck, W.; Gilber, P.; Chun, B.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, BC, Canada, 4–6 October 2010.
33. Blasing, T.; Batyuk, L.; Schmidt, A.D.; Camtepe, S.A.; Albayrak, S. An Android application sandbox system for suspicious software detection. In Proceedings of the 5th International Conference on Malicious and Unwanted Software (Malware 2010), Nancy, France, 19–20 October 2010.
34. Liu, J.; Liu, J.; Li, H.; Zhu, H.; Ruan, N. Who Moved My Cheese: Towards Automatic and Fine-Grained Classification and Modeling Ad Network. In Proceedings of the Global Communications Conference (GLOBECOM), Washington, DC, USA, 4–8 December 2016.
35. Chan, J.; Keng, J.; Jiang, L.; Wee, T.K.; Balan, R.K. Graph-aided directed testing of Android applications for checking runtime privacy behaviours. In Proceedings of the 11th International Workshop on Automation of Software Test, Austin, TX, USA, 14–15 May 2016; pp. 57–63.
36. Biswas, S.; Haipeng, W.; Rashid, J. Android Permissions Management at App Installing. *Int. J. Secur. Appl.* **2016**, *10*, 223–2322. [CrossRef]
37. PChome. Available online: <http://show.pchome.com.tw/pfb/> (accessed on 7 October 2018).



© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).