De-cloaking Malicious Activities in Smartphones Using HTTP Flow Mining

Xin Su[1,2], Xuchong Liu[1,2], Jiuchuang Lin[3,*], Shiming He[4], Zhangjie Fu[5] and Wenjia Li[6]
[1] Hunan Provincial Key Laboratory of Network Investigational Technology, Hunan Police Academy, Changsha, Hunan, 410138, China
[2] Departmentl of Information Technology, Hunan Police Acadeny, Changsha, Hunan, 410138, China
[e-mail: suxin@hnu.edu.cn, 14117874@qq.com]
[3] Key Lab of Information Network Security, Ministry of Public Security, Shanghai, China
e-mail: linjiuchuan@star.org.cn
[4] School of Computer and Communication Engineering, Hunan Provincial Key Laboratory of Intelligent Processing of Big Data on Transportation, Changsha University of Science and Technology, Changsha, Hunan, 410114
e-mail:heshiming_hsm@163.com
[5]School of Computer and Software, Nanjing University of Information Science and Technology, Nanjing, Jiangsu, 210044
e-mail: fzj@nuist.edu.cn
[6]Department of Computer Sciences, New York Institute of Technology, New York, NY, USA
e-mail: wli20@nyit.edu
*Corresponding author: Jiuchuang Lin

Received November 26, 2016; revised February 27, 2017; accepted March 21, 2017; published June 30, 2017

Abstract

Android malware steals users' private information, and embedded unsafe advertisement (ad) libraries, which execute unsafe code causing damage to users. The majority of such traffic is HTTP and is mixed with other normal traffic, which makes the detection of malware and unsafe ad libraries a challenging problem. To address this problem, this work describes a novel HTTP traffic flow mining approach to detect and categorize Android malware and unsafe ad library. This work designed AndroCollector, which can automatically execute the Android application (app) and collect the network traffic traces. From these traces, this work extracts HTTP traffic features along three important dimensions: quantitative, timing, and semantic and use these features for characterizing malware and unsafe ad libraries. Based on these HTTP traffic features, this work describes a supervised classification scheme for detecting malware and unsafe ad libraries. In addition, to help network operators, this work describes a fine-grained categorization method by generating fingerprints from HTTP request methods for each malware family and unsafe ad libraries. This work evaluated the scheme using HTTP traffic traces collected from 10778 Android apps. The experimental results show that the scheme can detect malware with 97% accuracy and unsafe ad libraries with 95% accuracy when tested on the popular third-party Android markets.

Keywords: Automatic execution, HTTP traffic, Android malware, flow mining, classification

1. Introduction

Smartphone apps have become the quintessential means of accessing personalized computing services like email. The ease of installing new mobile apps or upgrading existing apps makes the Android platform a popular choice among users and developers alike. But this rapid deployment and availability of mobile apps has made them attractive targets for malware and commercial advertisements issued through unsafe ad (advertisement) libraries. Malware authors take advantage of the update mechanism of mobile apps to infect existing mobile apps with malicious code and compromise the security of the smartphone. A lesser, but serious, threat is the embedding of unsafe ad libraries by developers, which are used in stealthy activities like downloading unsafe code, tracking users' personal profile and so on.

Recent studies show that malware based on Android platform accounts for 85% of mobile malware [1]. The private data of the users, such as the contacts list, text messages, location and other user specific data are the primary target for the attackers. To evade anti-virus signature scanners, the attackers take advantage of the Android update mechanism to download malicious code and change the nature of an originally safe mobile app. A similar functionality is achieved by the use of untrusted thirdparty ad libraries, which are more focused on trying to activate premium services on the smartphone or leak user specific preferences and other commercially relevant information. Due to the dynamic nature of the malicious code it is difficult to detect these threats using binary code analysis and anti-virus signature scanners. Also, several ad libraries are usually safe and do not try to steal the user's data, but the nature of the ad library can be inferred only after observing the data it sends and receives from the ad servers. Certain ad libraries repeatedly pop-up ads causing annoyance and disrupting the tasks of the smartphone users. While the financial losses due to these factors is substantial, a bigger loss is incurred due to the compromise of privacy of individual users leading to serious security concerns beyond the digital domain. Therefore, there is an urgent need to address these threats considering their impact in terms of user smartphone experience, individual privacy and financial losses. This work addresses the problem of detecting the malware and unsafe ad libraries with a goal of understanding the proliferation of such malcontent within an enterprise network or an Android marketplace.

A number of research works have characterized malware and unsafe ad libraries based on code analysis [2], [3]. These approaches have proposed a range of static and dynamic analysis techniques on a large number of Android apps to detect malicious apps. However, the accuracy of these approaches gets adversely affected due to the code obfuscation techniques used by malware authors. There are other works focusing on network level analysis and detect malicious behavior. In [4], the authors describe an approach to manually execute the apps and analyze the resulting traffic. However, this approach is not scalable due to the large number of apps being released into the Android market on a daily basis. In [5], the authors analyze malware using DNS traffic traces from cellular provider. However, this approach fails if the malicious apps use hard-coded IP addresses or if the DNS traffic behavior is similar to the behavior exhibited by benign apps.

Existing approaches have not addressed the detection of malicious apps and unsafe ad libraries while considering the correlation between the application level semantics and corresponding traffic of the mobile apps and the unsafe ad libraries. Study [6] shows that the majority of the activities of the Android apps are performed by over HTTP. Based on past research analysis, most network traffic generated from Android app and ad libraries is HTTP

traffic, especially, over 80% malware use the HTTP-based web traffic to receive bot commands from their C&C servers [25].

Our proposed approach focuses on the analysis of the HTTP flow traffic generated by the malicious apps with the goal of providing the network operator a detailed view of the proliferation of malcontent within a network. First, to extract the network behavior of an app, this work performs automatic execution of the mobile app and capture the resulting HTTP traffic traces. The execution of the Android app is essential to examine any malicious run-time behavior, such as leak personal information and to observe its traffic patterns corresponding to these activities. Second, to capture the activity related behavior of the mobile app, this work categorizes the HTTP traffic of the mobile app in three different metric categories: quantitative, semantic and timing related. Based on these categories, this work extracts several features from the HTTP traffic traces and use these features as an indicator of the application level behavior of the mobile app. Third, this work characterizes ad traffic by extracting useful features from well known malicious ad libraries in the wild, regardless of whether the ad library is embedded inside a benign or a malicious app. Through these features, this work can detect the presence of malicious ad libraries, even if the ad library traffic is intermixed with benign app traffic. Finally, this work generates compact fingerprints for the malware to categorize them based on similarity. The fingerprints can be distributed to the network operators to allow them track the extent of malicious activity in their network.

First, this work needs to be able to automatically execute several apps to extract statistically significant features while ensuring that the automatic execution is bounded within realistic time limits. The essential requirement of the automatic app execution process is that it must explore the relevant portions of the mobile app code in a detailed and comprehensive manner while simulating a typical smartphone user interaction with the mobile app. Towards this, this work developed AndroCollector, which is an automated Android app execution and traffic collection tool to execute the Android app based on depth-first exploration algorithm and captures the corresponding network traffic.

Second, there is need to identify suitable features under each of three categories: quantitative, semantic and timing, as an appropriate choice will enable proper characterization. Towards this, this work monitors the HTTP flows of the apps and extract various statistical traffic metrics, such as packet sizes and so on, and HTTP specific metrics, such as length of response in GET/POST requests and so on. Also, to capture the request/response semantics of malicious content, this work considers the timing related features such as the interval between a request/response cycle and so on. Using supervised learning algorithms, this work uses the extracted features to build a classifier to distinguish between the network behavior of a benign and a malicious app. Furthermore, this work uses the HTTP headers to generate fingerprints for different malware and categorize them.

Our key contributions are as follows. (1) This work designed AndroCollector, an automated traffic collection tool to automatically execute Android apps to collect real network traffic traces from 10K+ Android apps. (2) This work identified several HTTP traffic flow features from collected network traces of Android apps and showed that these features are quite accurate in capturing the malicious behavior of the mobile apps and unsafe ad libraries. (3) This work generated fingerprints from the host name and the invariant part of HTTP request to categorize malware and unsafe ad libraries. (4) This work performed extensive evaluation on real-life malware and unsafe ad libraries, using multiple classifiers and showed that our method has a high detection accuracy of 98%.

Organization. Section 2 describes the related work of malicious apps and unsafe ad libraries analysis and detection. Section 3 discusses the design details of AndroCollector tool.

Section 4 characterizes Android app and ad libraries based on several HTTP traffic and flow features. Section 5 discusses categorize Android malware based on HTTP fingerprinting. Section 6 evaluates our methodology and describe a fine-grained categorization approach for understanding malware distribution patterns. Conclusion describes in Section 7.

2. Related Work

In [7], the authors describe a coarse-grained protocol level classification based on signatures of applications developed from statistical information, such as packet sizes and other flow information. Discoverer [8] focuses on generating application-level signatures from the network traces of the application under study. However, those techniques do not work well for the Android apps where a majority of traffic is carried within HTTP and the major distinguishing features are present in the URLs.

Network level analysis of malicious behavior offers a complementary means of characterizing and mitigating malware. For example, a popular method of preventing or limiting the spread of malware is the use of Internet blacklists. IP blacklists provide a list of known bad actors in the form of IP addresses, which network operators can subsequently block; however, the use of DNS to build malicious network infrastructures has grown due to its resilience against IP blacklisting as discussed in [9]. Consequently, a significant amount of work has focused on analyzing networks at the DNS level [10], [11]. This has led to the creation of systems that are able to detect malicious domains through the use of passive DNS monitoring and machine learning techniques [12], [13], [14], [15], [16], [17], [18]. However, any blacklisting scheme is known to suffer from scalability issues as the number of blacklisted domains is not bounded and increases on a daily basis.

There have been many efforts that try to understand smartphone usage behavior. Profiledroid [4] aims to build app profiles at multiple levels, including network, but their technique completely relies on users running apps to generate traffic and does not scale for a large number of apps. Drebin [3] is a recently proposed approach, which tries to explain the behavior of malicious apps using features extracted through static analysis. However, this approach does not consider the run-time network behavior of the Android app and cannot detect obfuscated app behavior, which is revealed only after an obfuscated app unfolds itself and starts executing.

There have been a large number of directed efforts on analysis ad libraries. In AdRisk [19], the authors proposed a statistical approach to analyze ad libraries. In AdCache [6], the authors characterized ad traffic, but this work just focuses on limiting the energy and network signaling overhead caused by ads. MobiAd [20] and PrivAd [21] suggest local profiling and ad serving in order to protect the user privacy, using a third party to help in the anonymization phase. This work addresses the limitations of existing work by presenting a comprehensive and systematic characterization of malware apps and unsafe ad libraries using HTTP traces. The following sections will discuss the prevalence of HTTP in Android app development phase.

3. Automated Android App Traffic Tracing

3.1 AndroCollector

To accurately model the behavior of the Android app based on its HTTP traces, the Android app must be made to execute the various network related activities and try to simulate the

behavior of manual user interaction with the app. Therefore, this work designs an automatic app execution tool while focusing on two key design considerations: first, to execute the Android app in a comprehensive manner and second, to simulate the user's behavior to operate the Android app. The architecture of AndroCollector is shown in **Fig. 1**, which is composed of three components: Automatic Execution of Apps module, Traffic parser and Storage modules.

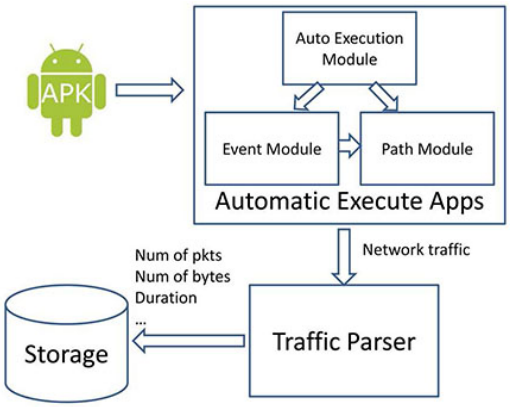


Fig. 1. Overview of AndroCollector

Automatic Execution of Apps This work designed this module base on two Android testing tools: monkeyrunner [22] and hierarchy viewer [23]. This component is responsible for executing Android apps and collecting their network traffic and consists of three sub-modules: Event module, Duration module and Auto Execute module.

Algorithm 1. Activity Exploration Algorithm

Input: Entry Point Activities |A|

```
1 function DFI(|A|)
2   for All Activities Ai, Aj in |A| do
3    Change to Activity Ai;
4    DEPTH_FIRST_IDENTIFICATION(Ai);
5   end for
6 end function
7 function DEPTH_FIRST_IDENTIFICATION(Ai)
8   Widgetset←GET_WIDGET(Ai);
9   for each widget in Widgetset do
10    if (Widgetset.isclickable==TURE) then
11     if(Ai→Aj)
12       record this activity transition;
13       DEPTH_FIRST_IDENTIFICATION(Aj);
14       Back to Activity Ai;
15     end if
16    end if
17  end for
18 end function
```

1) Event Module: The Event module implements the automatic install, execute and removal of Android apps based on the API of monkeyrunner. The module simulates user's behavior like clicking buttons, touching screen, pressing keyboard and so on, which are the set of user

interaction (UI) components, commonly known as widgets. These actions would trigger the app to access Internet and generate network traffic. To simulate user behavior events this work considers two types of user behavioral models: random and specific event. During the random behavior, the simulator chooses a random action, such as a button click or pressing key board to simulate a real user's operation on Android app and proceeds to explore all reachable activities from this action. On the other hand, the specific behavior event captures the scenario when the app needs to follow a specific execution path to trigger the events generating the network traffic. The set of such paths are implemented by using our Path Module as described in the following.

2) Path Module: The Path Module, using the API of hierarchy viewer, identifies the current activity and obtains feature information of all widgets in this activity e.g., (name, id, isClickable). Android apps consist of a number of activities where an activity is an app component that provides a screen with which users can interact to perform an action. When an app is launched, its Main Activity displays several clickable widgets, e.g., Button, Textview, Imageview and so on, which correspond to several options. Choosing one such widget allows the module to discover more possible widgets. The detailed exploration algorithm is described in Algorithm 1. Specifically, this work chooses those activities that lead to network traffic and store such paths in the Path Module.

3) Auto Execution Module: The Auto Execution Module is responsible for executing the app for traffic collection. This module executes apps automatically in two ways depending on the two kinds of behavioral events, random or specific, chosen from the from the Event Module. In the random behavior, the Auto Execution Module only receives random behavior events from the Event Module, and executes the app by choosing the reachable widgets in a random manner. For specific behavior, all the reachable paths are explored in depth and due to this exploration, specific behavior takes more time than random behavior. This module is also responsible for launching the traffic capture tool tcpdump, which automatically captures the network traffic flowing into and out of the Android app. The captured traffic traces are passed on to the Traffic Parser module for analysis.

Traffic Parser The Traffic Parser extracts traffic features (cf. Section 4) that correspond to different network behaviors of Android apps. The module considers individual flows to extract the traffic features. Since most Android apps run over HTTP protocol [6], [24], this module focuses on the feature extraction from HTTP traffic. The first class of features extracted from the collected traffic traces are based on flow statistics, which include features such as packet count, byte count, and so on. The second class is extracted from the client access patterns-based features, which shows the duration of apps visiting different servers, destination addresses and so on. The third class are HTTP specific features, which include different HTTP requests and responses, such as host name, request method, length of HTTP request/response and so on. These extracted features stored in a database and updated periodically by newer data.

Storage The storage component takes the results of traffic parser as input, stores them into a database and maintains periodical updates as observed from new data. The component also computes the mean, standard deviation, maximum, minimum, median, and sum, for further analysis. There is some meta-description maintained by the storage component about Android apps, such as package name, market place, whether contains ad library etc.

3.2 Ad Traffic Extraction

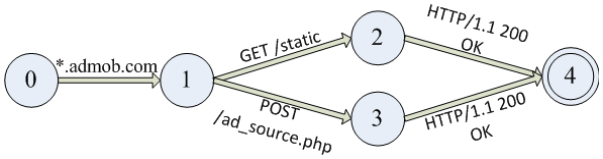
Since ad libraries are embedded in Android apps, their traffic is mixed with traffic of Android apps and makes it difficult to identify the specific nature of the ad traffic. Note that ad libraries

and the respective ad networks have high diversity [6] in terms of requesting resources related to displaying the ads or the ads themselves. This work utilized 100 most popular ad libraries as reference and analyzed the HTTP headers across these libraries. We observed that the combination of Host field, the invariant request and response patterns such as the URL query can be used to uniquely classify the ad libraries. This work was able to extract some distinguishing aspects based on the HTTP request mechanism employed by the ad-libraries. Some ad networks require HTTP POST method to update source information, e.g., InMobi and others require both GET and POST, e.g., Admob Some ad networks need to redirect HTTP request to another URL, e.g., googleadservices. Based on these three distinguishing characteristics, this work extracted a set of rules based on our observations and show them in **Table 1**.

Table 1. Rules extracted from example ad library

URL domain	HTTP request	HTTP response	Redirect URL
r.admob.com	POST /ad source.php	HTTP/1.1 200 OK (text)	
mm.admob.com	GET /static/xnetwork/arrow out.png	HTTP/1.1 200 OK (PNG)	
googleads.g.doubleclick.net	GET /aclk?sa=...&adurl= lawoethailand.net	HTTP/1.1 302 Found	lawoethailand .net
googleadservices.com	GET /paged/aclk?sa=...&adurl=airkx.com	HTTP/1.1 302 Found	airkx.com
umeng.com	POST /check config update	HTTP/1.1 200 OK (json)	

First, our approach checks whether the URL domain exists in the domain list of 100 ad libraries. Next, inspect the HTTP request and HTTP responses. If the HTTP response is HTTP/1.1 200 OK, extract traffic related to the URL domain. If the HTTP response is HTTP/1.1 302 Found, check the value of parameter of adurl to which the HTTP request redirects to and extract the traffic of the original URL domain and redirect URL domain and store them as one common trace for feature extraction. As an illustration, the library Admob sent a HTTP request to its servers, and received response HTTP/1.1 200 OK, which stands for successful request. The googleadservice library sent HTTP request which contains a adurl field showing the redirect URL, and received a response HTTP/1.1 302 Found, which means this ad network will connect to this URL to finish request subsequently. Based on these rules this work represents them as state machines as shown in **Fig. 2(a)** and **2(b)**, which captures the flow of ad network HTTP request and response. Since a state machine can represent a set of similar rules, thereby reducing the size of rules, and can make the ad traffic extraction more efficient.



(a) State Machine for Normal Ad

(b) State Machine for Redirect Ad

Fig. 2. Sample HTTP Traffic and Their Corresponding State Machines

4. HTTP Flow Mining Approach

4.1 Android Malware Characteristics

Existing works [6], [25] have noted the prevalence of HTTP usage in malware and ad libraries. This work provides a brief statistical overview to confirm this observation by studying a corpus of malicious and benign apps. This work collected a set of malware apps from Android Malware Genome Project [25], and also collected Android apps from Google Play and several popular third-party markets (the detail will be described in section 6.1). 94% of the apps requested the android.permission.INTERNET permission in their manifest files can be found from the collection of apps. Furthermore, in the malware data set, we noted that, 93% of samples use HTTP to receive bot commands from their C&C servers. Thus, this brief analysis shows the correlation between malware apps and HTTP. This is our motivation to study this correlation further and use this observation to detect malicious behavior in apps.

1) Android Apps: To understand the behavior of Android malware, this work has performed execution of Android apps infected with different malware and identified common behaviors across these apps. For this experiment, our test data set consisted 49 different malware families, where each malware family consists of malware binaries that are enhanced versions of each other without much change to the commonly observed malicious behavior. This work experimented with 1260 different malware binaries chosen uniformly from the malware families. Benign apps exhibit normal network behavior, such as online chat and watch videos, which are mostly user initiated without revealing any personal information to the external web server. Whereas, malware usually communicate with remote server to send user private information, e.g., IMEI, location, requesting dynamic loading of compiled Android code and converting the smartphone into bots. Furthermore, to avoid anti-virus signature detection, malware usually transfer a small amount of traffic data and end the communication in a short time-span.

Table 2. Malware activity

Malware name	Local activity	HTTP activity
Droidkungfu	Copy IMEI Number, Phone model number	HTTP Post to a hard-coded remote server – http://xxxxxx.xxxxxx.com:8511/search/sayhi.php
Anserverbot	Dynamic code loading and copying passwords, SMS etc	Using HTTP GET to retrieve encrypted commands from remote Bot master server
Plankton	Collects information, including the device ID, list of granted permissions to the infected app	HTTP Post to http://www.xxxxxx.com/ProtocolGW/installation
Basebridge	Collects sensitive information, such as IMEI, manufacture and model of the device	HTTP protocol:b3.8866.org on port 8080

Table 2 shows some typical activities exhibited by a few malware apps, which are categorized under: Local activity and HTTP Activity. The term Local Activity indicates the activity performed by the app within the phone and HTTP Activity indicates the specific HTTP calls used by the malicious app. Note that, the behavior of most existing malware is a composition of one or more of these activities as observed by the studies in [4]. Regardless of the type of malware, the use of HTTP to steal private information or receive control information can be seen to be the common feature. Furthermore, the network behavior of

metamorphic behavior remains unchanged, i.e., all malware binaries of a single malware family exhibit very similar or identical HTTP related behavior.

2) Ad Libraries: Since ad libraries are embedded inside Android apps, it is not immediately apparent to notice the behavior of a malicious ad library. Our test data consisted of 23 apps containing embedded malicious ad libraries. This work chose Android apps in such a way that their general behavior is well known and any extraneous network behavior must emanate from the embedded ad library. Therefore, in most cases, the nature of HTTP transmissions due to the ad libraries was a good indicator of the malicious behavior of the ad libraries. For instance, malicious ad libraries usually indulge in sending the phone call history information, browser bookmarks, installed apps list and other private information to the remote ad servers. **Table 3** shows a sample list of malicious ad libraries and their corresponding malicious activity. All the embedded ad libraries used HTTP transmissions to steal information or to compromise the smartphone user's privacy.

Table 3. Embedded ad library activity

Ad library	Malicious network activity
Soseco	Transmits call history information and uploads the list of installed apps
Enerysource	Downloads code that can open backdoors for future exploits
Adserver	Exposes location and IMEI information to its advertiser
Mobus	Reads SMS database looking for administrative information of the user's SMS Center and relays this information to its server

4.2 HTTP Flow Feature

To understand the relative differences between the HTTP traffic generated by benign and malicious apps, this work identified three different categories of HTTP flow features and compared them. The three flow categories are: quantitative, timing and semantics based features. Note that, these three categories can accurately capture the network level behavior of the Android app while being able to correlate this behavior to the execution of the app. As malware apps and ad libraries are expected to show variations in the network traffic volume and activity, these categories of features will enable us to observe such differences. Furthermore, since these feature categories are both statistical and semantic, they are content agnostic and are not dependent on the specific data set used to extract the features.

Our data set consisted of 1260 malware apps across 49 different malware families, as outlined in the previous section. 4868 benign apps have been chosen from a wide range of normal applications, such as games, books, finance, shopping, music and so on. For analyzing the ad library traffic we considered 4290 apps of which 3431 apps consisted of safe ad libraries and 23 apps consisted of malicious ad libraries using AndroCollector, executed all these apps individually and collected the corresponding flow characteristics. Since this work is interested in an aggregated view of the flow characteristics across these apps, this work uses the cumulative distribution function [25], CDF as the common metric of comparison across the various features. In all the figures, the X-axis represents the feature values, such as the number of packets and so on. The Y-axis represents the CDF of the corresponding value, which describes the probability that a feature value X with a given probability distribution will be found to have a value less than or equal to x_i where x_i is the x-axis co-ordinate.

$$F(x) = P(X < x_i) \tag{1}$$

1) Quantitative Features: This category measures and compares the volumes of traffic across malware and benign apps, and feature description is shown in **Table 4**. When malware communicate with malicious servers, they request update commands and leak private information with a fixed format. Also, malware do not generate large traffic volumes to avoid detection by anti-virus scanners or intrusion detection systems. Therefore, a malware trace might contain many flows with similar traffic size.

Table 4. Description of quantitative features

Feature name	Description
Number of packets	Number of packets transmit between app and server
Number of bytes	Number of bytes transmit between app and server
Number of received packets	Number of packets received by app
Average bytes of received packets	Average bytes of packets received by app
Average size of packets	Average size of packets transmit between app and server
In/out ratio	Ration of traffic size between sent and received of app

The first two features are numeric, i.e., the number of packets and the number of bytes sent within the execution time of the Android app. Benign apps have rich functionality, their network activities include, text chat, videos, image downloading and so on. Therefore, these network activities are expected to have a variable number of packets due to the variable size of the data involved. On the contrary, malware focuses on sending out private data out, which is usually in standard size regardless of the smartphone in use and hence, it is expected that the number of packets per flow is similar across multiple malicious apps. In **Fig. 3**(a) and **3**(c), 80% of malware flows contain about 10 packets or less has been observed. Only 30% and 50% of benign app and ad library flows achieve this number. Moreover, benign app and ad library transferred more packets per flow than malware.

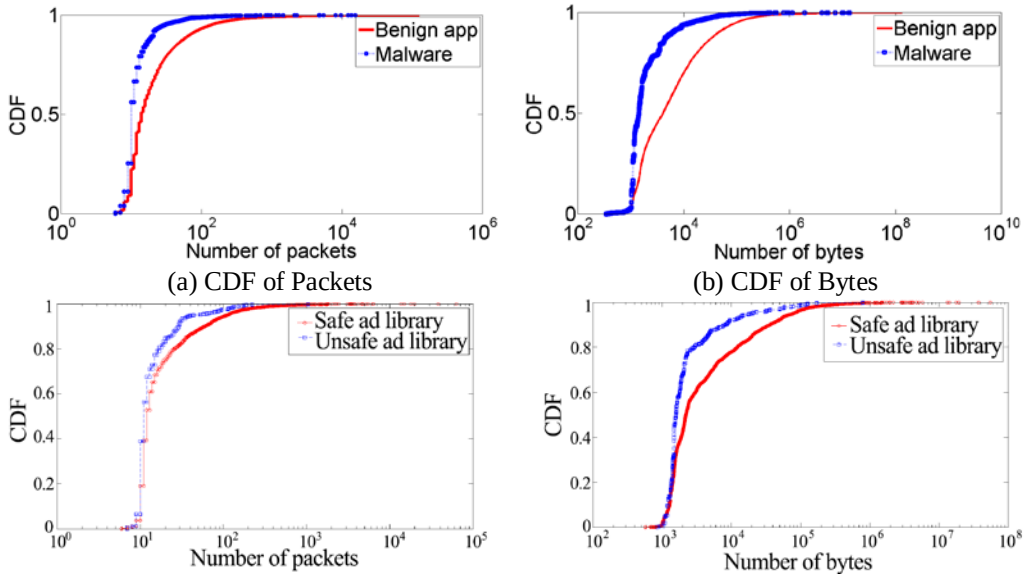
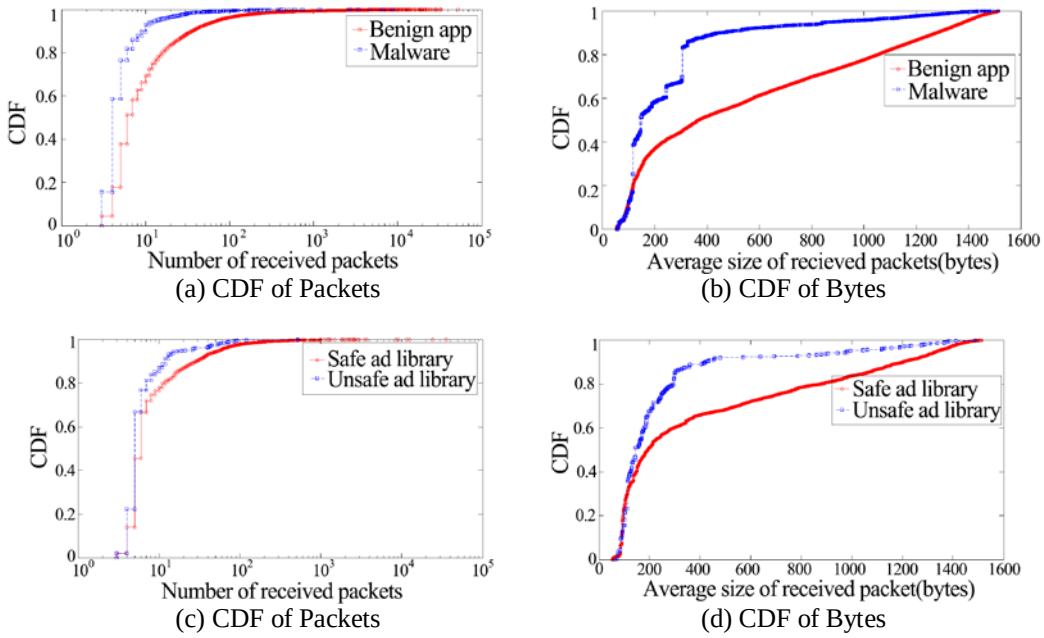


Fig. 3. Comparison of the Numeric Aggregates of Transfer Size

Next, the size of the incoming traffic in terms of the number of packets and the received bytes have been measured. These features represent the number of packets received during app communication with a remote server. Because benign apps may receive large size files from servers, due to the limitation of a packet length, the large size file is split into several segments thereby, increasing the number of received packets. Whereas in malware apps, the received command packets usually are small size packets and hence, remain undivided as shown in **Fig. 4**(a) and **4**(c). In these figures, benign app flows contain more received packets per flow than malware. Also, **Fig. 4**(b) and **4**(d) show the average packet size in each flow. The relative difference in packet sizes is clear from these results.

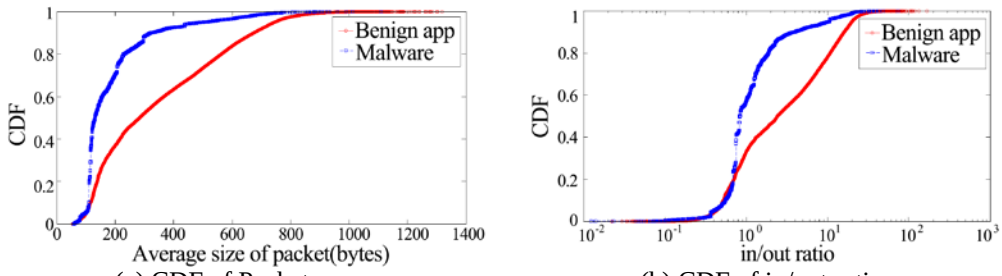


(a) CDF of Packets (b) CDF of Bytes

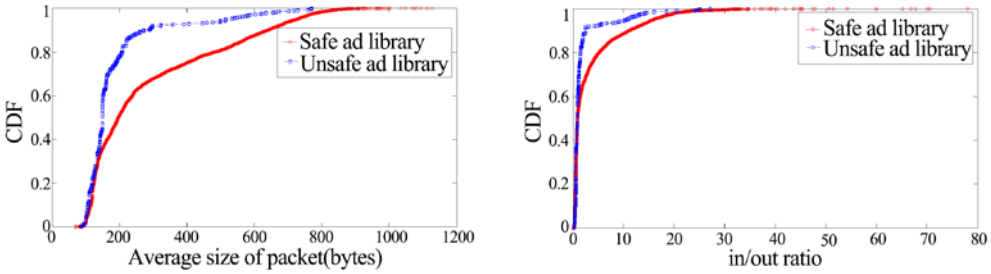
(c) CDF of Packets (d) CDF of Bytes

Fig. 4. Comparison of the Numeric Aggregates of Inward HTTP Flow Traffic

Finally, the CDF of the average data per packet as this gives an estimate of the uniformity of data leaked by malware apps or unsafe ad libraries has been compared. In benign apps, the packet size is not constrained as the user can download or upload data of any size. For malware apps the command packets typically have small size due to compact nature of the Botnet protocol communication.



(a) CDF of Packets (b) CDF of in/out ratio



(c) CDF of Packets (d) CDF of in/out ratio

Fig. 5. Data Sizes and Relative Traffic Ratios of HTTP Flows

In **Fig. 5**(a) and **5**(c), almost 50% of malware packet sizes are in the range from 101 bytes to 200 byte, and less than 30% of benign app packets sizes fall in this range. This work also measured the corresponding ratio of the traffic size going in and out of the smartphone as this feature captures the role of an app as a consumer or producer of data. The role of benign apps and safe ad libraries should be consumer, because they usually request large size files to indicate downloads. Whereas, malware apps and unsafe ad libraries should be in the role of a producer, because they usually send private data out, such as IMEI number, location, contact information and only receive small commands in response. Some malicious apps receive malicious code at run-time, however, this behavior typically happens in bursts and is short-lived. Moreover, once the code is downloaded, the malware reverts to its regular behavior of leaking more information out of the smartphone than downloading data from the Internet. From **Fig. 5**(b), for benign apps, note that about 20% of the ratios of the incoming and outgoing traffic are lesser than 1, which shows that this traffic contains more sent data than received data. As expected the HTTP traffic from Youku, PPTV, and PPS, which includes audio and video content, is mostly incoming traffic as the large values of the ratios show. Finally, note that 60% of the ratios in malware app are lesser than one, as malware traffic has unusually large uploads, which points to massive theft of sensitive user data.

A contrasting behavior is exhibited by ad libraries as can be seen from **Fig. 5**(d) where 50% of the ratios in ad library traffic are lesser than one. This is because incoming traffic of many ad libraries contain only contain text or Javascript code, which is typically small.

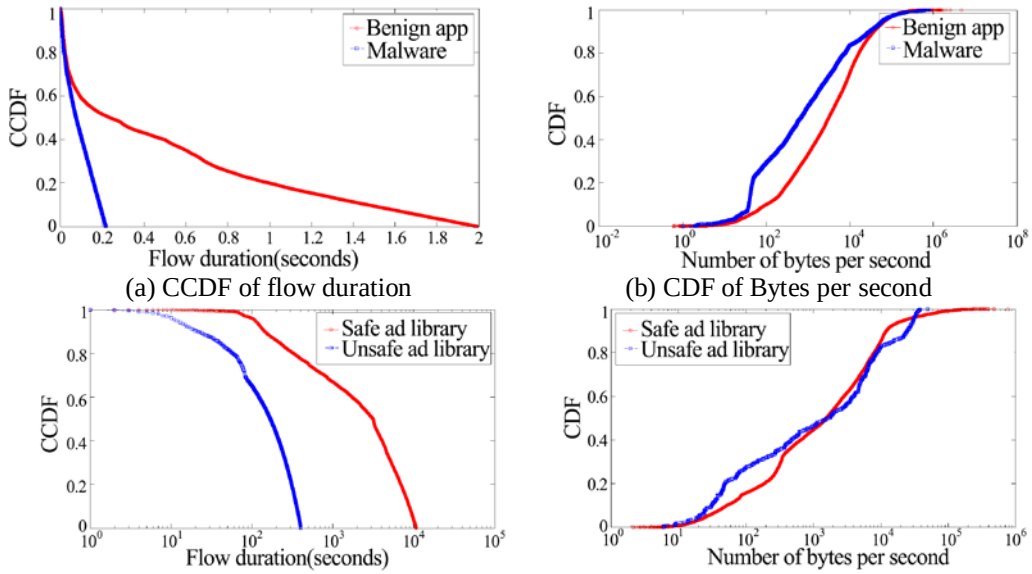
2) Timing Based Features: The second type of feature category is time-based features, which tries to capture the duration of activity of the Android app. The detail information of this kind of feature is shown in **Table 5**.

Table 5. Description of timing based features

Feature name	Description
Flow duration	TCP session length
Number of bytes per second	Number of bytes transmit between app and server per second

In order to keep the communication traffic under the radar, most malware apps use a shorter time-span than the benign apps or safe ad libraries. This work extracted two features that depend on time: flow duration and the number of bytes per second. For this feature category, complementary cumulative distribution function (CCDF) has been used, which is defined as shown:

$$F(x) = 1 - P(X < x_i) \tag{2}$$



(a) CCDF of flow duration (b) CDF of Bytes per second

(c) CCDF of flow duration (d) CDF of Bytes per second

Fig. 6. Flow Duration of HTTP Flows

The duration of flow is typically the TCP session length, which represents the amount of time an app requires to conduct its network functions with its destination server. In **Fig. 6**(a) and **6**(c), a CCDF plot of the HTTP flow duration in benign apps, malware apps and ad libraries. We notice that, for benign app and ad libraries, more than 40% flows have a duration shorter than 2 seconds. This is because many flows in benign apps and ad libraries only transfer small data like text or small image files for which the duration is short. This figure also shows that benign apps account for a larger proportion of long duration flows.

This feature represents the rate of bytes of each kind of app, which is an average measure computed over the entire duration of the app execution. **Fig. 6**(b) and **6**(d) show the CCDF of number of bytes per second in benign app, malware app and ad library. there is a clear gap can be observed, i.e., 70 bytes/s as compared to 1200 bytes/s, between benign app and malware respectively, which demonstrates that malware communication is lightweight, stealthy and ends in a short time period.

3) Semantic Features: More than 90% of apps run over HTTP protocol and that 93% of malware samples use HTTP to receive commands from their C&C servers can be found in the collected Android apps. Thus, considering this scenario, the network behavior can be correlated to the semantics of the different HTTP requests and responses. The network behavior changes with respect to the HTTP method, contacted hosts, URL paths or queries and so on.

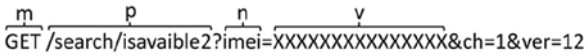


Fig. 7. HTTP request of Droidkungfu

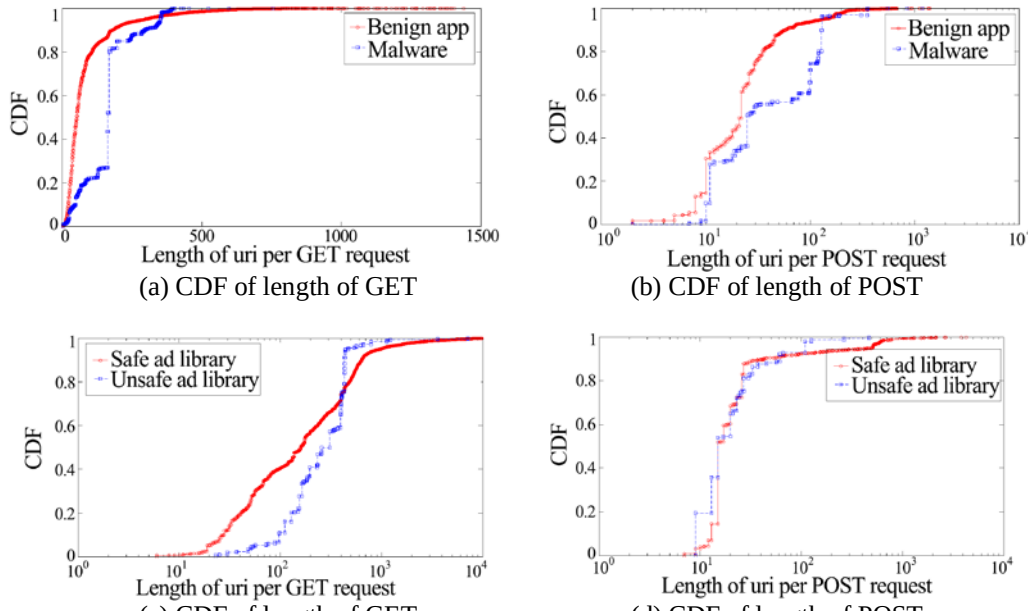
To better illustrate the HTTP request, **Fig. 7** shows a HTTP request of DroidKungfu as an example, where m represents the request method, e.g., GET, POST; p stands for page, namely the first part of the URL that includes the path and page name not including the parameters; n represents the set of parameter names and v represents value of parameters. By varying any of these variables, the network behavior and the corresponding HTTP flow features change.

Three features of importance should be considered: length of the URI POST/GET request, length of the page per GET/POST and length of the parameter per GET/POST. The detail information of this kind feature is listed in **Table 6**.

Table 6. Description of semantic features

Feature name	Description
Length of URI per GET/POST request	The number of resources requested by app
Length of page per GET/POST request	The length of paths visited by the app to obtain the resources
Length of parameter per GET/POST request	The length of parameter contain in each request

The length of URI per GET/POST request shows the number of resources requested by the app. Benign apps may request various kinds of files, whereas malware usually request commands update or leak private data out in a fixed format. In **Fig. 8**, we found that benign app flows have a clear difference with malware flows in length of URI per GET/POST request.

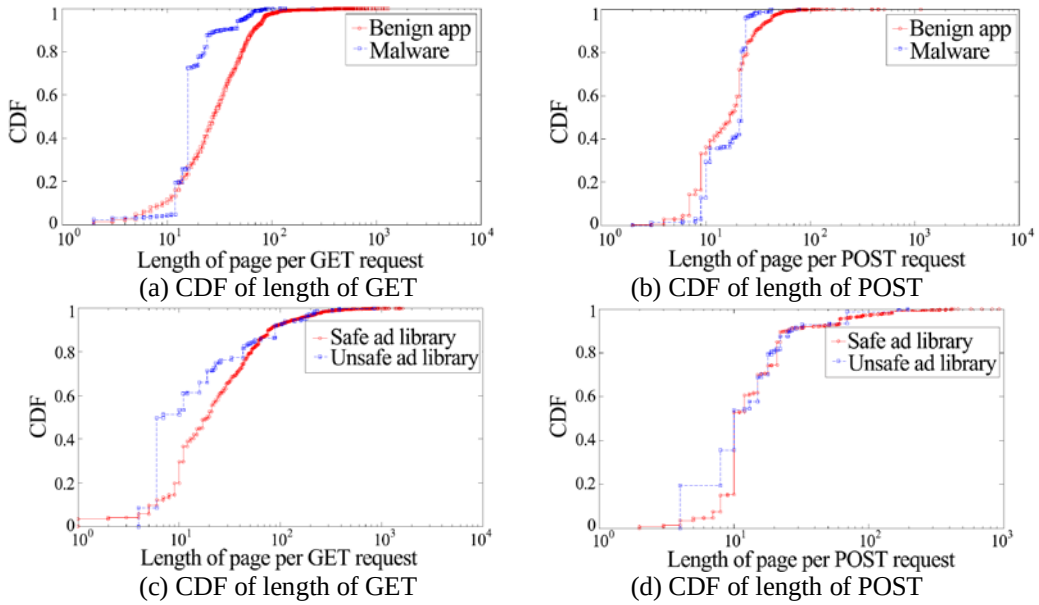


(a) CDF of length of GET (b) CDF of length of POST

(c) CDF of length of GET (d) CDF of length of POST

Fig. 8. URI lengths of HTTP requests

The length of page represents the paths visited by the app to obtain the resources and typically, the same HTTP request contains more than one resource path. Benign apps usually request multiple resources as they try to maximize the user experience and on the other hand, malicious apps request a small number of resources. In **Fig. 9**, the length of pages benign apps visited have larger length than malware in both GET and POST requests.

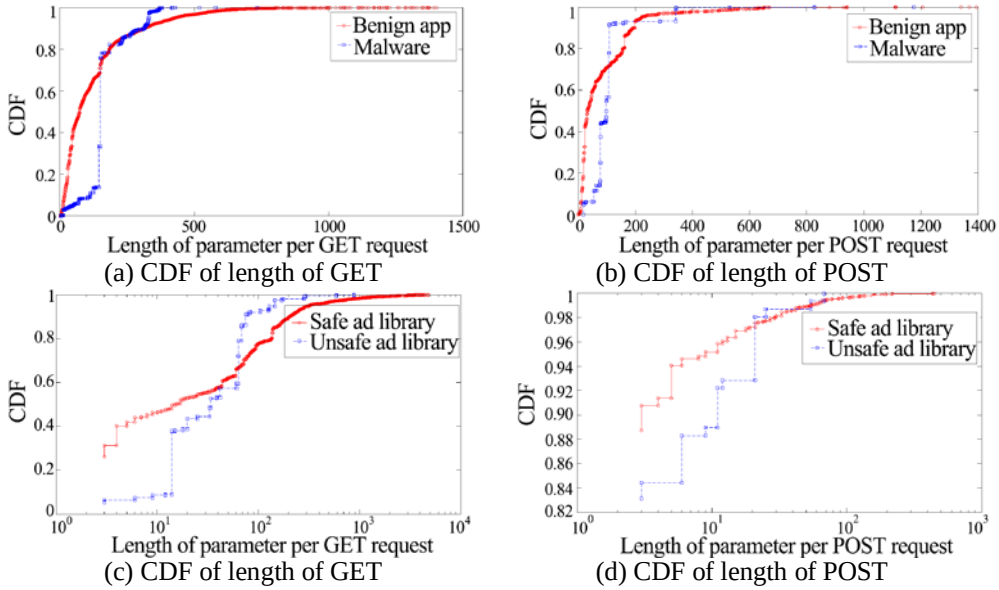


(a) CDF of length of GET (b) CDF of length of POST

(c) CDF of length of GET (d) CDF of length of POST

Fig. 9. Page lengths of HTTP requests

The GET/POST parameter is a query string, which is the part of a uniform resource locator (URL) that contains data to be passed to servers. Because benign apps have various types, they may send requests to servers with variable parameter format. However, malware ask command update and leak private data out with fixed parameter format and usually the parameter lengths are fixed within a small statistical threshold. In **Fig. 10**, the length of parameter benign apps sent to servers have larger length than malware in both GET and POST requests.



(a) CDF of length of GET (b) CDF of length of POST

(c) CDF of length of GET (d) CDF of length of POST

Fig. 10. Parameter lengths of HTTP requests

5. Categorization Using Fingerprinting

So far, this paper has shown that our HTTP flow mining approach can categorize Android malware appropriately. However, network operators and market providers need to have fine-grained knowledge about the malware families to which a particular malicious app belongs to and if a particular ad library is unsafe or not. The malware dataset used in this paper comes from Zhou et al [25], which contains 1,260 malware apps belonging to 49 different malware families. To address this issue, this work describes a categorization approach to fingerprint each malware family to aid network operators and market providers. Extracting the host name and the invariant part of the HTTP request as fingerprints to build similarity profiles for each malware family and unsafe ad library. A malware belonging to a particular malware family will have a similar fingerprint as the stored profile. The Host field is important since ad libraries usually contact the same host servers frequently and some libraries only contact one specific server. However, some malware authors use fast-flux DNS techniques to change the host name of remote servers but exhibit the same behavior across multiple flows. Therefore, in addition to the host name, identify the behavior of the malware by he structural similarity of HTTP requests.

For illustration, this work refers to the structure of the HTTP request shown in **Fig. 7**. Our approach to fingerprinting is as follows. First, divide the request into various components. The fingerprint is a three tuple $f = \{m, p, n\}$ extracted from the HTTP request where m represents the request method, e.g., GET, POST, HEADER, p stands for page, namely the first part of the URL that includes the path and page name not including the parameters, n represents the set of parameter names, e.g., $n = \{imei, ch, ver\}$. The parameter values do not consider because malware authors frequently obfuscate or encrypt them to avoid detection.

Algorithm 2. Measure similarity between two fingerprint sets

Input:
Labeled fingerprint set, fi{m,p,n}; Unlabeled fingerprint set, fj{m,p,n};

for each $f_i, i \in [0...N]$ **do**
  for each $f_j, j \in [0...M]$ **do**
    if $b(f_i.m, f_j.m) == 0$ **then**
      $d_p(f_i, f_j)$;
      $d_n(f_i, f_j)$;
      $s(f_i, f_j) = (d_p(f_i, f_j) + d_n(f_i, f_j))/2$;
      if $s(f_i, f_j) > t$ **then**
        label f_j ;
      else
        goto step 2;
      end if
    else
      goto step 2;
    end if
  end for
end for

Initially, a labeled fingerprint set has been created, and this set contains fingerprints extracted from known malware families, each family includes one or more HTTP fingerprints. Let f_i denotes the ith fingerprint of this family. Next, HTTP fingerprints from unknown malware app traffic traces have been extracted which are identified based on our detection method from Section 6.1, and create an unlabeled fingerprint set. To cluster similar fingerprints, first compare host name from each set, if the host names are matching, categorize the malware apps directly using the known labeled set. If not, use the similarity measurement algorithm shown in Algorithm 2 between the two fingerprints from the two sets to capture these similarities.

Algorithm 2 first defines a Boolean function $b(f_i, f_j)$ based on request method that is equal to 0, which means that f_i and f_j have the same value of m, e.g., both are GET or POST. Next, this algorithm defines Jaccard distance [26] as $d_p(f_i, f_j)$ and $d_n(f_i, f_j)$ to measure the similarity of p and n between f_i and f_j, respectively. After the similarity measurement, the algorithm define the overall similarity between f_i and f_j as $s(f_i, f_j) = (d_p(f_i, f_j) + d_n(f_i, f_j)) / 2$, and if the value is above a certain threshold t, we categorize f_j into the same malware family as f_i. We experimented with different values of threshold and found that a threshold of 0.6 gives good accuracy.

6. Evaluation

6.1 Android Malware Detection

This section describes our strategy for obtaining the data set suitable for evaluating our approach, the HTTP flow tracing details and the results of our classification.

1) Data Collection: Toward detecting malicious traffic, this work proposes a supervised machine learning approach based the HTTP flow features identified in Section IV. The first requirement is to obtain the training data set for the benign traffic and the malicious traffic in order to build a suitable classifier. To obtain benign apps, this work considered a trusted marketplace like Google play, downloaded 4868 well-known apps, and considered the HTTP flows of these apps as benign. However, in contrast, it is not possible to label traffic collected from malware apps as malicious in a straightforward manner because 86% malware apps are repackaged versions of legitimate applications with malicious payloads. Therefore, malware apps continue communication with benign servers and these benign traffic traces get mixed with malicious traces, thereby affecting the detection accuracy.

To address this issue, this work applied a blacklist to markup malicious app traffic. Our blacklist consists of two parts: the first contains domains associated with malware family, which are known to be malicious domains by security researchers, anti-virus companies and market providers [27], [28]. The second part is provided by MDL [29], which contains more than 80000 unique malicious domain names and IP addresses. Finally, this work extracted the same features (e.g., the features analyzed in Section 4) from both datasets to detect malware app and unsafe ad libraries. Let $A = \{a^{(i)}\}_{i=1,\ldots,N}$ be a set of Android app samples and $H\{a^{(i)}\}$ be the HTTP traffic trace collected by executing an Android app $a^{(i)} \in A$ for a

given time T, each app may generate one or more flows, $flow_j(a^{(i)})$ has been defined to denote each flow generated by the app. Translating each $flow_j(a^{(i)})$ into a pattern vector $v_j^{(i)}$ containing the features from Section 4, to model how each Android app uses the network. Using these feature vectors to train standard classifiers from Weka toolkit [30] to classify the testing set and label the flows as benign or malicious.

2) HTTP Flow Tracing: Using AndroCollector to execute Android apps in a real Android smartphone or emulator. As described in Section 3.1, the AndroCollector has two approaches to execute an Android app, specific and random. Using both techniques to collect the traffic traces as described in the following.

Specific Execution. To determine the maximum execution time under this technique, this work tested this collection method using an app called HindiSMSandJokesKhazana, which is a book app and has a large number of activities, 414, compared with other apps in our dataset. The result shows that specific behavior event executes all paths during five minutes. Based on this result, this work set the duration of traffic collection as 5 minutes and collected the traffic traces on the entire dataset. This work stored the extracted features into the storage module. The detailed results on the entire dataset are shown in **Table 7**.

Table 7. Traffic statistic from Android app using specific execution

Market/Dataset	#Unique apps	#Unique domains	#Unique IPs	#Flows	#HTTP flows
Google Play	4868	3078	6683	138229	139736
Third-party markets	4650	2614	6531	161147	159513
Android Malware Genome Project	1260	164	312	15389	15254

Random Execution. To determine maximum execution time for random events, this work used the result from [31], where the authors found that an app accessing network consumes requires about 250 seconds per user. Therefore, this work sets the execution time distribution in the range of 1 to 300 seconds. The distribution of execution time of each app follows according to the Poisson Distribution. Using random execution, the summary of our traffic trace information is shown in **Table 8**.

Table 8. Traffic statistic from Android app using random execution

Market/Dataset	#Unique apps	#Unique domains	#Unique IPs	#Flows	#HTTP flows
Google Play	4868	2359	3512	54326	50226
Third-party markets	4650	2003	3260	63337	57633
Android Malware Genome Project	1260	101	178	11940	11391

Table 9 shows some examples of the activity coverage achieved by these two methods. Specific coverage outperforms random execution in many cases but still gives reasonable results.

Table 9. Traffic statistic from Android app using random execution

App name	Total activities	Activities executed in specific mode	Activities executed in random mode
AccuWeather	16	14(87.5%)	3(18.75%)
WeatherBug	35	27(77.1%)	6(17.1%)
Wacai	82	36(43.9%)	6(7%)
QQ	82	41(50%)	11(26.8%)
Taobao	94	40(42.6%)	27(28.7%)
Wochacha	127	52(40.9%)	28(22%)

3) Results of Our Detection Approach: This experiment used a range of classifiers from Weka data mining tool kit over the training data and 14 different HTTP flow features discussed in Section 4.2 to build the classification model and tested the accuracy on the third-party apps in our data set. First, this experiment performed a 10-fold cross validation approach to find out which classification algorithm performs better. Moreover, this experiment fed HTTP flow features generated by specific and random execution into these classifiers for purpose of comparing detection results whether affect by different app execution modes.

Table 10. Results of 10-fold cross validation on app traffic generated by specific execution

Algorithm	Correctly classified	Incorrectly classified	Precision	Recall	F-Measure	ROC-Area	Classifier build time
J.48	98.798%	1.202%	0.988	0.988	0.988	0.961	24.06s
RandomForest	98.909%	1.091%	0.989	0.989	0.989	0.978	29.56s
Bayesnet	97.148%	2.852%	0.972	0.971	0.972	0.935	4.19s
SMO	93.689%	6.311%	0.878	0.937	0.906	0.5	14.42s
ZeroR	93.688%	6.312%	0.878	0.937	0.906	0.5	0.02s

Table 11. Results of 10-fold cross validation on app traffic generated by random execution

Algorithm	Correctly classified	Incorrectly classified	Precision	Recall	F-Measure	ROC-Area	Classifier build time
J.48	91.366%	8.634%	0.914	0.914	0.914	0.905	16.12s
RandomForest	92.688%	7.312%	0.927	0.927	0.927	0.916	20.79s
Bayesnet	90.868%	9.132%	0.908	0.928	0.908	0.889	1.87s
SMO	87.528%	12.472%	0.858	0.875	0.866	0.5	8.63s
ZeroR	87.528%	12.472%	0.858	0.875	0.866	0.5	0.01s

The validation results are shown in **Table 10**, **11**, Random Forest can classify more correctly instances, get higher precision, recall and f-measure in both execution modes. For the detection results, using features generated by specific execution is better than random execution. Because specific execution can generate more comprehensive features than random, Therefore, this work uses the features extracts from traffic generated by specific execution in the rest of experiments.

This experiment finally tested our classification approach on the testing set of 4650 apps and compare with two well-known Android malware detection approaches, Drebin [3] and FEST [13]. Both of them use machine learning algorithms to detect Android malware based on static features, such as permission, API calls. The results are shown in **Table 12**. To verify the result of detection, first, this experiment recalls our assumption that apps from GooglePlay

dataset are benign app, from malware dataset are malware app and from third-party are unknown. Second, this experiment uses two tools to validate our result: Androguard and Virus Total. Androguard is a well-known, open-source project with volunteer-submitted definitions and provides a lightweight signature-based malware detection tool. Virus Total is an online service that scans submitted files with more than 40 of up-to-date commercial anti-virus products and provides the results from each product. **Table 12** shows that our approach outperforms other two approaches, which correctly classified 4537 Android apps and mis-labeled 16 malware as benign app. First, our approach extracts more comprehensive features than other two approaches. Second, our approach can capture dynamic behaviors of Android malware, which can be more unique than static behaviors used by FEST and Drebin. . This experiment also found that 16 apps have been mis-labeled as malware by our detection method. These results represent a low factor of error given the efficiency of our approach in identifying malicious flows at run-time.

Table 12. Detection results comparison on testing set

Approach	Correctly classified	False positive	False negative
Our approach	4537(97.67%)	16(0.34%)	92(1.98%)
FEST	4445(95.59%)	122(2.62%)	83(1.79%)
Drebin	4408(94.79%)	47(1.01%)	195(4.19%)

6.2 Unsafe ad library Detection

This experiment extracted safe ad and unsafe ad traffic from 3454 apps, in which 3423 apps contained 33 safe ad libraries and 23 apps contained 6 unsafe ad libraries. This experiment applied the extraction rules from Section 3.2 to extract ad traffic from our traffic traces, verified our results by deep analysis of the payload of flows and compared the results with those in [19]. After verification, sosceo, plankton, waps and enerysource are unsafe libraries can be found. **Table 13** shows the results of applying our HTTP flow classification approach on this data set.

Table 13. Results of 10-fold cross validation on ad library traffic

Algorithm	Correctly classified	Incorrectly classified	Precision	Recall	F-Measure	ROC-Area	Classfier build time
J.48	98.889%	1.11%	0.988	0.989	0.989	0.922	0.76s
RandomForest	99.121%	0.879%	0.991	0.991	0.991	0.951	1.28s
Bayesnet	95.967%	4.033%	0.973	0.96	0.965	0.9	0.18s
SMO	97.202%	2.798%	0.945	0.972	0.958	0.5	1.68s
ZeroR	97.202%	2.798%	0.945	0.972	0.958	0.499	0.01s

In the testing set, this experiment used 2724 apps downloaded from several popular third-party markets, such as ZOL, Hiapk and so on, which generated 16818 HTTP flows in total. Note that, an ad library may be embedded in multiple apps and an app might contain several ad libraries. Therefore, this experiment classify the ad traffic with respect to the number of flows and not with respect to the number of apps. **Table 14** shows the results of our classification technique using multiple classifiers, among which Random Forest gives the best possible performance.

Table 14. Results of ad library traffic testing set classification

Algorithm	Correctly classified	False positive	False negative	Time
J.48	15513(92.24 %)	490(2.91%)	815(4.85%)	2.9s
RandomForest	16121(95.86%)	329(1.95%)	368(2.19%)	3.3s
Bayesnet	15310(91.03%)	1069(6.36%)	439(2.61%)	3.3s
SMO	15611(92.82%)	498(2.96%)	709(4.22%)	1.9s
ZeroR	15611(92.82%)	498(2.96%)	709(4.22%)	1.2s

6.3 Results of Categorization

To evaluate our categorization approach, this experiment focuses on the malware which have been identified by our detection **Table 15** shows the results of our categorization. The table shows a distribution of 75 apps which are clustered into the corresponding malware family. Noting that the most prevalent malware family in our Android app dataset was DroidKungfu. This malware is capable of rooting the vulnerable Android phones and may successfully evade the detection from current mobile anti-virus software.

Table 15. Results of categorizing Android malware

Malware family	# of apps
ADRD	3
DroidDreamLight	6
DroidKungfu	50
GoldDream	6
Plankton	10

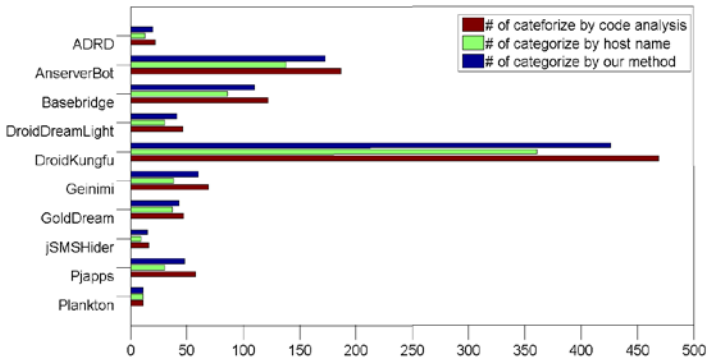


Fig. 11. Accuracy of malware app Categorization

To further demonstrate our categorization method, this experiment selected 10 popular malware families from the 49 malware families of our dataset, which use the HTTP-based web traffic to receive bot commands from their C&C servers. This experiment used AndroCollector to collect traffic from malware contain in the 10 malware families, extracted the host names and fingerprints from malware traffic traces and applied Algorithm 2 to categorize them. **Fig. 11** shows the results our categorization approach. **Fig. 11** can conclude that host names in HTTP headers can categorize malware. Especially, 11 malware in Plankton have been categorized by using the host name. However, only 72.2% of malware were successfully categorized by using host name, which means that the host name is not sufficient for all malware and relied on the similarity algorithm to categorize other malware. These results show the validity of our categorization approach in providing useful information to the

network operators regarding various malware infection classes and targets.

7. Conclusion

This paper proposed a novel method to detect and categorize Android malware and unsafe ad libraries based on HTTP traffic flow mining. First, this paper designed AndroCollector, to capture traffic traces automatically, which can eliminate the manual labor involved in executing an app under various conditions and user interactions. Next, this paper extracted HTTP traffic characteristics for malware apps and applied state machine model to extract ad traffic from the mixture of Android app network traffic traces. This paper provided a detailed analysis of the HTTP traffic characterization and showed the correlation to app behavior. This paper applied classification algorithms to detect malware and unsafe ad libraries based on several HTTP flow features. Finally, this paper extracted host name and invariant parts of HTTP request headers as fingerprints to categorize malware and the unsafe ad libraries. Our categorization helps market providers in making informed decisions when dealing with specific malware incidents. Our comprehensive evaluation shows that our method has high accuracy and validates our approach can characterize Android malware behavior.

8. Acknowledgement

This work is supported by the Research Foundation of Education Bureau of Hunan Province, China(No.16B085, No.16C0047), the Science and Technology Projects of Hunan Province (No.2016JC2074, No.2016JC2075), the Open Research Fund of Key Laboratory of Network Crime Investigation of Hunan Provincial Colleges(No.2016WLFZZC008), the National Science Foundation of China(No.61471169), the Key Lab of Information Network Security, Ministry of Public Security (No.C16614).

References

[1] McAfee. http://www.mcafee.com, 2012.
[2] Antonio Bianchi Christopher Kruegel Sebastian Poeplau, Yanick Fratantonio and Giovanni Vigna, “Execute this! analyzing unsafe and malicious dynamic code loading in android applications,” in *Proc. of Network & Distributed System Security Symposium*, 1-16, 2014. Article (CrossRef Link).
[3] Malte Hubner Hugo Gascon Daniel Arp, Michael Spreitzenbarth and Konrad Rieck, “Drebin: Effective and explainable detection of android malware in your pocket,” in *Proc. of Network & Distributed System Security Symposium*, 2014. Article (CrossRef Link).
[4] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Profiledroid: Multilayer profiling of android applications,” in *Proc. of the 18th annual international conference on Mobile computing and networking*, 11(1): 137-148, 2012. Article (CrossRef Link).
[5] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W Lee, “The core of the matter: Analyzing malicious traffic in cellular carriers,” in *Proc. of Network & Distributed System Security Symposium*, 2013.
[6] N. Vallina-Rodriguez, J. Shah, A. Finamore, H. Haddadi, and et al., “Breaking for commercials: Characterizing mobile advertising,” in *Proc. of the 2012 Internet Measurement Conference*, 343-356, 2012. Article (CrossRef Link).
[7] T.T.T. Nguyen and G. Armitage, “A survey of techniques for internet traffic classification using machine learning,” *Communications Surveys Tutorials*, IEEE, 10(4):56–76, 2008. Article (CrossRef Link).

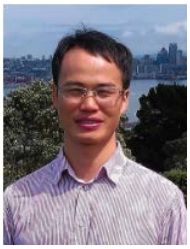
[8] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: automatic protocol reverse engineering from network traces," in *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.
[9] P. Royal, "Analysis of the kraken botnet," *Technical report*, Damballa Labs, 2008.
[10] S. Hao, N. Feamster, and R. Pandrangi, "An internet wide view into dns lookup patterns," *Technical report*, Verisign Labs, 2010.
[11] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "Dns performance and the effectiveness of caching," *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002. Article (CrossRef Link).
[12] Xin Su, Dafang Zhang, Wenjia Li, and Kai Zhao, "A Deep Learning Approach to Android Malware Feature Learning and Detection," in *Proc .of Trustcom 2016*: 244-251, 2016. Article (CrossRef Link).
[13] K. Zhao, D.F. Zhang, X. Su, and W.J. Li, "Fest: A feature extraction and selection tool for android malware detection," in *Proc .of 2015 IEEE Symposium on Computers and Communication*, 714-720, 2015. Article (CrossRef Link).
[14] Bin Gu and Victor S. Sheng, "A Robust Regularization Path Algorithm for ν-Support Vector Classification," *IEEE Transactions on Neural Networks and Learning Systems*, 1:1-8, 2016. Article (CrossRef Link).
[15] Yuhui Zheng, Byeungwoo Jeon, Danhua Xu, Q.M. Jonathan Wu, and Hui Zhang, "Image segmentation by generalized hierarchical fuzzy C-means algorithm," *Journal of Intelligent and Fuzzy Systems*, 28(2): 961-973, 2015. Article (CrossRef Link).
[16] Xuezhi Wen, Ling Shao, Yu Xue, and Wei Fang, "A rapid learning algorithm for vehicle classification," *Information Sciences*, 295(1): 395-406, 2015. Article (CrossRef Link).
[17] Bin Gu, Victor S. Sheng, Zhijie Wang, Derek Ho, Said Osman, and Shuo Li, "Incremental learning for ν-Support Vector Regression," *Neural Networks*, 67: 140-150, 2015. Article (CrossRef Link).
[18] Gu Bin, Victor S. Sheng, and Shuo Li, "Bi-parameter space partition for cost-sensitive SVM," in *Proc. of the 24th International Conference on Artificial Intelligence*, 3532-3539, 2015 Article (CrossRef Link).
[19] M. C. Grace, W. Zhou, X. Jiang, and A. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 101-112, 2012. Article (CrossRef Link).
[20] H. Haddadi, P. Hui, and L. Brown, "Mobiad: private and scalable mobile advertising," in *Proc. of MobiArch*, 2010. Article (CrossRef Link).
[21] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis, "Serving ads from localhost for performance, privacy, and profit," in *Proc. of Hot Topics in Networking*, 2009.
[22] Monkeyrunner. http://developer.android.com/tools/help/monkeyrunnerconcepts.html, 2012.
[23] Hierarchy Viewer. http://developer.android.com/tools/help/hierarchy-viewer.html, 2010.
[24] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of the 2012 IEEE Symposium on Security and Privacy*, 95-109, 2012. Article (CrossRef Link).
[25] Zhangjie Fu, Xingming Sun, Qi Liu, Lu Zhou, and Jiangang Shu, "Achieving Efficient Cloud Search Services: Multi-keyword Ranked Search over Encrypted Cloud Data Supporting Parallel Computing," *IEICE Transactions on Communications*, E98-B(1): 190-200, 2015. Article (CrossRef Link).
[26] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, 51(1):117–122, 2008. Article (CrossRef Link).
[27] AnserverBot. "Security alert: Anserverbot," *new sophisticated android bot found in alternative android markets*, 2012. Article (CrossRef Link).
[28] DroidKungFu, http://www.fortiguard.com/encyclopedia/virus/android droidkungfu.a!tr.html, 2012.
[29] MDL. http://www.malwaredomainlist.com/ mdl.php.
[30] Weka. http://www.cs.waikato.ac.nz/ml/weka/.
[31] Q. Xu, J. Erman, A. Gerber, Z.Q Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proc. of the 2011 ACM SIGCOMM conference on Internet measurement conference*, 329-344, 2011. Article (CrossRef Link).

Xin Su received his Ph.D degree from college of Information Science and Electronic Engineering, Hunan University, Changsha, China, in 2015. He is currently a Lecture at the Hunan Provincial Key Laboratory of Network Investigational Technology, Hunan Police Academy, Changsha, China. His research interests include network security, mobile phone security, big data mining.

Xuchong Liu received his Ph.D degree from school of Information Science and Engineering, Central South University, Changsha, China, in 2010. He is currently a Professor at the Hunan Provincial Key Laboratory of Network Investigational Technology, Hunan Police Academy, Changsha, China. His research interests include Information security, network security, big data analysis.

Jiuchuang Lin received his Master degree from Fudan University, Shanghai, China, in 2011. He is currently a vice director at the Key Lab of Information Network Security of Ministry of Public Security, the Third Research Institute of Ministry of Public Security, Shanghai, China. His research interests include computer bud mining, system security assessment.

Shiming He received her Ph.D degree from Hunan University, Changsha, China, in 2013. She is currently a lecture at Changsha University of Science and Technology, Changsha, China. Her current research interests include privacy preserving, wireless network and mobile computing.



Zhangjie Fu received his Ph.D degree from Hunan University, Changsha, China, in 2012. He is currently a vice professor at School of Computer and Software, Nanjing University of Information Science & Technology, Nanjing, China. His research interests include cloud computing, Android security and big data security.

Wenjia Li received his Ph.D degree from University of Maryland Baltimore County in 2011. He is currently an assistant professor at New York Institute Technology. His research interests include cyber security, mobile computing.