Research Article

AppFA: A Novel Approach to Detect Malicious Android Applications on the Network

Gaofeng He,¹ Bingfeng Xu ,² and Haiting Zhu¹

¹*School of Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing 210003, China*

²*College of Information Science and Technology, Nanjing Forestry University, Nanjing 210037, China*

Correspondence should be addressed to Bingfeng Xu; bingfengxu@njfu.edu.cn

Received 21 October 2017; Revised 29 January 2018; Accepted 13 March 2018; Published 17 April 2018

Academic Editor: Georgios Kambourakis

Copyright © 2018 Gaofeng He et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We propose AppFA, an Application Flow Analysis approach, to detect malicious Android applications (simply *apps*) on the network. Unlike most of the existing work, AppFA does not need to install programs on mobile devices or modify mobile operating systems to extract detection features. Besides, it is able to handle encrypted network traffic. Specifically, we propose a constrained clustering algorithm to classify apps network traffic, and use Kernel Principal Component Analysis to build their network behavior profiles. After that, peer group analysis is explored to detect malicious apps by comparing apps' network behavior profiles with the historical data and the profiles of their selected peer groups. These steps can be repeated every several minutes to meet the requirement of online detection. We have implemented AppFA and tested it with a public dataset. The experimental results show that AppFA can cluster apps network traffic efficiently and detect malicious Android apps with high accuracy and low false positive rate. We have also tested the performance of AppFA from the computational time standpoint.

1. Introduction

Recently, the mobile platform has gained more and more popularity, and there are a large amount and wide variety of feature-rich mobile applications (or apps) that users can install and experience. As an example, the Google Play store has already had more than 3 million apps by Sep. 2017 [1]. With these feature-rich apps, mobile users can work, play games, and communicate with each other anytime and anywhere. Meanwhile, since a majority of these apps need Internet access, they are bringing challenges to network security and management. For instance, malicious apps can compromise mobile users' privacy and steal users' confidential data [2, 3]. As indicated in [4], more than 70% of known malicious mobile apps (also known as mobile malware) steal user credentials and information. Besides, mobile malware is also proposing new challenges to security protection of enterprise networks. Malicious apps may be part of many bots and cause a dramatically increasing influence on the network traffic (DoS attack) [5].

Researchers have done extensive work to detect malicious mobile apps and several methods have been proposed and

evaluated [6–12]. Commonly, in order to detect malicious mobile apps, several steps should be done. First, detection features such as user's operating behavior, API usage, and application network behavior should be defined and extracted. Then detection models are constructed and, finally, new apps are compared with the constructed models for mobile malware detection. Depending on the locations where these steps are performed, current approaches can be generally categorized into two main groups: *client-side* and *server-side* detection. For client-side detection, these steps are all performed at mobile devices, while, for server-side detection, the main steps are carried out on remote servers.

Even though server-side detection approaches are conducted remotely, their detection features are mainly collected and processed on mobile devices and later sent to remote servers for modeling and detection [13]. Therefore, current approaches are required to install some kind of program on mobile devices or modify operating systems (e.g., modify Android source code) to collect detection feature information. Obviously, this will increase energy consumption of mobile devices. Also, these methods will be difficult to be applied for mobile device protection in large organizations.

As illustrated in literature [14], it is hard to ensure that all mobile devices have installed information collection programs and it is impractical to manually audit every employee's personal device due to the privacy issue and also the large amount of mobile devices.

In this paper, we propose AppFA (App Flow Analysis), a novel approach to detect malicious Android apps from network traffic. Contrary to client-side and server-side methods, AppFA is implemented at the network level and, thus, it is a new kind of *network-side* detection approach. Notably, AppFA does not need to install programs or modify operating systems to extract detection features; therefore, it is lightweight and easy to deploy. We notice that literature [15] is also a kind of network-side approach, while it only analyzes HTTP traffic and cannot be applied to encrypted network traffic. Previous works [6, 7] have also proposed methods to detect mobile malware from network traffic. These methods need offline training and install programs (such as VPN proxy) on mobile devices to get to know exactly what flows come from what app for traffic labelling. Thus they still belong to client-side or server-side approaches.

In this work, AppFA can analyze encrypted network traffic through network behavior profile construction. The apps traffic is clustered by constrained clustering; thus we do not need to install programs on mobile device to determine the origin of apps traffic. We also use peer group analysis to avoid offline model training. Our main contributions are the following:

- (i) Providing a lightweight and efficient framework for detecting malicious Android apps on the network
- (ii) Proposing an efficient algorithm for clustering mobile apps network traffic
- (iii) Outlining a method for detecting malicious apps by constructing network behavior profile and using peer group analysis
- (iv) Carrying out extensive experiments with the public dataset.

The rest of this work is structured as follows: The motivation of our work is presented in Section 2. In Section 3, we discuss relevant related work in detail. Section 4 introduces the architecture and main components of AppFA. The methodology is presented in Section 5. Experimental evaluation and discussion of the proposed methods are presented in Section 6. Finally, in Section 7, we conclude the paper with a discussion of potential future work.

2. Motivation and Observations

As investigated by Statista, Android accounts for more than 86% of the global mobile OS market until the 1st quarter of 2017 [16]. The popularity of Android devices makes it a desirable target. For example, the top 20 mobile malware programs are all related to Android [17]. One of the reasons for the popularity of Android malware may be that Android app package elements can easily be modified by third parties [18]. With open-source tools such as

apktool (<https://github.com/iBotPeaches/Apktool>) and jadx (<https://github.com/skylot/jadx>), malware writers can easily graft some malicious code on popular apps to ensure a wide diffusion of their malicious code. As an evidence, MalGenome [19], a reference dataset in the Android security community and also used in our experiments, has 80% of the malicious samples known to be built via repackaging other apps. Therefore, in this work, we mainly focus on detection of Android malicious repackaged apps.

It is popular to detect Android malware by code and resource file analysis [20], while there are very few studies considering malicious Android apps detection at the network level. In order to detect Android malware on the network, we have analyzed network traffic of Android apps carefully and obtained several observations. The first observation is that network behaviors of repackaged apps are significantly different from those of their original versions. This observation is also validated by Shabtai et al. [7]. Taking the Android malware *AnserverBot* [21] as an example, network behaviors of the repackaged app and the original app (*com.camelgames.mxmotor*) are compared in Table 1.

For comparison, the two apps were run in a real phone, respectively, and the network traffic was collected in the first 5 minutes. After that, the total packet sizes and the amount of tcp connections were calculated as network behaviors. Obviously, as shown in Table 1, there is a clear difference between the network behavior of the repackaged and original apps; particularly the total packet sizes of the repackaged app are significantly larger than the original app (2.3 Mb versus 193.2 kb).

We further compared apps' network behaviors with their similar apps. Again, taking *com.camelgames.mxmotor* as the example, we selected its 3 similar apps from the Google Play store. The chosen strategies are described as flows: we first searched the key word "*moto*" in the Google Play store; then we selected the top 3 game apps from the search results as the similar ones. The network behaviors of these apps are compared as in Table 2. Apparently, their network behaviors are close to each other. This phenomenon of similar apps having similar network behaviors is also validated in [7]. These preliminary observations of network behaviors of Android apps give us clues for detection of Android repackaged malware.

We also observed that more and more Android apps adopt encrypted network connections to transmit data between smart devices and remote servers. For instance, *com.baidu.BaiduMap* generates several SSL flows at the startup stage, as shown in Figure 1. Therefore, methods for detecting Android repackaged malware should handle encrypted network traffic.

Based on the observations above, we propose to detect malicious Android apps by comparing apps' network behaviors with their historical data and the ones of their similar apps. Comparison with the historical data can help us find out self-updating malicious apps, and comparison with the behaviors of similar apps can detect other types of repackaged malware. Self-updating is a new technique for repackaging apps and it cannot be detected by applying regular static or dynamic analysis methods [7]. The details of network behavior construction and similar app selection are illustrated

Server host	S. port	Protocol (application layer)
123.125.112.76	443	Ssl
119.75.222.71 [client.map.n.shifen.com] [clie...	443	Ssl
117.122.217.133 [www.qqlog.cn]	80	Http
114.112.103.15 [www.qqchannel01.cn]	80	Http
61.136.186.33 [ofloc.map.n.shifen.com] [ofloc...	80	Http
114.112.103.15 [www.qqchannel01.cn]	80	Http
117.122.217.130 [117.122.217.130]	80	Http
112.80.255.99 [ofloc.map.n.shifen.com] [ofloc...	80	Http
23.201.102.36 [ai015.g.akamai.net] [allto2.cl...	80	Http
61.136.186.152 [static.n.shifen.com] [asolick...	443	Ssl
118.194.55.187	5223	Ssl
222.199.191.31 [szlshare.jonodns.com] [gsr0.b...	443	Ssl
119.75.222.71 [client.map.n.shifen.com] [clie...	443	Ssl
119.75.222.71 [client.map.n.shifen.com] [clie...	443	Ssl
119.75.222.71 [client.map.n.shifen.com] [clie...	443	Ssl
112.80.255.99 [ofloc.map.n.shifen.com] [ofloc...	443	Ssl
111.202.114.105 [newvector.map.n.shifen.com] ...	443	Ssl

FIGURE 1: Example of network flows of Android app *com.baidu.BaiduMap*. The traffic is captured and analyzed by Networkminer. Note that there exists several encrypted SSL flows and the number is more than HTTP traffic.

TABLE 1: Comparison of network behaviors of original app *com.camelgames.mxmotor* and its repackaged version (repackaged by AnserverBot).

	Total packet sizes	Number of tcp connections
Repackaged app	2.3 Mb	18
Original app	193.2 kb	11

TABLE 2: Comparison of network behaviors of similar apps.

	Total packet sizes	Number of tcp connections
com.camelgames.mxmotor	193.2 kb	11
air.com.aceviral.motox3m	182.8 kb	14
com.skgames.trafficrider	175.1 kb	13
com.topfreegames.bikeracefreeworld	132.7 kb	9

in Section 5. Meanwhile, a novel constrained clustering algorithm is elaborated for app traffic clustering. Thus our method can be applied on the network straightforwardly and does not need to install programs on mobile devices to collect flow information.

3. Related Work

There has been extensive work on detecting malicious mobile apps. Literature [4, 5, 22, 23] gave surveys of mobile malware in the wild and the proposed techniques for detecting them. In this section, we mainly focus on behavior-based malware detection methods and only review the most related ones.

Generally, current behavior-based mobile malware detection approaches can be categorized into two main groups: *client-side* and *server-side* detection. Client-side detection approaches run locally and apply anomaly methods on the set of features which indicate the state of the app. The pBMDS [8] is based on correlating user inputs with system calls to detect anomalous activities. A Hidden Markov Model (HMM) is used to learn application and user behaviors from two major aspects: process state transitions and user operational patterns. Built upon these two aspects, the pBMDS identifies behavioral differences between user initiated applications and malware compromised ones. Zhang et al. [11] combined dynamic tracing of the permission requests for resources

usage by applications with tracking sensitive operations on the granted resources (using taint tracking). This combination enabled them to understand how applications utilize the permissions to access sensitive system resources. Dai et al. [24] presented a malware detection system for the Windows Mobile platform. They used API interception techniques for monitoring and analyzing the application's behavior and compared it to the patterns within the predefined library of malicious behavior characteristics. Shabtai et al. [7] presented a behavior-based anomaly detection system for detecting meaningful deviations in a mobile application's network behavior. Semisupervised C4.5 Decision Tree algorithm was used for learning the normal behavioral patterns and for detecting deviations from the application's expected behavior. Their methods were implemented and evaluated on Android devices. Damopoulos et al. [25] proposed a fully fledged tool able to dynamically analyze any iOS software in terms of method invocation that can be used to trace software's behavior to decide if it contains malicious code or not.

Server-side detection approaches are carried out on remote servers mainly motivated by limited computational resources of the mobile device [26]. Burguera et al. [9] have developed an Android framework named "Crowdroid" that includes a client application installed on the device. The application monitors Linux kernel system calls and sends them to a centralized server after preprocessing. On the server, a dataset is built from the list of the system calls, the list of running applications and the device information. *K*-Means algorithm is then used for clustering the applications into two groups, that is, benign and malware applications. Shamili et al. [13] utilize a distributed Support Vector Machine algorithm for malware detection on a network of mobile devices. The phone calls, SMSs, and data communication related features are used for detection. During the training phase support vectors (SVs) are learned locally on each device and then sent to the server where SVs from all of the client devices are aggregated. Finally, the server distributes the whole set of SVs to all the clients, and each client updates his own SVs.

Unlike the work mentioned above, our proposed system runs at the network level directly without necessarily having access to the mobile devices. We notice that the most similar work was carried out by Chen et al. [15]. Chen et al.'s method [15] was also implemented at the network level and identified abnormal network behaviors by conducting 3-step check action, including (1) identifying HTTP POST and HTTP GET packages, (2) checking whether the device was exposing unique device identifiers such as IMEI and IMSI, and (3) determining the legitimacy of the remote server by querying the domain name server. So their method can only be applied to HTTP traffic. Contrary to the work of Chen et al. [15], we use the combination of signature matching and constrained mobile network traffic clustering for app identification and, thus, our method is suitable for both plaintext and ciphertext traffic such as HTTPS. The work introduced in [6] is also intended to detect Android malware from network traffic. In Garg et al. [6], detection features were first extracted from DNS, HTTP, and TCP traffic and then machine learning algorithms (such as Decision Trees, Bayesian Networks, and Random Forests) were used to detect malicious mobile apps,

FIGURE 2: Main components of AppFA.

while their detection features were obtained in the mobile device and they needed to train classification models offline. In AppFA, we straightforwardly extract detection features on the network to build apps' network behavior profiles and use peer group analysis to avoid offline model training.

Recently, there are an increasing number of research works that analyze network traffic to identify app, such as [27–32]. However, most of them were focused on plaintext flows (e.g., HTTP) and tried to collect identification features from HTTP headers. These methods may fail due to the emergence of encrypted network traffic. In this paper, we investigate a new constrained clustering method to cluster the network traffic generated by the same app.

4. System Design

The system architecture of AppFA is shown in Figure 2. It is of modular design and each module has special functions. *Packet Filter* module captures link-layer frames or reads them from a file, and filters them according to configurable rules. In order to detect malicious apps online and provide support for network management, AppFA is designed to analyze the first n nonzero packets (packets contain application data) of each network flow. The parameter n is configurable for different network management purpose. For example, the value of n can be set to a small positive number such as 50 for real-time malicious app detection and -1 for full analysis, namely, considering all nonzero packets in flows. In AppFA, the packet filter module can be implemented based on well-known library such as *libpcap* (<http://www.tcpdump.org/release/>).

Session Builder module organizes network traffic into sessions. For app identification and malicious behavior

detection, we define two types of sessions: *flow session* and *app session*. The *flow session* is defined by the {source IP, source port, destination IP, destination port, transport protocol} tuple, where source and destination can be swapped and the transport protocol is mainly considered as TCP and UDP in this work. In deployment, AppFA determines when a flow session is completed by one of the following three conditions: (1) received n nonzero packets; (2) detecting REST/FIN packet; (3) timeout; for example, there is no packet exchanging in 3 minutes. With flow sessions, one can extract basic features and packet contents efficiently for app identification and malicious behavior detection. The *app session* is defined as the collection set of all flow sessions generated by the same app: {flow_session_1, flow_session_2, ..., flow_session_m}. The app sessions constructed by *Session Builder* module will be invoked by *Profile Feature Generator* module to construct app network behavior profiles, as depicted in the right of Figure 2.

Basic Feature Extractor module extracts basic packet features from flow sessions, which include *packet size*, *packet interarrival time*, *packet order*, and *packet direction*. The packet direction feature distinguishes outgoing from incoming packets. Other advanced features such as flow duration, total packets sent, and received and burst sizes commonly used in traffic analysis [33–35] can be calculated from these basic features. So we extract these basic features firstly and then generate appropriate advanced features (identification or detection features) for different traffic analysis purposes (clustering or peer group comparing). After the basic feature extraction, a flow may look like {+50, *30, +100, *500, -1300, *20, -400, ...}, where outgoing packet sizes and incoming packet sizes are denoted by positive and negative


```

> Form item: "dur" = "A71YLvXU0s="
> Form item: "pv" = "NcGGfm3G+vA="
> Form item: "dtype" = "pkVRNzuxWio="
> Form item: "ts" = "U3q+Jhme19wVZQETebnQ=="
> Form item: "appkey" = "Z8d1/T6KV8AvnLZ7CmDJz5V00chYvtoIm1gzXmeLrk="
> Form item: "channel" = "G8dJsw51aTGcfHBEVsk0q=="
> Form item: "uid" = "LUWzPaPEKc//WbWq6Hj3pIzsuS0BbpaX/2fywKbrnKrAEFwMn+mg=="
> Form item: "nt" = "1y8T+1sBZKjYtr+Hnzr30A=="
> Form item: "nuId" = "HYmw5y26YyWn/X3uZmazcudy71fD6F+kR8JNRWIs3jhXeRK2z1A=="
> Form item: "adr" = "e06tbtR2ih0="
> Form item: "did" = "g5aEv8ytp5d/f6C975r+f4w=="
> Form item: "mid" = "keRg4wVvRuI0PQ+zbrLgu9G2gumQ7X9uVh1IHpcC0Jg3jvHVzy61Q=="
> Form item: "uid3" = "8a+3M1gf8EANGDR+8wVp+dt213ZuJdc1gKw05sTas2Tu913H1nc/rsK6HCU hv3Sv9gaXJLw5pY="
> Form item: "inschannel" = "G8dJsw51aTGcfHBEVsk0q=="
> Form item: "e" = "1"

```

FIGURE 3: Example of key-value pairs.

signs, and the packet interarrival times are labelled by asterisk. The packet order features are also reflected by the number sequences inherently. For example, for the flow $\{+50, *30, +100, *500, -1300, *20, -400, \dots\}$, the first packet is an outgoing packet whose size is 50 bytes, and the second packet is also an outgoing packet and the packet time interval is 30 ms. The third packet is an incoming packet whose size is 1300 bytes. The time interval between the second and the third packet is 500 ms and so forth.

Note that the packet contents are equally saved through *Packet Content Extractor* module for app identification. Similar to work [30], we focus on key-value pairs in HTTP headers. In detail, *justniffer* (<http://justniffer.sourceforge.net/>) is first used to transform the raw packet traffic into HTTP messages. Then, HTTP messages are tokenized by several tokenizers such as space, “\r\n”, “?”, and “&” and each HTTP request will be broken into various parts including method, page, and query. Finally, queries are divided into key-value pairs. Figure 3 is an example of key-value pairs used in our experiments.

After obtaining the basic features from flows, we begin to generate identification features in *Identification Feature Generator* module. After that, we will identify the app for each flow through *App Traffic Clustering* module, which returns the clustering results to *Session Builder* module for forming app sessions. With the app session information, detection features will be created in *Profile Feature Generator* module. At last, app network behavior profiles will be constructed in *App Behavior Profile Constructor* module and malicious apps will be detected in *Malicious App Detection* module, respectively. The independent treatment of different functional modules can make AppFA architecture clear and scalable.

5. Methodology

As depicted in Figure 2, the functionality of app traffic clustering and malicious app detection (including identification/detection feature generation) are the core components of AppFA and the technical details are clarified in the next subsections.

5.1. App Traffic Clustering. The basic idea of our app traffic clustering method is illustrated in Figure 4. In the figure, there are two apps and the flow sessions of them are represented by solid and dotted lines, respectively. For each flow, signature matching is first used to identify HTTP flows that have recognized signatures. Note that, in this paper, we use the term *signature* for plaintext matching and *feature* for

traffic analysis. With this step, there are two flows identified (labelled as red and green), as shown in the left dashed box of Figure 4. After the signature matching, constrained clustering algorithm (the second dashed box in Figure 4) is exploited to cluster all flows. Compared to the ordinary clustering algorithms, constrained clustering algorithm adopts background information (i.e., identified flows) to improve cluster accuracy. By constrained clustering, flows such as encrypted traffic that cannot be identified by signatures will be classified into appropriate clusters, that is, apps. Finally, we obtain app sessions that will be utilized for creating apps’ network behavior profiles, as depicted in the last step in Figure 4.

Formally, the entire app identification procedure is described in Algorithm 1. In the *while loop*, the clustering signal can be a timeout for cyclic identification (e.g., every 1 hour) or an app session completed for real-time identification. For the details of selecting the initial signature seeds S_d , one can refer to literature [30].

Algorithm 1 uses a method similar to the FLOWR system [30] to carry out signature matching (line (6)) and takes advantage of constrained *K*-means clustering algorithm [36] for flow clustering (line (7)). For signature matching, the key-value pairs in HTTP header are considered as app’s signatures and an initial set of seeding app signatures is set up to bootstrap the learning of new ones. Compared to FLOWR, we refine the process of counting cooccurrence of app signatures with the constrained clustering results. In FLOWR, if start time of $flow_2$ is less than T seconds after the start time of $flow_1$, their signatures will be considered as a cooccurrence instance. However, as noted in literature [30], if T is overestimated, FLOWR is more likely to mix flows from different apps, thus inducing noise and overutilizing system resources. To overcome this problem, in AppFA, we further consider the clustering results to count cooccurrence of app signatures besides temporal information. That means *only the flows that occurred in T seconds in the same cluster will be counted as cooccurrence*. This will reduce noises since the flows are filtered by constrained clustering.

After signature matching, constrained *K*-means clustering is carried out to identify the remaining unknown flows. Algorithm 2 shows the constrained clustering algorithm exploited in AppFA. In the constrained flow clustering, the flows identified by signature matching which belong to the same app must be clustered into one cluster (must-link constraints, lines (4)–(8) in Algorithm 2), and those generated by different apps must be clustered into different clusters (cannot-link constraints, lines (9)–(13) in Algorithm 2). The clustering features are listed in Table 3 (totally 11 features). We select the time of the first packet sent as one of the features because flows observed within short time intervals are likely to come from the same app [32]. The other features are chosen as they are proved to be efficient in clustering network traffic [36].

In Algorithm 2, we set the value of cluster number p to be equal to the number of apps identified by signature matching. This is because the accuracy of mobile app identification is already higher than 95% [29, 32] and it is proper to assume that the popular apps (malware writers usually graft some malicious code on popular apps to ensure a wide diffusion

FIGURE 4: Illustration of the basic idea of app identification. Solid and dotted lines represent different apps and black color indicates that the flows have not been identified.

- (1) let S denotes signature set and S_d stands for the signature seeds
- (2) **while** the clustering signal is received **do**
- (3) **if** S is empty **then**
- (4) $S = S_d$
- (5) **end if**
- (6) carry out signature matching
- (7) carry out constrained flow clustering
- (8) update S with the clustering results
- (9) **end while**

ALGORITHM 1: App traffic clustering procedure.

TABLE 3: Identification feature set.

Time of the first packet sent
Number of packets
Volume of bytes
Min., max., mean, and std. dev. of packet size
Min., max., mean, and std. dev. of interpacket time

TABLE 4: Network behavior profile feature set.

Number of flows
Number of packets of outgoing and incoming flows
Volume of bytes of outgoing and incoming flows
First k_1 results of KPCA transformations of outgoing packet sizes
First k_2 results of KPCA transformations of incoming packet sizes
First k_1 results of KPCA transformations of outgoing packet intervals
First k_2 results of KPCA transformations of incoming packet intervals

of their malicious code) can all be recognized. Therefore, each cluster will correspond to one app when the constrained K -means clustering is finished. This is appropriate for the following network behavior profile construction and malicious behavior detection. In practice, if the actual number of apps is greater than p , namely, some apps cannot be identified by signature matching, the corresponding flows will be misclustered. This may change network behaviors of apps since unrelated flows will be included and the traffic

features such as the number of packets and the volume of bytes will be enlarged. In this work, we use Kernel Principal Component Analysis to remit this problem, as described in the following subsection.

5.2. Network Behavior Profile Construction. After the app identification, the network behavior profiles for apps are constructed from app sessions. In AppFA, we define the app's network behavior profile as a set of chosen network traffic features as listed in Table 4. Formally, the network behavior profile can be defined as follows:

$$\text{netBehaviorProfile} := \{feature_1, \dots, feature_n\}, \quad (1)$$

$$feature_i \in \{connection_features, data_features\}.$$

Since malicious apps need to establish network connections to transmit confidential data or carry out defined attack steps [18], in this work we mainly choose *connection_features* and *data_features* for constructing apps' network behavior profiles, as shown in (1). The connection features describe how many network connections have been established, and the data features represent characteristics of packets. The whole selected features for consisting of apps' network behavior profiles are listed in Table 4. In the table, the first two lines are connection features and the rest are data features.

Furthermore, for overcoming the misclassification problem (as illustrated in Section 5.1) and distinguishing minor traffic variations from significant differences, we do not use these features directly. Instead, KPCA (Kernel Principal Component Analysis) is applied to transform basic features such as packet size and packet time interval. Basically, KPCA

Input: Data set F ; cluster number p ; must-link constraints $L_m \subseteq F \times F$; cannot-link constraints $L_c \subseteq F \times F$

Output: Flow clusters

```

(1) Let  $c_1 \dots c_p$  be the initial cluster centers
(2) for each flow  $f_i$  in  $F$  do
(3)   select the closest cluster  $c$ 
(4)   for each  $(f_i, f) \in L_m$  do
(5)     if  $f \notin c$  then
(6)       goto step (2)
(7)     end if
(8)   end for
(9)   for each  $(f_j, f) \in L_c$  do
(10)    if  $f \in c$  then
(11)      goto step (2)
(12)    end if
(13)  end for
(14)  assign  $f_i$  to the cluster  $c$ 
(15) end for
(16) for each cluster  $c_i$  do
(17)   update its center by averaging all of the flows  $f_j$  that
       have been assigned to it
(18) end for
(19) iterate between step (2) and step (18) until convergence
(20) return  $c_1 \dots c_p$ 

```

ALGORITHM 2: Constrained flow clustering algorithm.

is one approach of generalizing linear PCA into nonlinear case using the kernel method. It has been proved that KPCA has the best performance in feature extraction and is robust to noise [37]. The details of KPCA used in AppFA are described as follows.

First, Gaussian function defined in (2) is selected as the kernel function:

$$g(x, y) = \exp(-\gamma \|x - y\|^2), \quad (2)$$

where the value of γ is set to 0.001 as indicated in [37].

Then, we compute a Gram/kernel matrix K with

$$K_{i,j} = g(x_{(i)}, x_{(j)}). \quad (3)$$

Next, the kernel matrix K is centered via the following function:

$$\begin{aligned} K_{\text{centered}} &= K - 1_N K - K 1_N + 1_N K \\ &= (I - 1_N) K (I - 1_N), \end{aligned} \quad (4)$$

where 1_N is an $N \times N$ matrix with all elements equal to $1/N$ and N is the number of data points.

After that, the nonzero eigenvalues λ_i and the eigenvectors α_i of the centered kernel matrix K_{centered} are calculated as follows:

$$K_{\text{centered}} \alpha_i = \lambda_i \alpha_i. \quad (5)$$

Also, the eigenvectors α_i are normalized as

$$\alpha_i = \frac{1}{\sqrt{\lambda_i N}} \alpha_i. \quad (6)$$

Finally, we sort the eigenvectors in the descending order of corresponding eigenvalues and perform projections onto the given subset of eigenvectors. This step can be represented as follows:

$$d_j = \sum_{i=1}^N \alpha_{ji} g(x, x_i), \quad j = 1, 2, \dots, k, \quad (7)$$

where k is the dimension of the new data.

For each app, the length of its network behavior profile is $(1 + 2 * (f_o + f_i) + 2 * (k_1 + k_2))$ and f_o is the number of outgoing flows and f_i is the number of incoming flows.

5.3. Malicious App Detection. With apps' network behavior profiles, we propose to use peer group analysis [38] to detect malicious apps. The main idea of the detection method is illustrated in Figure 5. An app's network behavior profile is compared with both its historical data and the profiles of its peer group for malware detection. Comparison with the historical profiles can help us find out self-updating malicious apps [7], and comparison with the profiles of its peer group can detect repackaged malicious apps, as observed in Section 2.

For AppFA, an app's peer group is defined as the set of its *behavior-similar apps*. Apps are considered behavior-similar if they satisfy the following two conditions: ① their main functionality is similar, for example, all for mailing; ② their network behavior profiles are similar. If any of the above two conditions is not met, the peer group will be empty, and we will only compare apps' profiles with their historical data.

FIGURE 5: Illustration of the procedure of malicious app detection. The numbers in angle brackets are examples of features listed in Table 4.

FIGURE 6: Example of similar apps recommended by Google Play.

In practice, one can resort to app stores such as Google Play to find out candidates that will meet condition ①. Figure 6 shows that when viewing the details of an app, Google Play will recommend the similar ones. These similar apps are determined by several features such as category of apps, keywords in the title and description, and size of the apk (<https://www.quora.com/How-does-the-Google-Play-Store-determine-similar-apps>). Generally, these recommended apps belong to the same category and have similar functionality. Therefore, in order to determine the peer group of an app, we first use the similar apps recommended by app stores such as Google Play as candidates and later filter them by condition ②.

For condition ②, the Euclidean distance is used to measure the similarity of network behavior profiles. Suppose A is the app to be analyzed and B is the app satisfying condition ① for A . The similarity of network behavior profiles between A and B is calculated as

$$d(A, B) = \sqrt{(nBP(A) - nBP(B))(nBP(A) - nBP(B))^T}, \quad (8)$$

where nBP is the abbreviation of *netBehaviorProfile* defined in (1).

Once the similarities between A and the recommended apps are calculated by (8), the results are sorted in the order of increasing distance (i.e., decreasing similarity). The first g apps will be selected as the peer group members of A , as shown in (9). The optimal value of g can be determined by the experiments which is discussed in the next section. Note that the peer group members can be updated. For example, if one of the peer group members has been flagged as malicious, it will be removed from the peer group and a new one will be added. AppFA can also reselect the peer groups every T time interval.

$$\text{peerGroup}(A) := \{app_1, \dots, app_g\}, \quad (9)$$

where $d(A, app_i) \geq d(A, app_j)$ if $i < j$.

As illustrated in Figures 2 and 5, for an identified app, AppFA first constructs its network behavior profile and then

compares the profile with both its historical profiles and the profiles of its peer groups for malicious apps detection. Denote o as the profile of the identified app A and G as the matrix of the compared profiles (historical profiles or peer group profiles). Each column of G is one profile and the column length is l ($l = (1 + 2 * (f_o + f_i) + 2 * (k_1 + k_2))$). AppFA makes the feature vectors the same length by padding zeros.

For peer group analysis, firstly, o and G are normalized by min-max normalization procedure defined in

$$\begin{aligned} o_i &= \frac{o_i}{\max(o_i, G_{i,1}, G_{i,2}, \dots, G_{i,g})} \\ G_{i,j} &= \frac{G_{i,j}}{\max(o_i, G_{i,1}, G_{i,2}, \dots, G_{i,g})} \end{aligned} \quad (10)$$

$$i = 1, 2, \dots, l, \quad j = 1, 2, \dots, g.$$

Then the distance between o and G is calculated by Mahalanobis distance:

$$\tau = \sqrt{(o - m)^T S^{-1} (o - m)}, \quad (11)$$

where m is the weighted mean vector of profiles,

$$m_{i,1} = \sum_{j=1}^g w_{A,j} G_{i,j}, \quad 1 \leq i \leq l, \quad (12)$$

and S is the covariance matrix and it is calculated in

$$S = E \{ (G - E[G]) (G - E[G])^T \}. \quad (13)$$

In (12), $w_{A,j}$ is the weight of the j th closest peer group member of the analyzed app A . The weights of peer group members are obtained from their proximity to A . In detail, we define the proximity of the j th closest peer group member of the target app A in

$$\text{prox}_{A,j} = \exp(-d(A, j)), \quad (14)$$

where $d(A, j)$ is the Euclidean distance between app A and its j th closest peer group member defined in (8). Based on the proximity measure we defined above, the weight of the j th closest peer group member of app A is defined in

$$w_{A,j} = \frac{\text{prox}_{A,j}}{\sum_{j=1}^g \text{prox}_{A,j}} \quad (15)$$

Finally, the state of apps is judged as follows:

$$\begin{aligned} \tau &> t_s, & \text{malicious app} \\ \tau &\leq t_s, & \text{normal app} \end{aligned} \quad (16)$$

where t_s is the threshold and it can be set to different values by network administrators based on the actual network condition. We evaluated different values of t_s in our experiment.

FIGURE 7: Experimental setup for the detection of mobile malicious apps.

6. Evaluation

6.1. Data Collection

Experimental Setup. The experimental setup used for AppFA is shown in Figure 7. We have implemented a prototype of AppFA with the help of *ourmon* [39] and CCCG [40]. Ourmon is an open-source network monitoring and anomaly detection system and CCCG is a general framework for constrained clustering. A Ubuntu 16.04 computer has been configured as the access point. For packet capturing, the smart phone is connected to the Internet by WIFI and network traffic is collected at the access point by *tcpdump* (<http://www.tcpdump.org/>). Each pcap file is fixed up to 100 MB. When packet capture is complete, the TCP and UDP flows are split by *SplitCap* (<https://www.netresec.com/?page=SplitCap>) from pcap files. For apps' traffic clustering and network behavior profile construction, we use *tshark* (<https://www.wireshark.org/docs/man-pages/tshark.html>) to extract IP address, packet sizes, and packet interval times from network flows. After that, a Python program is written to calculate the statistical features as listed in Tables 3 and 4. The KPCA transformation is accomplished with the help of scikit-learn (<http://scikit-learn.org/>). The data analysis is also completed in the Ubuntu 16.04 computer (with 4 GB memory and Pentium Dual-Core CPU T4500).

Traffic Generation. We use the public MalGenome [19] dataset in our experiments. Since we mainly focus on repackaging and updating malwares, 93 typical information collection malwares, including repackaging and updating attack types, are selected from MalGenome to test the detection rate of AppFA (the malicious apps dataset was downloaded from <http://www.malgenomeproject.org/> in 2015. These selected malwares are the ones run without any errors). We have installed all these malicious apps in HUAWEI Honor 8 and a MI Note 4 phones and run these apps one by one 50 times to collect network traffic. In detail, these malwares are analyzed and run by GroddDroid [41] to make sure that malicious codes will be triggered. We also use GroddDroid to run other malwares besides the selected ones, and their traffic will be mainly used in the local detection, as described in

Section 6.3. Particularly, the background traffic including *weather forecasting*, *email checking*, *QQ*, and *Webchat* (*QQ* and *Webchat* are the most famous apps in China) is allowed in our experiments. The ground truth of the originator of network traffic is determined by *Packet Capture* (<https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture&hl=zh>) to test the accuracy of app traffic clustering.

For testing the false positive rates of AppFA, we rely on the *GooglePlayAppsCrawler.py* project to identify the 100 most popular free apps. These 100 free apps have also been run one by one 50 times. While different from malwares, the benign apps are run by droidbot [42] to make sure to generate enough traffic. With these captured traffic, we take up all nonzero packets for app traffic clustering and network behavior profile construction. Namely, in our experiments, the value of n in *Packet filter* model is set to -1 .

The data used in the experiments is summarized in Table 5. TrafficSet 1 consists of network traffic generated by the selected 93 malwares, and TrafficSet 2 consists of network traffic of benign apps. TrafficSet 3 is the traffic of all malwares; thus it includes TrafficSet 1. TrafficSet 3 is mainly used for malicious apps detection in local networks.

6.2. Experimental Results

Accuracy of Traffic Clustering. TrafficSet 1 is used for testing the accuracy of app traffic clustering and *Packet Capture* is exploited to get the ground truth of the originator of network traffic. Recall that, in Algorithm 1, app traffic clustering must be started with signature seeds. In order to obtain apps' signature seeds, we randomly choose several HTTP flows for each app and extract the key-value pairs in their HTTP headers as signature seeds, as described in Section 4. The value of cluster number p in Algorithm 2 is set to 97 ($97 = 93 + 4$), since there are 93 malwares and 4 background apps, that is, *weather forecasting*, *email checking*, *QQ*, and *Webchat*. The experimental results are shown in Table 6.

As shown in Table 6, for each app, when 5 HTTP flows are chosen to generate signature seeds, only 601 flows (in all 223,220 flows) are misclustered. This proves the efficiencies of our proposed method for app traffic clustering.

Experimental Results of Malicious App Detection. Similar to previous work, in order to measure the effectiveness of malicious apps detection, accuracy (detection rate) and false positive rate metrics are defined in (17) and (18), where TP, FN, FP, and TN stand for true positive, false negative, false positive, and true negative, respectively.

$$\text{Accuracy} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (17)$$

$$\text{False positive rate} = \frac{\text{FP}}{\text{FP} + \text{TN}}. \quad (18)$$

We first examine the detection rates and false positive rates of AppFA with different values of t_s and k_i ($i = 1, 2$). For the selected malicious apps, the apps with the same functionality (satisfying condition ①) are determined by Google Play. We first analyze the malicious apps' functionalities manually and choose a keyword for each app. Then

FIGURE 8: Detection rates of AppFA with different values of t_s and k_i ($i = 1, 2$).

we search the keyword in Google Play and use the returned apps as candidates. After that, the peer group is determined by (8) and (9). In our experiments, the largest value of member count g is set to 10. Note that t_s is the threshold defined in (16) and k_i is the number of KPCA transformations defined in Table 4. We set the member count g in peer group analysis to 5; namely, for each tested app, its top 5 similar apps are selected and analyzed. The detection rate is shown in Figure 8, and the false positive rate is depicted in Figure 9. As illustrated in Figures 8 and 9, the first 4 KPCA transformations of packet sizes and intervals are good enough for constructing network behavior profile. When $k_1 = k_2 = 4$ and $t_s > 1.2$, the detection rate is higher than 90% and the false positive rate is lower than 0.4%. The detection rate is as high as 97% when $t_s = 2$. The experimental results demonstrate the effectiveness of our proposed approach.

We then test AppFA with different value of g . The values of t_s and k_i ($i = 1, 2$) are set to 2 and 4, respectively. The experimental results are shown in Figures 10 and 11. As illustrated in these figures, when g increases, the detection rates and the false positive rates are slightly changed. This can be explained as the Mahalanobis distance τ defined in (11) is calculated by considering the weighted mean vector of profiles. So the most similar app has the largest weight, and the added peers will have smaller weights and thus provide little contribution to the final detection. As observed in our experiments, $g = 5$ is suitable for practical deployment.

Further, AppFA with different value of cluster number p is also tested. Note that the actual cluster number is 97 for TrafficSet 1. In order to evaluate how app traffic clustering impacts the detection of repackaged malware, we slightly change the value of p and the experiments results are as shown in Figure 12. In the experiments, we set $k_1 = k_2 = 4$, $t_s = 2$, and $g = 5$. As shown in Figure 12, the cluster number indeed impacts the detection rate of AppFA. Particularly, when $p = 99$, the detection rate is only 88%; namely, only 82

TABLE 5: Data summary.

Traffic set	Apps	Quantity	Number of collected flows (TCP and UPD)	Amount of data (bytes)
TrafficSet 1	Selected malicious apps (for testing accuracy of traffic clustering and detection rates)	93	223,220	53.8 G
TrafficSet 2	Benign apps (for testing false positive rates of AppFA)	100	305,103	64.0 G
TrafficSet 3	All malicious apps (for malicious apps detection in local networks)	1260	≈3.1 million	187.2 G

TABLE 6: Experimental results of traffic clustering.

Number of HTTP flows for generating signature seeds	Number of flows correctly clustered	Number of misclustered flows
2	201,357	21,863
3	218,946	4,274
4	221,384	1,836
5	222,619	601

FIGURE 9: False positive rates of AppFA with different values of t_s and k_i ($i = 1, 2$).

malicious apps are correctly detected. However, when $p < 97$ ($p = 95$ and 96 in Figure 12), the detection rates are almost the same, namely, 97%. Therefore, AppFA will be competent with a few unknown apps when carrying out app traffic clustering.

Finally, we evaluate AppFA with the remaining malicious apps besides the selected 93 samples. The rest of the malicious apps contain repackaged and other types of malwares. These malwares are found to have some errors when run on our phones. However, they also produced considerable network traffic. We test AppFA with these apps for examining the general capability of AppFA. The parameters are set up as follows: $k_1 = k_2 = 4$, $t_s = 2$, and $g = 5$. The experimental results are shown in Table 7. The detection rate is 73.4% and the false positive rate is 3.5%. Compared to the results

FIGURE 10: Detection rates of AppFA with different values of g . FIGURE 11: False positive rates of AppFA with different values of g .

displayed in Figures 8, 9, 10, and 11, the detection rate is significantly reduced and the false positive rate is increased. The possible reasons will be discussed in Section 6.4.

Note that AppFA is designed to perform nearly real-time malicious apps detection; thus efficiency is also a big concern. Table 8 presents the computational performance for major

FIGURE 12: Detection rates of AppFA with different values of p .

TABLE 7: Experimental results for the malicious apps besides the selected 93 samples.

Detection rate	False positive rate
73.4%	3.5%

TABLE 8: The computational performance.

Procedures	Time
Constrained flow clustering (Algorithm 1)	3 min
App identification (Algorithm 2)	5.2 min
Network behavior profile construction	1.7 min
Peer group analysis	45 sec

steps in terms of average running time using the experimental data. Generally, the proposed method can be deployed online to detect malicious apps. Specifically, it is fast, typically less than one minute, to exploit peer group analysis to perform malicious app detection. The app identification procedure is the most time-consuming step, because the signature matching and constrained flow clustering are carried out on each flow. Parallel computing such as cloud resources can be further used to speed the malicious apps detection and scale up to more Internet traffic data.

Detection of Malicious Repackaged Apps for Encrypted Traffic. As shown in Table 4, AppFA does not consider traffic content and mainly uses traffic statistical features to detect malicious Android apps. Thus, AppFA can deal with encrypted traffic. Since, in MalGenome, there is no app whose all connections to attack servers are encrypted, we have created a new Android repackaged malware that communicates with all malicious servers by encrypted connections. In detail, we first modify *AnserverBot*'s source code and make sure its network connections will be encrypted by TLS protocol. Then we graft it into a popular game app *air.com.aceviral.motox3*. Finally, we collect network traffic of the repackaged *air.com.aceviral.motox3* and its peer group. In the experiments,

TABLE 9: Experimental results when searching similar apps locally.

Detection rate	False positive rate
69.2%	4.7%

the selected peer group of the repackaged *air.com.aceviral.motox3* is $\{com.wordmobiles.bikeRacing, com.topfreemgames.bikeracefreeworld, com.skgames.trafficrider, com.tomico.wheeliechallenge \text{ and } mad.moto.racing.bike\}$. We repeat to detect the repackaged *air.com.aceviral.motox3* 10 times, and the detection rate is 100%. The results show that AppFA can handle encrypted network traffic.

6.3. Detection in Local Networks. In the above experiments, the peer group for each app is determined with the help of app store such as the Google Play. That means AppFA has to access the Internet when performing malicious apps detection. In fact, AppFA can also be enhanced to choose the apps' peer groups locally, namely, from the set of already identified apps. Note that AppFA uses signature matching to identify apps on the network and the identified ones can be treated as a mini app store. By this, AppFA can work on local networks. To find out similar apps from the already identified ones, an efficient method based on information retrieval technologies is proposed and is shown in Algorithm 3.

The basic idea of Algorithm 3 is that similar apps may have similar functionalities, and the context (keywords) may be similar. Therefore, we use web-page searching technologies to match similar app. The *Matching* function in line (4) can be realized by TF-IDF (Term Frequency-Inverse Document Frequency) [43]. Again, peer groups can be chose by (8) with the results returned by Algorithm 3.

We have implemented the local similar app searching and evaluated AppFA's performance in local networks. All apps, including all malicious and benign apps, are considered as identified apps and Algorithm 3 is carried out to find out similar apps. When $k_1 = k_2 = 4$, $t_s = 2$, and $g = 5$, for the 93 typical repackaged malwares, the experimental results are listed in Table 9. The detection rate is 69.2%, lower than 97% in Figure 8. The experimental results indicate that AppFA can work in local networks, and the proper peer groups do affect the final detection significantly.

6.4. Discussion. Figures 8, 9, 10, and 11 show that AppFA gets high detection rates and low false positive rates when detecting repackaged apps. However, the detection rate may become lower when it is applied to other types of malicious apps such as sending SMS without notification. This may be due to the fact that other types of mobile malwares transmit less data through network than repackaged apps. Note that the detection features used in AppFA are mainly related to the packets sizes and the number of flows. So the detection rate may be declined if the apps' network behaviors are slightly changed.

Compared to the latest work done in [6], AppFA has a slightly lower detection rate (the detection rate is 95–99.9% as reported in [6]). However, Garg et al.'s method [6] considered only 18 malware apps and 14 genuine apps and assumed


```

(1) extract all plaintext keywords from HTTP headers of the
    tested app  $A$ . Denote the set of keywords as  $W(A)$ .
(2) extract all plaintext keywords from HTTP headers of other
    identified apps. Denote the set of keywords of identified
    app  $i$  as  $W(i)$ ,  $i = 0, 1, \dots, n-1$ .
(3) while  $0 \leq n$  do
(4)    $Scores[i].value = Matching(W(A), W(i))$ 
(5)    $Scores[i].app = i$ 
(6)    $i++$ 
(7) end while
(8) sorting  $Scores[]$  with  $Scores[i].value$  by decreasing order
(9) return the first  $mScores[i].app$ ,  $i = 0, 1, \dots, m-1$ .

```

ALGORITHM 3: Searching similar apps locally.

TABLE 10: Experimental results when the fact that what traffic is generated by what app is exactly known. In the experiments, $k_1 = k_2 = 4$, $t_s = 2$, and $g = 5$.

	Selected malicious apps	Remaining malicious apps
Detection rate	98.3%	85%
False positive rate	0.03%	2.7%

that the traffic generated by apps was already known (their data were collected on the mobile device, so their method is actually a kind of client-side approach). In our work, much larger samples are taken into account and constrained clustering is exploited to determine to what app the flows belong. The accuracy of clustering is also a factor affecting the detection rate. To confirm this, we further take the same assumption as Garg et al.'s method [6]; namely, we assume the accuracy of Algorithm 1 is 100% and know exactly what traffic is generated by what app (in our experiments, we use app *Packet Capture* to obtain perfect ground truth of what flows came from what app). At this time, the experimental results are given in Table 10. Apparently, the detection rates are much improved.

In practice, one can deploy both our method and other methods such as [6, 15] simultaneously. As indicated by our experiments, AppFA is efficient in detecting repackaged and self-updating malicious apps. In fact, [15] is suitable for HTTP analysis and [6] can detect C&C communication efficiently. So our work can be a complement of existing work on malicious app detection.

In this work, AppFA mainly uses the statistical features (refer to Table 4) of network traffic to detect Android malicious repackaged applications. Therefore, attackers (repackaged apps) may change the characteristics of their traffic to remain undetectable. But it is quite difficult in practice. As illustrated in Section 2, repackaged apps usually introduce additional network traffic; thus the attackers must remove some normal network connections to keep network behaviors the same to evade detection. However, the removal of

normal network connections will impact the functionalities of apps and may cause errors. This gives additional clues to detect repackaged malware. Meanwhile, the disappearance of some network connections may be abnormal as well. Therefore, our proposed method is hard to evade.

7. Conclusion

In this paper, we propose a novel approach, AppFA, to detect malicious apps at the network level. In AppFA, apps are first identified from network traffic by signature matching and constrained clustering. Then Kernel Principal Component Analysis is employed to construct app network behavior profile and distinguish minor traffic variations from significant differences. At last, we take advantage of peer group analysis to detect malicious apps to avoid time-consuming offline model training. Notably, AppFA does not need to install programs or modify operating systems to collect feature information. Thus, it is very convenient to be used and the cost is low. The experimental results show that AppFA can detect Android repackaged malware with the detection rate higher than 90% and a false positive rate lower than 0.4%.

The apps' network behaviors may be significantly changed by version update and thus cause false positives. This needs to be further investigated. In the future work, AppFA will be extended to include more network traffic features and detect more types of malicious apps. We will also make AppFA an open-source project to facilitate further researches of malicious app detection.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grants 61702282 and 61502250, Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant 17KJB520023, NUPTSF under Grant NY217143, and Nanjing Forestry University (GXL016, CX2016026).

References

- [1] *Android statistics: Number of android applications*, September 2017, <http://www.appbrain.com/stats/number-of-android-apps>.
- [2] A. Mylonas, M. Theoharidou, and D. Gritzalis, "Assessing Privacy Risks in Android: A User-Centric Approach," in *Risk Assessment and Risk-Driven Testing*, Lecture Notes in Computer Science, pp. 21–37, Springer International Publishing, Cham, 2014.
- [3] Does., *Does your mobile anti-virus app protect or infect you? the truth behind du antivirus security*, September 2017, <https://research.checkpoint.com/mobile-anti-virus-app-protect-infect-truth-behind-du-antivirus-security/>.
- [4] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11) held in Association with the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pp. 3–14, October 2011.
- [5] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, Article ID 3017427, 2017.
- [6] S. Garg, S. K. Peddoju, and A. K. Sarje, "Network-based detection of Android malicious apps," *International Journal of Information Security*, pp. 1–16, 2016.
- [7] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, vol. 43, pp. 1–18, 2014.
- [8] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "PBMDs: a behavior-based malware detection system for cellphone devices," in *Proceedings of the 3rd ACM Conference on Wireless Network Security (WiSec '10)*, pp. 37–48, Hoboken, NJ, USA, March 2010.
- [9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11) Held in Association with the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pp. 15–25, October 2011.
- [10] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "MADAM: a multi-level anomaly detector for android malware," in *Computer Network Security*, vol. 7531 of *Lecture Notes in Computer Science*, pp. 240–253, Springer, Berlin, Germany, 2012.
- [11] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 611–622, 2013.
- [12] S. Wang, Z. Chen, X. Li, L. Wang, K. Ji, and C. Zhao, "Android Malware Clustering Analysis on Network-Level Behavior," in *Intelligent Computing Theories and Application*, vol. 10361 of *Lecture Notes in Computer Science*, pp. 796–807, Springer International Publishing, Cham, Switzerland, 2017.
- [13] A. S. Shamili, C. Bauckhage, and T. Alpcan, "Malware detection on mobile devices using distributed machine learning," in *Proceedings of the 20th International Conference on Pattern Recognition (ICPR '10)*, pp. 4348–4351, August 2010.
- [14] Y. Wang, J. Wei, and K. Vangury, "Bring your own device security issues and challenges," in *Proceedings of the 2014 IEEE 11th Consumer Communications and Networking Conference, CCNC 2014*, pp. 80–85, USA, January 2014.
- [15] P. S. Chen, S.-C. Lin, and C.-H. Sun, "Simple and effective method for detecting abnormal internet behaviors of mobile devices," *Information Sciences*, vol. 321, Article ID 11536, pp. 193–204, 2015.
- [16] *Global mobile os market share in sales to end users from 1st quarter 2009 to 1st quarter 2017*, 2017, <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operatingsystems/>.
- [17] "It threat evolution q3 2017. statistics," November 2017, <https://securelist.com/it-threat-evolution-q3-2017-statistics/83131/>.
- [18] L. Li, D. Li, T. F. Bissyande et al., "Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.
- [19] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, Calif, USA, May 2012.
- [20] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur, and V. Ganmoor, "AndroSimilar: Robust signature for detecting variants of Android malware," *Journal of Information Security and Applications*, vol. 22, pp. 66–80, 2015.
- [21] Y. Zhou and X. Jiang, *An analysis of the anserverbot trojan*, 2011, http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf.
- [22] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.
- [23] P. Yan and Z. Yan, "A survey on dynamic mobile malware detection," *Software Quality Journal*, pp. 1–29, 2017.
- [24] S. Dai, Y. Liu, T. Wang, T. Wei, and W. Zou, "Behavior-based malware detection on mobile phone," in *Proceedings of the 2010 6th International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2010*, China, September 2010.
- [25] D. Damopoulos, G. Kambourakis, S. Gritzalis, and S. O. Park, "Exposing mobile malware from the inside (or what is your mobile app really doing?)," *Peer-to-Peer Networking and Applications*, vol. 7, no. 4, pp. 687–697, 2014.
- [26] D. Damopoulos, G. Kambourakis, and G. Portokalidis, "The best of both worlds. A framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones," in *Proceedings of the 7th European Workshop on System Security, EuroSec 2014*, Netherlands, April 2014.
- [27] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC '11)*, pp. 329–344, November 2011.
- [28] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "NetworkProfiler: Towards automatic fingerprinting of Android apps," in *Proceedings of the 32nd IEEE Conference on Computer Communications, IEEE INFOCOM 2013*, pp. 809–817, Italy, April 2013.
- [29] Q. Xu, T. Andrews, Y. Liao et al., "Flowr: A self-learning system for classifying mobile application traffic," in *Proceedings of the 2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2014*, pp. 569–570, usa, June 2014.
- [30] Q. Xu, Y. Liao, S. Miskovic et al., "Automatic generation of mobile app signatures from traffic observations," in *Proceedings*

- of the 34th IEEE Annual Conference on Computer Communications and Networks, IEEE INFOCOM 2015, pp. 1481–1489, Hong Kong, May 2015.
- [31] J. Sun, L. She, H. Chen et al., “Automatically identifying apps in mobile traffic,” *Concurrency Computation*, vol. 28, no. 14, pp. 3927–3941, 2016.
 - [32] G. He, B. Xu, and H. Zhu, “Identifying Mobile Applications for Encrypted Network Traffic,” in *Proceedings of the 5th International Conference on Advanced Cloud and Big Data, CBD 2017*, pp. 279–284, China, August 2017.
 - [33] G. He, M. Yang, X. Gu, J. Luo, and Y. Ma, “A novel active website fingerprinting attack against Tor anonymous system,” in *Proceedings of the 2014 18th IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD 2014*, pp. 112–117, Taiwan, May 2014.
 - [34] G. He, M. Yang, J. Luo, and X. Gu, “A novel application classification attack against Tor,” *Concurrency and Computation: Practice and Experience*, vol. 27, no. 18, pp. 5640–5661, 2015.
 - [35] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar, “I know why you went to the clinic: Risks and realization of HTTPS traffic analysis,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 8555, pp. 143–163, 2014.
 - [36] Y. Wang, Y. Xiang, J. Zhang, W. Zhou, G. Wei, and L. T. Yang, “Internet traffic classification using constrained clustering,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 2932–2943, 2014.
 - [37] L. J. Cao, K. S. Chua, W. K. Chong, H. P. Lee, and Q. M. Gu, “A comparison of PCA, KPCA and ICA for dimensionality reduction in support vector machine,” *Neurocomputing*, vol. 55, no. 1–2, pp. 321–336, 2003.
 - [38] Y. Kim and S. Y. Sohn, “Stock fraud detection using peer group analysis,” *Expert Systems with Applications*, vol. 39, no. 10, pp. 8986–8992, 2012.
 - [39] “Ourmon network monitoring and anomaly detection system,” April 2013, <http://ourmon.sourceforge.net>.
 - [40] B. Babaki, T. Guns, and S. Nijssen, “Constrained Clustering Using Column Generation,” in *Integration of AI and OR Techniques in Constraint Programming*, vol. 8451 of *Lecture Notes in Computer Science*, pp. 438–454, Springer International Publishing, Cham, 2014.
 - [41] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong, “GroddDroid: A gorilla for triggering malicious behaviors,” in *Proceedings of the 10th International Conference on Malicious and Unwanted Software, MALWARE 2015*, pp. 119–127, USA, October 2015.
 - [42] Y. Li, Z. Yang, Y. Guo, and X. Chen, “DroidBot: A lightweight UI-guided test input generator for android,” in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion, ICSE-C 2017*, pp. 23–26, Argentina, May 2017.
 - [43] A. Aizawa, “An information-theoretic perspective of tf-idf measures,” *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.