Hindawi
Security and Communication Networks
Volume 2020, Article ID 8630748, 19 pages
https://doi.org/10.1155/2020/8630748

Research Article

On-Device Detection of Repackaged Android Malware via Traffic Clustering

Gaofeng He,[1,2] Bingfeng Xu,[3] Lu Zhang,[4] and Haiting Zhu[1]

[1]*College of Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing 210003, China*
[2]*Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education, Nanjing, China*
[3]*College of Information Science and Technology, Nanjing Forestry University, Nanjing 210037, China*
[4]*College of Information Engineering, Nanjing University of Finance & Economics, Nanjing 210046, China*

Correspondence should be addressed to Haiting Zhu; htzhu@njupt.edu.cn

Received 3 July 2019; Revised 14 March 2020; Accepted 18 May 2020; Published 31 May 2020

Academic Editor: Pino Caballero-Gil

Malware has become a significant problem on the Android platform. To defend against Android malware, researchers have proposed several on-device detection methods. Typically, these on-device detection methods are composed of two steps: (i) extracting the apps' behavior features from the mobile devices and (ii) sending the extracted features to remote servers (such as a cloud platform) for analysis. By monitoring the behaviors of the apps that are running on mobile devices, available methods can detect suspicious applications (simply, *apps*) accurately. However, mobile devices are typically resource limited. The feature extraction and massive data transmission might consume substantial power and CPU resources; thus, the performance of mobile devices will be degraded. To address this issue, we propose a novel method for detecting Android malware by clustering apps' traffic at the edge computing nodes. First, a new integrated architecture of the cloud, edge, and mobile devices for Android malware detection is presented. Then, for repackaged Android malware, the network traffic content and statistics are extracted at the edge as detection features. Finally, in the cloud, similarities between apps are calculated, and the similarity values are automatically clustered to separate the original apps and the malware. The experimental results demonstrate that the proposed method can detect repackaged Android malware with high precision and with a minimal impact on the performance of mobile devices.

1. Introduction

Recently, Android platforms (e.g., smartphone, smartwatch, and tablet) have become increasingly popular worldwide. According to Statista, Android is accounted for more than 74% of the global mobile OS market as of December 2019 [1]. One of the most important reasons for Android's popularity is that mobile users can conveniently download various types of apps from app stores [2]. For example, an official Android market, namely, the Google Play store, already had more than 2.9 million apps in January 2020 [3]. These feature-rich apps make the Android platform attractive and vibrant.

Android apps, i.e., Android Package Kit (APK) files, are archives in ZIP format, which include developer bytecodes, resource files, and a manifest file. Unfortunately, in contrast to traditional executable software, Android apps are easy to repackage. With open-source tools such as apktool (https://github.com/iBotPeaches/Apktool) and jadx (https://github.com/skylot/jadx), one can easily add code or modify resource files to realize various objectives. For instance, malware writers can graft ad code on popular apps to make money or graft malicious code to steal user privacy information (for example, the Xavir malware has infected over 800 android apps. These repackaged apps are often self-signed and do not require mobile devices to be rooted).

According to [4–6], most Android malware programs are repackaged apps, and the majority of new Android malware samples are polymorphic variants of known malware. Repackaged Android malware is becoming a substantial threat to Android security.

Researchers have extensively investigated the detection of (repackaged) Android malware, and several methods have been proposed and evaluated [7–14]. Previous works can be generally divided into *off-device* and *on-device* detection. Off-device detection is typically adopted by security analyzers who analyze APK files at a dedicated analysis server. The analyzers can check the APKs with signatures or dynamically run them in a real or virtual execution environment to record their behaviors [15]. If malicious code or behaviors are identified, the apps will be judged as malware. Off-device detection can also be conducted via network monitoring [14]. We will refer to off-device detection methods of this type as *network-based methods* in the following.

On-device detection is implemented partially or completely locally, namely, at users' mobile devices. The recognition of malware from the installed apps is attempted. Technically, on-device detection also identifies malware by analyzing the behaviors of the apps. Compared to off-device detection, on-device detection can detect malware that is running on mobile users' devices and identify more sophisticated malicious apps, such as those which are triggered by specified events. For example, by observing the deviations of app network behaviors at the mobile device, one can detect self-updating malware that may be triggered by a specified geo-location, device type, or OS version [8]. Nevertheless, on-device detection is more challenging due to the limited resources of mobile devices.

Typically, several steps should be conducted to detect malicious Android apps at mobile devices. First, detection features such as the user's operating behaviors, API usages, and application network behaviors should be defined and extracted. Then, based on the extracted features, detection models are constructed, and, finally, new apps are compared with the constructed models for malware detection. Depending on where these steps are performed, current approaches can be categorized into two main groups: *client-side* detection and *server-side* detection [14]. For client-side detection, these steps are all conducted at the mobile devices, while for server-side detection, the main steps (model construction and malware detection) are conducted on remote servers.

In practice, the application of the client-side approaches could be restricted because they consume substantial amounts of resources and power. According to reference [8], the storage of 10 Android malware detection models requires 7272 kB $\pm$ 15 memory consumption and the learning of these models requires 13% $\pm$ 1.5 CPU consumption. Worse, the more models that are trained and saved, the more resources and power are consumed. Therefore, the server-side methods are typically adopted [16]. However, the resource and power consumptions of the server-side methods remain high (e.g., 7% performance overhead and approximately 3% battery overhead [17]). This is because the detection features are still mainly collected and processed at the mobile devices [10, 17].

In this paper, we propose a novel method for detecting repackaged Android malware based on clustering apps' traffic in the edge computing platform. Although Android malware can use emulator/sandbox detection [18] and packing [19, 20] to impede dynamic and static analysis, it is difficult for them to hide the traffic to be inspected. The basic strategy of our method is that the feature extraction and preprocessing can be offloaded to the edge. Thus, the workloads of mobile devices are reduced. The designed architecture for Android malware detection consists of three components: mobile devices, edge servers, and the cloud. The mobile devices are used normally, and an additional *edge-client* app will be installed and run in the background. The edge-client app operates identically to an Android VPN app such as *Packet Capture* (https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture&hl=en US), except it only records the flow metainformation, such as the flow starting time and flow protocol, and it does not need to save and manipulate the packet contents. Hence, it is lightweight and imposes a negligible burden on the mobile devices.

The edge servers collect mobile network traffic and extract detection features from the traffic. The traffic preprocessing that is conducted by the edge servers not only reduces the amount of data to be sent but also protects the mobile users' privacy because the cloud can only observe the coarse-grained features. The extracted network traffic features are sent to the cloud for final processing. In the cloud, similarities between apps are calculated based on the received features. After that these similarity values are automatically clustered to separate original apps and repackaged malware. We conducted an extensive set of experiments, and the results demonstrated that the average detection accuracy was 96.9%.

Our main contributions are as follows:

(i) A lightweight and efficient framework that is based on edge computing for the detection of Android malware is proposed. Compared to the typical server-side approaches, the proposed framework will have minimal impact on the performances of the mobile devices. In addition, compared to the network-based methods, the detection accuracy can be increased, and the privacy of mobile users can be better protected.

(ii) A traffic clustering method for the identification of repackaged Android malware is proposed. We cluster the network traffic that is generated by the same apps to determine which one have been repackaged. First, we use the plaintext contents and flow statistical features to calculate the similarities between the apps. Then, the similarity values are clustered to separate the original apps and the repackaged malware automatically. Via clustering, it is unnecessary to model the apps' network behaviors or extract matching signatures for malware detection in advance. Hence, this approach will be more efficient in practice.

(iii) Extensive experiments are conducted on public datasets.

The remainder of this paper is organized as follows. The preliminaries of Android malware and edge computing are introduced in Section 2. In Section 3, we discuss relevant related work. Section 4 describes the new integration architecture of the cloud, edge, and mobile devices for Android malware detection. The methodology is presented in Section 5. An experimental evaluation of our method in terms of accuracy and performance is presented in Section 6. We discuss the limitations of the proposed method in Section 7. Finally, in Section 8, we present the conclusions of the paper and discuss potential future work.

2. Preliminaries

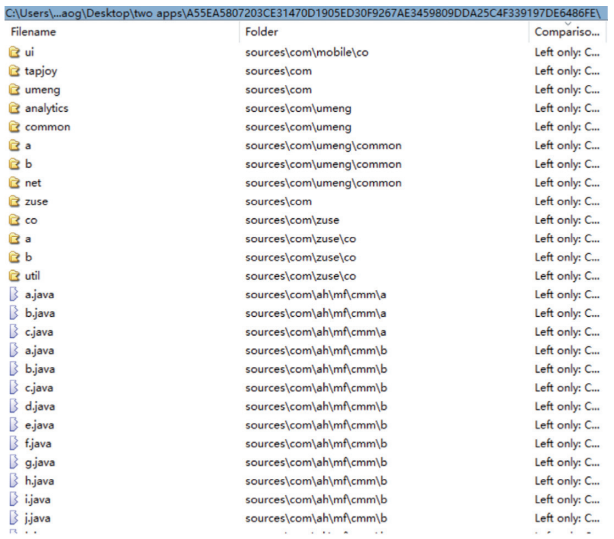
In this section, we introduce important concepts regarding Android malware and edge computing, which form the basis of this work.

2.1. Android Malware.

2.1. Android Malware. As discussed in Section 1, Android apps (APKs) are easy to repackage, which is convenient for attackers who would like to create Android malware. In fact, most Android malware is produced by repackaging benign apps, especially favorite apps to ensure a wide diffusion of the malicious code [4]. As evidence, MalGenome [21], which is a reference dataset in the Android security community, 80% of the malicious samples were built via repackaging other apps. Repackaged apps can be distributed through third-party markets, which do not typically require screening or integrity evaluation of the uploaded apps [22]. For example, many of the apps in F-Droid are found to be repackaged by the same developer [23]. Consequently, we detect mainly malicious repackaged Android apps in this study.

Repackaged Android malware can be efficiently detected by analyzing its network traffic [7, 14]. This is because the repackaged apps are typically functionalized to interact with remote servers to receive commands or return sensitive data, and their network behaviors will differ from the original network behaviors. To better explain this, we decompile two apps, namely, 00575DE650F78413C91E8E613ADF909981 2F325AA9CF36C72E81C79B230F58F4 and A55EA58072 03CE31470D1905ED30F9267AE3459809DDA25C4F3391 97DE6486FE, and compare their corresponding folders to identify the newly added files. The added files are further analyzed (by reading the Java code) to investigate their network behaviors. In this example, the app's name is the SHA256 value of the APK file, and 00575∗ corresponds to the original app and A55EA∗ is the repackaged app. We used *Jadx to* decompile APKs and the file comparison is conducted with *WinMerge* (http://winmerge.org/). Figure 1 presents the comparison results.

As shown in Figure 1, there are a total of 250 additional Java code files, and these code files are distributed in the sources/com folder. We read these added Java codes carefully and determine that Android API *HttpPost* is invoked 8


C:\Users\...aog\Desktop\two apps\A55EA5807203CE31470D1905ED30F9267AE3459809DDA25C4F339197DE6486FE\

Filename	Folder	Compariso...
ui	sources\com\mobile\co	Left only: C...
tapjoy	sources\com	Left only: C...
umeng	sources\com	Left only: C...
analytics	sources\com\umeng	Left only: C...
common	sources\com\umeng	Left only: C...
a	sources\com\umeng\common	Left only: C...
b	sources\com\umeng\common	Left only: C...
net	sources\com\umeng\common	Left only: C...
zuse	sources\com	Left only: C...
co	sources\com\zuse	Left only: C...
a	sources\com\zuse\co	Left only: C...
b	sources\com\zuse\co	Left only: C...
util	sources\com\zuse\co	Left only: C...
a.java	sources\com\ah\mf\cmm\a	Left only: C...
b.java	sources\com\ah\mf\cmm\a	Left only: C...
c.java	sources\com\ah\mf\cmm\a	Left only: C...
a.java	sources\com\ah\mf\cmm\b	Left only: C...
b.java	sources\com\ah\mf\cmm\b	Left only: C...
c.java	sources\com\ah\mf\cmm\b	Left only: C...
d.java	sources\com\ah\mf\cmm\b	Left only: C...
e.java	sources\com\ah\mf\cmm\b	Left only: C...
f.java	sources\com\ah\mf\cmm\b	Left only: C...
g.java	sources\com\ah\mf\cmm\b	Left only: C...
h.java	sources\com\ah\mf\cmm\b	Left only: C...
i.java	sources\com\ah\mf\cmm\b	Left only: C...
j.java	sources\com\ah\mf\cmm\b	Left only: C...

Figure 1: Folders and files that were only present in repackaged app A55EA*, as indicated by *left only* in the third column.

times and *HttpGet* is invoked 5 times. Hence, there are at least 13 flows that would never be produced by the original app, namely, 00575*. Therefore, the original app can be efficiently distinguished from the repackaged version by comparing their network traffic. Inspired by these observations, we propose clustering the mobile apps' traffic to separate original apps and repackaged malware automatically.

2.2. Edge Computing. Edge computing can be defined in terms of the technologies that enable computations to be performed at the edge of the network, on downstream data on behalf of cloud services, and on upstream data on behalf of Internet of Things (IoT) services [24]. Perhaps the most important concept of edge computing is that the edge can consist of any computing and network resources along the path between mobile/IoT devices and cloud data centers. For example, the edge could be a phone between a smart bracelet and the cloud or a wireless router between the smart TV and the cloud if the phone or the wireless router performs computations on behalf of the smart devices or the cloud. In these examples, the computations could be data anonymization for privacy protection or content caching for performance improvement. Figure 2 illustrates the paradigm of edge computing.

As illustrated in Figure 2, cloud computing-like capabilities, i.e., edge servers are deployed at the edge of the network for data (pre) processing and forwarding. In edge computing, the mobile/IoT devices could send computation tasks to the edge servers, namely, the devices can distribute code to the edge to be executed; hence, the mobile/IoT devices can be designed to conduct complicated operations, such as machine learning or encryption. Similarly, the cloud may push services to the edge servers, and the edge can be deployed and updated online. In most cases, the devices should be connected to the edge and the cloud

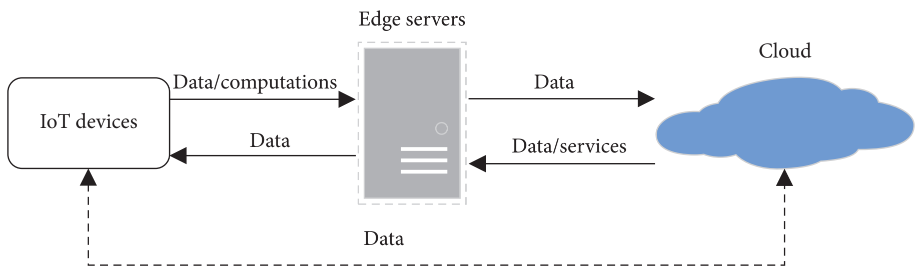


Figure 2: Paradigm of edge computing.

simultaneously, as represented by the solid and dotted lines in Figure 2. This is because, not all data must be preprocessed by the edge servers, and, more importantly, the robustness of the entire system can be improved. If the edge breaks down, the cloud can continue to provide services. In this study, we obey the edge computing paradigm that is illustrated in Figure 2 in the design of the Android malware detection system.

3. Related Work

Repackaged Android apps are one of the major sources of mobile malware, and extensive studies have been conducted on Android malware detection. In this section, we will review the most closely related studies. The typical studies of edge computing are also reviewed to provide more detailed background information.

3.1. Off-Device Detection. Repackaged Android apps and malware can be efficiently detected via code comparison of APKs. DroidMoss [25] focuses on app code and conducts pairwise similarity comparisons for the detection of repackaged apps in Android markets. The detection results demonstrate that 5% to 13% of apps that are hosted on third-party marketplaces were repackaged. However, the pairwise comparison approach cannot be scaled up because millions of Android apps are now available in markets. To resolve this issue, Shao et al. [26] proposed ResDroid for the detection of repackaged apps in Android markets via a divide-and-conquer strategy. First, they built a *k-d* tree and inserted clusters that contained one or more apps into the *k-d* tree. Then, the apps were compared with their neighbors in the *k-d* tree, and the number of comparisons was reduced. Assuming that the apps from the official market were benign, Massvet [27] downloaded all apps from Android markets in advance and constructed *v-core* (a geometric center of a view graph) and *m-core* (mapping the features of a Java method to an index) databases for repackaged Android malware detection. Both databases were used to vet new apps that were submitted to the market. New apps would be reported as repackaged malware if their *v-cores* or *m-cores* were similar to records in the databases and the code was not legitimately reused.

In addition to code comparison, behavior-based and UI-based detection methods are also emerging. DroidChain [28] constructs a behavior chain model that is composed of typical behavior processes of Android apps for the detection of malware. In DroidChain, the typical behaviors include privacy leakage, SMS financial charging, malware installation, and privilege escalation. According to Tian et al. [29], repackaged malware was difficult to detect partly because of their behavioral similarities to benign apps. To overcome this problem, they proposed a new repackaged Android malware detection technique that was based on code heterogeneity analysis. The code structure of an app was partitioned into multiple dependence-based regions, and each region was independently classified according to its behavioral features. De Lorenzo et al. [30] proposed VizMal for visualizing the execution traces of Android applications and highlighting potentially malicious behaviors. Lin et al. [31] proposed the detection of repackaged Android apps based on static UI features. They detected a total of 3,723 repackaged app pairs in the Anzhi market and 15,856 repackaged pairs in the Mi market. These results demonstrate that repackaged malware still poses a severe threat to the security of mobile users even though various types of detection methods have been proposed and adopted by Android markets [32].

Recently, several network-based approaches have been proposed for the detection of repackaged apps and malware. Arora and Peddoju [33] demonstrated the effectiveness of using network traffic to detect Android malware. Wu et al. [34] divided app HTTP traffic into 2 categories: primary module traffic and nonprimary module traffic. For primary module traffic, the HTTP flow distance algorithm and the Hungarian Method were used to calculate the traffic similarity and identify similar app pairs. CREDROID [35] identified malicious apps on the basis of Domain Name Server (DNS) queries and the data that are transmitted to remote servers. Zulkifli et al. [36] proposed a method for detecting Android malware that is based on seven network traffic features and the J48 decision tree algorithm. Chen et al. [37] introduced the imbalanced data gravitation-based classification algorithm for the classification of imbalanced data of malicious apps. He et al. [14, 38] proposed the identification of encrypted apps' flows via traffic correlation and the detection of repackaged Android apps via comparison of the network behaviors of similar apps. In these methods, the detection is conducted mostly in the router or at the network monitoring node; hence, the performances of mobile devices are not affected. However, the network behaviors of malware must be modeled in advance, which is challenging in practice.

3.2. On-Device Detection. The detection of Android malware can also be conducted at users' mobile devices [39]. Alzaylaee et al. [40] extracted features from real devices and used a deep learning system to detect malicious Android apps. Shamili et al. [41] proposed a distributed Support Vector Machine (SVM) algorithm for the detection of Android malware on a network of mobile devices. Their detection features include short duration calls, medium duration calls, and the number of outgoing SMS (Short Message Service), among others. The experimental results demonstrated that the average computation time per client during the training phase is nearly 10 seconds. Zhao et al. [42] implemented a malware detection framework, namely, RobotDroid, using an SVM active learning algorithm. RobotDroid monitors all the running apps to identify their running characteristics. These characteristics are later input into learning modules to generate the behavioral characteristics for the classification of normal apps and malware. According to their performance evaluation, the time of accessing the GPS is increased from 80 ms to almost 120 ms when the detection system is executed. Hence, the performance of a smartphone that utilizes this framework would be significantly degraded. Shabtai et al. [8] monitored the apps' network behaviors at the mobile devices to detect repackaged Android malware, and the detection accuracy exceeds 80%; however, the performance overhead of their method is not negligible. For example, it requires 7272 kB ± 15 memory consumption and 13% ± 1.5 CPU consumption for learning 10 Android malware detection models.

To further improve the performance of on-device detection, Crowdroid [10] sends the detection features to a centralized server for processing. Crowdroid only records the system calls as the detection features, and it is lightweight. Talha et al. [43] proposed APK Auditor, which is a system that uses permissions as static analysis features for Android malware detection. APK Auditor consists of three components: (1) a signature database; (2) an Android client; and (3) a central server. The Android client sends the app to the central server for analysis. The central server extracts the permissions that are requested by the app and computes the permission malware score based on the permissions in malware. If the score exceeds a threshold value, the app is classified as malware. The results demonstrate that APK Auditor achieves 92.5% specificity; however, it lacks the benefits of dynamic analysis as it cannot detect dynamic malicious payloads.

Monet [17] also uses backend servers and a client app to detect Android malware. The client app constructs a directed graph of the app components and the system components, and it records the potentially dangerous system calls as the detection features. These features are forwarded to the backend server, which conducts further detection by applying a signature matching algorithm and returns the detection results. The experimental results demonstrate that Monet can detect Android malware with 99% accuracy. However, the performance overhead of Monet is high. As shown in [17], Monet had 8% overhead on memory and I/O benchmarks and 5.5% overhead for battery. Most recently, Arshad et al. [44] proposed SAMADroid, which is a 3-level hybrid model for the detection of Android malware. Similarly, SAMADroid has a client app and an analysis server. The client app hooks the *Strace* tool and traces the system calls that are invoked by the user app. If the user app remains running on the device, the client app continues tracing the system calls of the user app and generates a log file of the system calls. Later, the log file is sent to the SAMADroid server with the identifier of the user app. At the server, the same user app is downloaded from stores and statically analyzed. Then, the static features and the dynamic features that are extracted from the log file are embedded in a vector space. Finally, the features are provided as input to the machine learning tool for classification. Their experimental results demonstrate that random forest yields 99.07% accuracy on the static features and 82.76% on the dynamic features. The performance overhead of SAMADroid is 1.8% for memory and 0.6% for CPU. In previous work [45], we used mobile edge computing to detect malicious apps. We extend it to the edge computing paradigm and refine the clustering method in this study.

3.3. Edge Computing Research. The vision and challenges of edge computing are introduced in the literature [24]. The basic strategy of edge computing is to deploy cloud computing at the edge of the network to improve the performance and create new applications for IoT [46]. However, the deployment location of the edge and the functionalities of the edge are controversial. Therefore, in applications, edge computing can be implemented as fog computing [47], mobile edge computing [48, 49], and mobile cloud computing [50]. In the fog computing paradigm, the analysis of local information is conducted on the "ground," and the coordination and global analytics are conducted in the "cloud." One can argue that all objects that are outside the cloud constitute the ground; thus, fog computing is almost the same as edge computing.

Mobile edge computing deploys cloud services at the edge of mobile networks such as 5G. Namely, the edge is deployed in close proximity to the eNodeB, which loops the traffic through the mobile edge computing servers for further data processing. Thus, in mobile edge computing, the edge is fixed, and the edge servers can be provided by telecommunication companies. Mobile cloud computing focuses mainly on mobile delegation: mobile devices should delegate the storage of bulk data and the execution of computationally intensive tasks to remote entities. A remote entity can be a centralized cloud or another mobile device. Therefore, the edge can consist of mobile devices in mobile cloud computing. Hence, one of the most active areas of research in the field of mobile cloud computing is the delegation of tasks to external services, especially to other mobile devices. Task delegation is also essential for edge computing [51].

In this study, we provide a lightweight and efficient framework for the detection of repackaged Android malware that is based on edge computing. In the proposed framework, the mobile devices need only to record the flow starting time, flow protocol, server-side IP address, server-side port, app name, and version for each network flow. Later, these data are sent to the edge servers for traffic tagging. Since the amounts of data that are saved and sent are small, the performances of mobile devices are not significantly degraded. The edge servers extract detection features from the labeled app network traffic and send the extracted features to the cloud to be clustered. After clustering, the original and repackaged apps can be efficiently distinguished. In our study, only network traffic is considered in Android malware detection. In contrast to previous network-based methods such as [14, 34–36], we do not need to model apps' network behaviors in advance, and our method is more useful in practice.

4. Framework for Android Malware Detection

The on-device detection of Android malware is challenging because mobile devices typically have limited resources. To alleviate the storage and computing limitations and prolong the lifetimes of the mobile devices, in this study, we propose a new framework for on-device detection of repackaged Android malware. The proposed framework is illustrated in Figure 3.

Our framework is composed of three layers: mobile devices, the edge, and the cloud. First, we explain the possible locations of the edge; later, we introduce the functionalities of each component. In our framework, the only requirement for the edge is that it can monitor and process the mobile network traffic. The mobile devices could connect to the Internet via WiFi or 3G/4G; hence, the edge nodes can be wireless gateways, household security gateways such as BitDefender Box and LTE eNodeBs. The edge nodes can also be routers that are located in the backbone network if they can observe the traffic that is generated by the mobile devices. Therefore, the deployment locations of the edge nodes are flexible in practice. Figure 4 illustrates the possible deployment scenarios of the edge.

In the proposed framework, the mobile devices must collect and send flow metaformation to the edge for data preprocessing. For this, an edge-client app is designed to be run on the mobile devices. The edge-client app operates as an Android VPN app. Its main objective is to collect and transfer the flow metainformation, which includes the app name, flow starting time, and flow protocol, to the edge iteratively. With the flow metainformation, the edge server preprocesses the network traffic by adding a label (the app name) to each flow. After that it extracts detection features from these labeled flows. The traffic labeling and feature extraction are illustrated in detail in Section 5.2. Once the feature extraction has been completed, the edge server sends these features to the cloud. With the received features, the cloud conducts the malware detection tasks and notifies the mobile devices of the detection results.

To further illustrate the advantages of the proposed framework, we compare it with the available frameworks. Since the proposed framework is based on edge computing, we will refer to it as the edge-computing-based framework in the following. Similarly, the available frameworks are termed

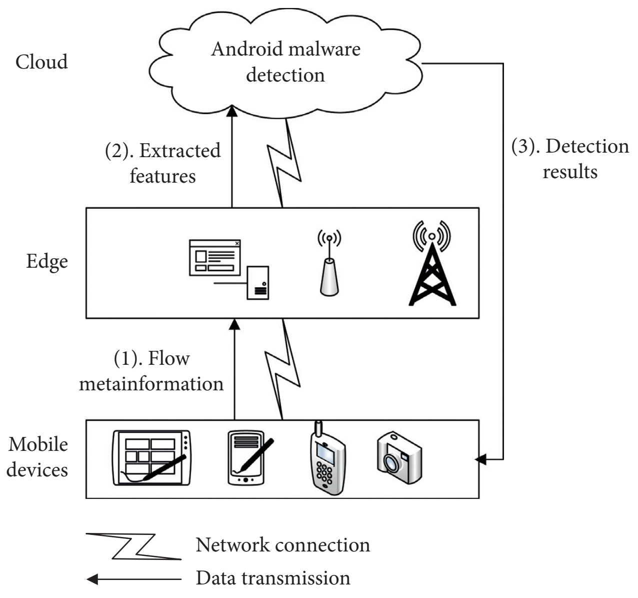


Figure 3: Framework for android malware on-device detection that is based on edge computing. Mobile devices send flow metainformation, such as the app name and the server-side IP address, to the edge. Edge servers capture app network traffic and extract HTTP contents and flow statistical features from the captured traffic. Later, the features are sent to the cloud for malware detection. The cloud also interacts with the mobile devices to return the detection results.

as the cloud-based framework (the server-side detection methods that were described in Section 1) and the network-based framework (the network-based methods that were introduced in Sections 1 and 3) for differentiation. Compared to the popular cloud-based framework (Figure 5(a)), the most prominent advantage of the edge-computing-based framework is that the feature extraction process is offloaded to the edge and the mobile devices must only save and transmit the flow metainformation. Thus, the proposed edge-computing-based framework is resource-friendly for mobile devices.

The network-based framework (Figure 5(b)) monitors network traffic without any interference with mobile devices. However, it is less accurate since the network nodes cannot know the origins of the network flows precisely, namely, it cannot identify app traffic with 100% accuracy. As demonstrated in the literature [14], since the network node cannot identify the app for each flow (e.g., some flows lacks signatures or are encrypted), the apps' network behaviors are incomplete, and the malware detection accuracy is reduced. In the proposed edge-computing-based framework, we design a lightweight edge-client app for gathering app information. Furthermore, mobile users' privacy can be better protected in our proposed framework. By deploying one's own edge server, a mobile user can control the types of information that can be used as detection features. In contrast, in the network-based framework, the network nodes can observe the full traffic content and extract any information they want, which severely threatens the privacy of mobile users.

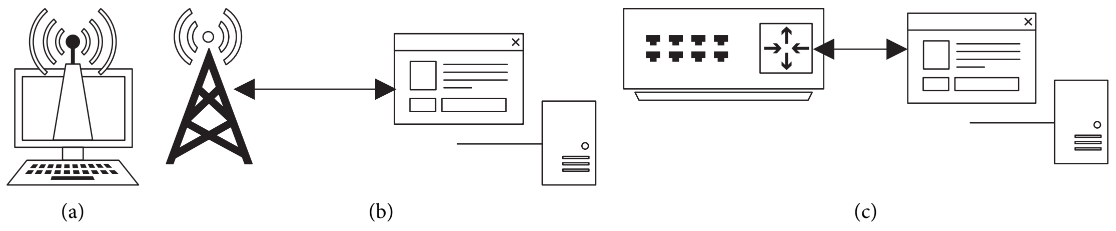


FIGURE 4: Possible deployments of the edge. The edge could be the wireless gateways or the dedicated servers that are connected to the eNodeB or the routers. (a) Wireless gateway. (b) eNodeB + server. (c) Router + server.

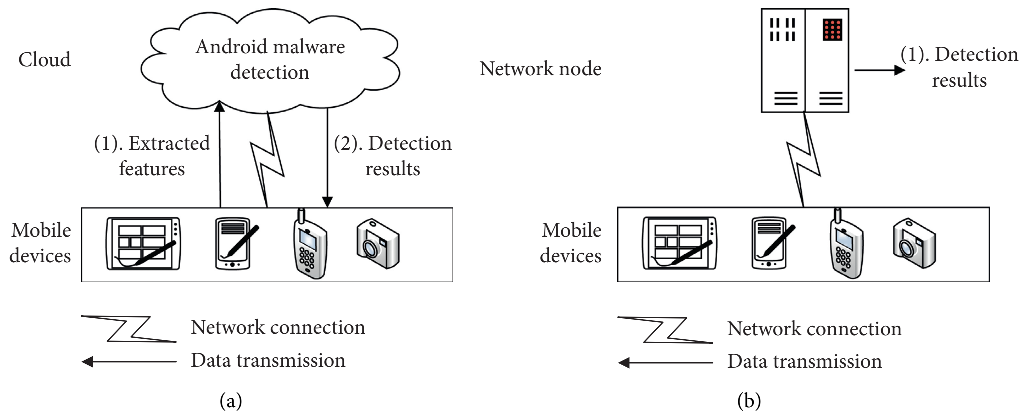


FIGURE 5: Available frameworks for android malware detection. Various approaches, such as [10], send the extracted features to a remote server other than to the cloud. We still regard them as following the cloud-based framework because there is no essential difference for malware detection. As illustrated in (b), the detection results are typically presented to the network administrator and are not fed back to the mobile devices. (a) Framework based on cloud. (b) Framework based on network.

After introducing the general architecture, we outline the detection methods that can be used in the edge-computing-based framework. The cloud can use various methods to detect Android malware. For example, it can use conventional signature matching [35] and network behavior analysis [8, 14] to identify malicious apps. However, these types of methods require a priori information (such as signatures or behavior models). In this paper, we propose a novel repackaged malware detection method that clusters the network traffic that is generated by the same app from various mobile devices. Figure 6 presents an example of this process.

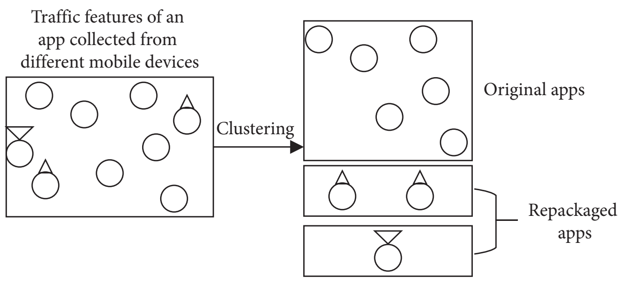

Traffic features of an app collected from different mobile devices
Clustering
Original apps
Repackaged apps

FIGURE 6: Illustration of traffic clustering for repackaged malware detection. The plain circles represent original apps and the circles with triangles and inverted triangles are repackaged ones.

The motivations behind the traffic clustering are twofold: first, most Android malware is produced by repackaging popular apps [4], and popular apps are often widely distributed. Second, the network traffic generated by the repackaged malware differs from that generated by the original app (as illustrated in Section 2.1). As illustrated in Figure 3, many mobile devices would connect to the cloud for malware detection. Hence, the cloud can observe the network traffic features of both the original apps and their repackaged versions with large probabilities due to the wide distribution of popular apps and the large number of mobile devices. If we regard the features of the original apps as the normal data and the features of the repackaged versions as the abnormal data, the repackaged malware detection problem is transformed into the *abnormal detection problem*. Clustering is an efficient method for solving this problem [52]. By clustering apps' network traffic features, we need not model apps' network behaviors in advance, and, more importantly, we can detect various repackaged versions of the original app, as illustrated in Figure 6. The details of the proposed malware detection method will be described in detail in Sections 5.3–5.5.

5. Methodology

In this section, we introduce the technical details of the proposed clustering method for Android malware detection. Table 1 defines the main notation that we use in this paper.

Table 1: Main notation used in this paper.

Notation	Meaning
i	Android app i
u	Number of mobile devices with app i installed
r	Number of devices with repackaged app i
f	Network flow
$C\text{-IP}_f$	Client-side IP address of f
$S\text{-IP}_f$	Server-side IP address of f
$S\text{-Port}_f$	Server-side port of f
$\text{App}N_f$	Name of the app that generates f
$\text{App}V_f$	Version of the app that generates f
T_i	Set time interval
T_f^u	Recording time of flow f_u at the edge server
d_i	Feature set of app i
w_j	Plaintext word in the feature sets
$V(d_i)$	Numerical vector of traffic contents
$F_{d_i}^j$	Feature vector for the encrypted flow j in d_i
$\text{TB}_{n\times m}(d_i)$	Traffic behaviors of d_i
$\text{CS}_{d_{ik}}$	Content similarity between d_i and d_k
$\text{BS}_{d_{ik}}$	Behavior similarity between d_i and d_k
$S_{d_{ik}}$	Final similarity between d_i and d_k

5.1. System Overview. Figure 7 illustrates the main procedure of the proposed approach. App i is the app for vetting, and it has been installed on u mobile devices. These u devices are connected to e edge servers. We assume that r devices have installed the repackaged malware (the malware is also presented as App i to deceive the users), and $r < (1/2)u$. Hence, most of the devices have installed the original version. This situation is reasonable because repackaged Android malware is typically distributed in third-party markets, which are less popular than official markets [23]. Thus, the numbers of downloads and installations are relatively small. We will use this characteristic to classify the clustering results.

To detect the repackaged malware, the edge server labels the network flows that are generated by App i according to the flow metainformation. Then, it extracts suitable traffic features (such as traffic content and statistics) from the labeled flows and sends the features to the cloud for processing. Therefore, there are u feature sets for App i in the cloud. The cloud filters these u sets according to the app versions and removes common traffic. After that it calculates the pairwise similarities of the feature sets. Finally, it clusters the similarity values and identifies r repackaged malware instances.

The traffic labeling, feature extraction, and filtering (labels ①, ②, and ③ in Figure 7), the similarity calculation on the traffic contents and behaviors (labels ④ and ⑤), and clustering of similarity values (label ⑥) are the core elements of the proposed method. They will be described in detail in the following sections.

5.2. Traffic Labeling, Filtering, and Feature Extraction

5.2.1. Traffic Labeling. In this study, traffic labeling is conducted to identify the corresponding app for each network flow, which is known as the app identification problem [53]. Extensive works have been conducted on the identification of apps from mobile network traffic [38, 54, 55]. However, the achieved identification accuracies are all lower than 100%. In our method, we design an edge-client app for collecting flow metaformation and identifying apps accurately. The flow metainformation is defined in the following.

Definition 1 (flow metainformation). For flow f, the flow metainformation includes the client-side IP address $C\text{-IP}_f$, the flow starting time T_f, the flow protocol P_f, the server-side IP address $S\text{-IP}_f$, the server-side port $S\text{-Port}_f$, the app name $\text{App}N_f$, and the app version $\text{App}V_f$. The flow f is bidirectional and P_f is the transport layer protocol, such as TCP or UDP.

The edge-client app operates similarly to a VPN app. The app sends the flow metainformation to the edge server at a fixed time interval T_i. Once it has received the flow metainformation, the edge server saves it into a metainformation database and can label mobile app traffic accurately. Denote the flow observed by the edge server as f_u. The edge server labels f_u as follows:

Step 1: extract the recording time T_f^u of f_u and its flow protocol P_f^u, client-side IP address $C\text{–IP}_f^u$, server-side IP address $S\text{–IP}_f^u$, and server-side port $S\text{–Port}_f^u$.

Step 2: search the metainformation database with P_f^u, $C\text{–IP}_f^u$,$S\text{–IP}_f^u$, and $S\text{–Port}_f^u$. If records match, the database returns the record that has the most recent flow starting time T_f^r. Otherwise, the edge server waits for a while and repeats Step 2.

Step 3: if T_f^u is close to T_f^r, the edge server labels flow f_u as $<\text{App}N_f, \text{App}V_f>$. Otherwise, it waits for a while and repeats Steps 2 and 3.

In the above steps, the client-side IP address $C\text{–IP}_f^u$ is used to identify a mobile device uniquely, which is correct when the edge server is the wireless gateway or the eNodeB. In such scenarios, the mobile devices directly connect to the edge; hence, the IP address is unique for each device. However, if the edge server is deployed at the backbone router, as illustrated in subgraph (c) of Figure 4, the $C\text{–IP}_f^u$ may be confusing because mobile devices could be located behind NATs (Network Address Translations), and they will share the same public IP address. To solve this issue, the edge-client app can insert special indicators into the packets to uniquely represent a device. In this study, we always assume that $C\text{–IP}_f^u$ is unique, and we will investigate the implementation of the device-indicator in future work.

In steps 2 and 3, the edge server utilizes a wait-and-repeat strategy to ensure that the traffic labeling is fresh and accurate. The edge server must wait because the mobile devices send flow metainformation to the edge server at a fixed time interval T_i (60 s in our experiments), and the saved flow metainformation in the database may be outdated. In our implementations, the waiting time is set to T_i, and if $|T_f^u - T_f^r| > 24\,\text{h}$, the returned flow metainformation is regarded as obsolete.

5.2.2. Traffic Feature Extraction. After traffic labeling, the edge server extracts detection features from the labeled

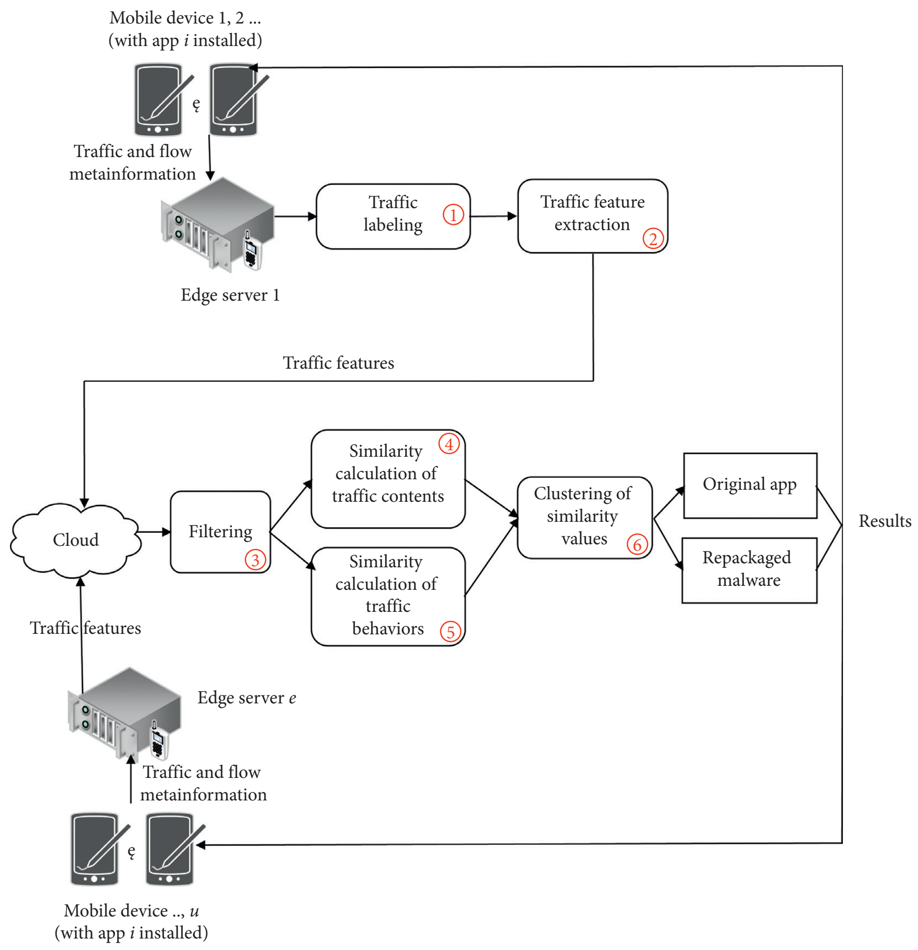


Figure 7: The main procedure of the proposed method for Android malware detection. The core parts of the proposed method are labeled by circled numbers.

traffic. The extracted traffic features are categorized into two groups: traffic content features and traffic behavior features. Traffic content features are the plaintext contents that are extracted from HTTP flows, and traffic behavior features are the statistics of network flows, such as packet sizes and intervals. Since Android apps may use an encrypted HTTPS protocol to transmit data [54], traffic behavior features also must be considered. The traffic content and traffic behavior features are described in detail in Sections 5.3 and 5.4.

5.2.3. Traffic Filtering. After traffic feature extraction, the edge servers send the extracted features, along with the client-side IP address, server-side IP addresses, server-side ports, app name AppN_f, and app version AppV_f, to the cloud. Note that App i is installed on u devices; hence, the cloud eventually has u feature sets for App i. Figure 8 illustrates the structure of the feature set. The cloud further filters these u feature sets to increase the detection accuracy. First, it divides the u sets into v categories according to the app version AppV_f. Therefore, each category contains the feature sets that are generated by the same version of App i. Then, for each category, the cloud removes the common traffic from the feature sets. The common traffic is defined as network

Client-side IP address,
App name: AppN_f,
App version: AppV_f,
Flow 1: server-side IP address 1, server-side port 1, flow features,
...
Flow f: server-side IP address f, server-side port f, flow features

Figure 8: Illustration of the feature set. The set contains the client IP address, app name, and app version. Each network flow that is generated by App i in a device is represented by the features extracted from the flow and by the server IP address and port.

flows that are contained in all features sets. These flows have the same server-side IP address and the same server-side port number. The reason for this step is that the common traffic will obfuscate the similarity calculations between apps. Otherwise, the similarity values between features sets are all high, and they cannot be effectively used to distinguish the repackaged malware from the original apps.

After the filtering has been completed, the similarities between feature sets are calculated, and the similarity values are clustered to identify the repackaged malware.

5.3. Similarity Calculation of Traffic Content. Android apps use mainly HTTP and HTTPS protocols to communicate with their remote servers [54]. For HTTP traffic, since it is plaintext based, we can efficiently use the flow content to calculate the similarity. The basic strategy is to convert HTTP traffic to document files, and the document files can be handled by information retrieval technologies for similarity calculation. HTTP flow content has also been used in app identification from network traffic [55]. However, in this study, we only consider partial HTTP header information to better protect mobile users' privacy.

In detail, first, we reassemble HTTP flows from packets and extract the flow contents by saving the corresponding packet contents into a text file as ASCII code. Figure 9 shows such an example. Then, we preprocess the text file by deleting the HTTP request content and the HTTP request header, except for the GET (or POST) and HOST fields. The HTTP response content is also deleted, namely, only the HTTP response header, the GET (or POST), and the HOST filed in the HTTP request header are saved. Via this approach, the mobile users' privacy can be protected from the cloud (for example, the mobile phone information (HUAWEI) is leaked by the User-Agent field in Figure 9). After that, the text file is spilt using special characters (e.g., {/, ., ,, =, ...}), and the processed file becomes the flow content features.

The similarity is calculated via the improved TF-IDF algorithm [56] and the cosine similarity. The term frequency measures the count of words in a document. We use the augmented frequency measurement. Denote the document (the feature set) as d_i and the word as w_j. Term frequency $tf(w_j, d_i)$ is calculated as

$$tf\left(w_j, d_i\right) = 0.5 + \frac{0.5 \times f\left(w_j, d_i\right)}{\max\left\{f\left(w, d_i\right): w \in d_i\right\}}, \tag{1}$$

where $f(w_j, d_i)$ is the number of occurrences of w_j in document d_i.

The inverse document frequency measures of the amount of information a word provides. Typically, it is the logarithmically scaled inverse fraction of the number of documents that contain the word. The inverse document frequency of w_j is calculated as

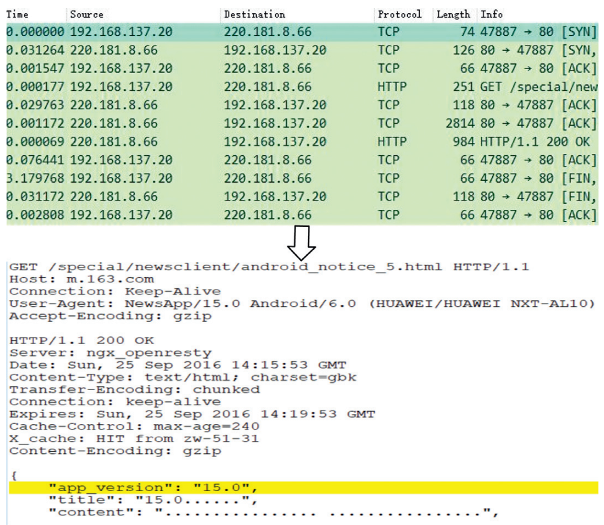
$$idf\left(w_j\right) = \log\left(\frac{u}{\left|\left\{d, w_j \in d\right\}\right|} + 0.01\right), \tag{2}$$

where u is the number of documents and $|\{d, w_j \in d\}|$ is number of documents in which the term w_j appears.

Denote by $p(w_j, d_i)$ the product of $TF(w_j, d_i)$ and $IDF(w_j)$; it is formulated as

$$p\left(w_j, d_i\right) = \frac{\log\left(tf\left(w_j, d_i\right)\right) \times idf\left(w_j\right)}{\sqrt{\sum_{j=0}^{c} \log^2\left(tf\left(w_j, d_i\right)\right) \times idf^2\left(w_j\right)}}. \tag{3}$$

After the TF-IDF operation, all documents are converted into equal-length numerical vectors. We use the cosine similarity to measure the similarity between two document vectors. For documents d_i and d_k, the corresponding



```
Time      Source           Destination      Protocol Length Info
0.000000 192.168.137.20   220.181.8.66     TCP        74 47887 → 80 [SYN]
0.031264 220.181.8.66     192.168.137.20   TCP       126 80 → 47887 [SYN,
0.001547 192.168.137.20   220.181.8.66     TCP        66 47887 → 80 [ACK]
0.000177 192.168.137.20   220.181.8.66     HTTP      251 GET /special/new
0.029763 220.181.8.66     192.168.137.20   TCP       118 80 → 47887 [ACK]
0.001172 220.181.8.66     192.168.137.20   TCP      2814 80 → 47887 [ACK]
0.000069 220.181.8.66     192.168.137.20   HTTP      984 HTTP/1.1 200 OK
0.076441 192.168.137.20   220.181.8.66     TCP        66 47887 → 80 [ACK]
3.179768 192.168.137.20   220.181.8.66     TCP        66 47887 → 80 [FIN,
0.031172 220.181.8.66     192.168.137.20   TCP       118 80 → 47887 [FIN,
0.002808 192.168.137.20   220.181.8.66     TCP        66 47887 → 80 [ACK]
```

```
GET /special/newsclient/android_notice_5.html HTTP/1.1
Host: m.163.com
Connection: Keep-Alive
User-Agent: NewsApp/15.0 Android/6.0 (HUAWEI/HUAWEI NXT-AL10)
Accept-Encoding: gzip

HTTP/1.1 200 OK
Server: ngx_openresty
Date: Sun, 25 Sep 2016 14:15:53 GMT
Content-Type: text/html; charset=gbk
Transfer-Encoding: chunked
Connection: keep-alive
Expires: Sun, 25 Sep 2016 14:19:53 GMT
Cache-Control: max-age=240
X_cache: HIT from zw-51-31
Content-Encoding: gzip

{
    "app_version": "15.0",
    "title": "15.0......",
    "content": "................ ................",
```

Figure 9: Example of converting an HTTP flow to a text file.

numerical vectors are denoted as $V(d_i)$ and $V(d_k)$ and the length of vector is t. The similarity between d_i and d_k is calculated as

$$\mathrm{CS}_{d_{ik}} = \frac{\sum_{l=1}^{t} \left(V\left(d_i\right)_l \times V\left(d_k\right)_l\right)}{\sqrt{\sum_{l=1}^{t} V\left(d_i\right)_l^2} \times \sqrt{\sum_{l=1}^{t} V\left(d_k\right)_l^2}}. \tag{4}$$

Suppose that, in category v_i (we divide the u feature sets generated by App i in to v categories in the traffic filtering step), there are c feature sets. For each feature set, we save the flow features of all HTTP flows into a text file, and c text files are obtained. Then, we calculate the numerical vector for each feature set via equation (3). Last, we compare the numerical vector with each other as equation (4), and gain c similarity values ($\mathrm{CS}_{d_{ii}} = 1$) for each feature set. We represent these similarity values of feature set d_i ($i = 1, 2, \ldots, c$) as a vector S^i_{contents} and $S^i_{\text{contents}} = (\mathrm{CS}_{d_{i1}}, \mathrm{CS}_{d_{i2}}, \ldots, \mathrm{CS}_{d_{ic}})$. The vectors S^i_{contents} ($i = 1, 2, \ldots, c$), in combination with the similarity values of traffic behaviors, will be clustered to identify the abnormal data points, namely, the repackaged malware.

5.4. Similarity Calculation of Traffic Behaviors. In addition to calculating the similarity values of traffic contents, the similarity of traffic behaviors must be calculated due to encrypted flows. We use m ($m = 11$ in our study) statistical features as listed in Table 2 to represent the behavior of an encrypted network flow. The units of the flow duration and the packet interval time are milliseconds in our experiments.

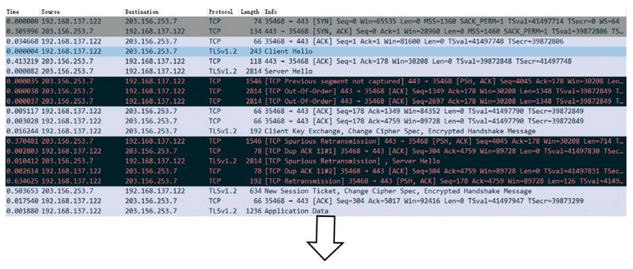
With the flow statistical features, for feature set d_i, the jth encrypted flow can be represented as a vector in the feature space:

$$F^j_{d_i} = \left(f^1_j, f^2_j, \ldots, f^m_j\right), \tag{5}$$

where f^l_j represents the value of the lth statistical feature, $1 \le l \le m$. Figure 10 shows an example of feature representation for encrypted traffic. Consequently, the traffic behaviors of d_i can be represented as a matrix of f^p_j:

Table 2: Flow statistical features.

Bytes sent by the app and bytes sent by the server
Numbers of outgoing and incoming packets
Mean and std. dev. of the outgoing and incoming packet sizes' flow duration
Mean and std. dev. of the interpacket time



```
Time     Source          Destination     Protocol Length Info
0.000000 192.168.137.122 203.156.253.7   TCP        74 35468 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1360 SACK_PERM=1 TSval=41497714 TSecr=0 WS=64
0.305996 203.156.253.7   192.168.137.122 TCP       134 443 → 35468 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=39872806 TS…
0.034668 192.168.137.122 203.156.253.7   TCP        66 35468 → 443 [ACK] Seq=1 Ack=1 Win=81600 Len=0 TSval=41497748 TSecr=39872806
0.000004 192.168.137.122 203.156.253.7   TLSv1.2   243 Client Hello
0.413219 203.156.253.7   192.168.137.122 TCP       118 443 → 35468 [ACK] Seq=1 Ack=178 Win=30208 Len=0 TSval=39872848 TSecr=41497748
0.000082 203.156.253.7   192.168.137.122 TLSv1.2  2814 Server Hello
0.000035 203.156.253.7   192.168.137.122 TCP      1546 [TCP Previous segment not captured] 443 → 35468 [PSH, ACK] Seq=4045 Ack=178 Win=30208 Len…
0.000038 203.156.253.7   192.168.137.122 TCP      2814 [TCP Out-Of-Order] 443 → 35468 [ACK] Seq=1349 Ack=178 Win=30208 Len=1348 TSval=39872849 T…
0.000037 203.156.253.7   192.168.137.122 TCP      2814 [TCP Out-Of-Order] 443 → 35468 [ACK] Seq=2697 Ack=178 Win=30208 Len=1348 TSval=39872849 T…
0.005117 192.168.137.122 203.156.253.7   TCP        66 35468 → 443 [ACK] Seq=178 Ack=1349 Win=84352 Len=0 TSval=41497790 TSecr=39872849
0.003028 192.168.137.122 203.156.253.7   TCP        66 35468 → 443 [ACK] Seq=178 Ack=4759 Win=89728 Len=0 TSval=41497790 TSecr=39872849
0.016244 192.168.137.122 203.156.253.7   TLSv1.2   192 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
0.370481 203.156.253.7   192.168.137.122 TCP      1546 [TCP Spurious Retransmission] 443 → 35468 [PSH, ACK] Seq=4045 Ack=178 Win=30208 Len=714 T…
0.002803 192.168.137.122 203.156.253.7   TCP        78 [TCP Dup ACK 11#1] 35468 → 443 [ACK] Seq=304 Ack=4759 Win=89728 Len=0 TSval=41497830 TSec…
0.010412 203.156.253.7   192.168.137.122 TLSv1.2  2814 [TCP Spurious Retransmission] , Server Hello
0.002614 192.168.137.122 203.156.253.7   TCP        78 [TCP Dup ACK 11#2] 35468 → 443 [ACK] Seq=304 Ack=4759 Win=89728 Len=0 TSval=41497831 TSec…
0.634625 192.168.137.122 203.156.253.7   TCP       192 [TCP Retransmission] 35468 → 443 [PSH, ACK] Seq=178 Ack=4759 Win=89728 Len=126 TSval=4149…
0.503653 203.156.253.7   192.168.137.122 TLSv1.2   634 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
0.017540 192.168.137.122 203.156.253.7   TCP        66 35468 → 443 [ACK] Seq=304 Ack=5017 Win=92416 Len=0 TSval=41497947 TSecr=39873299
0.001880 192.168.137.122 203.156.253.7   TLSv1.2  1236 Application Data
```

<2761, 16696, 16, 12, 173, 1294, 289, 1177, 3392, 121, 190>

Figure 10: Example of the conversion of an encrypted flow to a feature vector.

$$\mathrm{TB}_{n\times m}(d_i) = \left(F_{d_i}^1, F_{d_i}^2, \ldots, F_{d_i}^n\right)^T, \tag{6}$$

where n is the number of encrypted flows that are contained in d_i and $n \geq 1$.

Since different feature sets may differ in terms of the number of encrypted flows, traffic behaviors $\mathrm{TB}_{n_i\times m}(d_i)$ and $\mathrm{TB}_{n_k\times m}(d_k)$ may differ in terms of different dimension, namely, the value of n_i should not be equal to that of n_k. To efficiently calculate the similarity between $\mathrm{TB}_{n_i\times m}(d_i)$ and $\mathrm{TB}_{n_k\times m}(d_k)$, we calculate the Frobenius norm of $\mathrm{TB}_{n_i\times m}(d_i)$ and $\mathrm{TB}_{n_k\times m}(d_k)$. The formula is presented in equation (7), where $\mathrm{norm}(\cdot)$ normalizes the value of f_j^l to [0, 1]:

$$\mathrm{FN}(d) = \sqrt{\sum_{j=1}^{n}\sum_{l=1}^{m}\left|\mathrm{norm}\left(f_j^l\right)\right|^2}. \tag{7}$$

Then, the distance between $\mathrm{TB}_{n_i\times m}(d_i)$ and $\mathrm{TB}_{n_k\times m}(d_k)$ is calculated as

$$\mathrm{dis}(d_i, d_k) = \left|\mathrm{FN}(d_i) - \mathrm{FN}(d_k)\right|. \tag{8}$$

Finally, the similarity between $\mathrm{TB}_{n_i\times m}(d_i)$ and $\mathrm{TB}_{n_k\times m}(d_k)$ can be calculated by

$$\mathrm{BS}_{d_{ik}} = 1 - \frac{\mathrm{dis}(d_i, d_k)}{\max\left[\mathrm{dis}\left(d_r, d_q\right), r, q = 1, 2, \ldots, u, r \neq q\right]}. \tag{9}$$

In the above process, n_i and n_k are assumed to exceed 1. If $n_i = 0$ ($n_k = 0$) and $n_k \geq 1$ ($n_i \geq 1$), $\mathrm{BS}_{d_{ik}}$ is set to 0 to reflect the significant difference between d_i and d_k. However, if $n_i = 0$ and $n_k = 0$, namely, there is no encrypted flow in both d_i and d_k, $\mathrm{BS}_{d_{ik}}$ is set to 1 because they exhibit the same behaviors, i.e., they do not produce any encrypted traffic.

Similarly, we let $\mathrm{BS}_{d_{ii}} = 1$ and represent these similarity values as a vector $S_{\text{behavior}}^i = \left\{\mathrm{BS}_{d_{i1}}, \mathrm{BS}_{d_{i2}}, \ldots, \mathrm{BS}_{d_{iu}}\right\}$.

5.5. Clustering of Similarity Values and Malware Detection.

From the traffic content similarity $\mathrm{CS}_{d_{ik}}$ and the traffic behavior similarity $\mathrm{BS}_{d_{ik}}$, we can determine the final similarity value $\mathrm{BS}_{d_{ik}}$ between the feature sets d_i and d_k, as expressed in equation (10). In this equation, q is the weight value. In our experiments, we set $q = 0.5$, namely, the traffic content similarity and the traffic behavior similarity are assigned the same weight. This is because apps typically produce almost the same numbers of HTTP and HTTPS flows [57]. We also evaluate various weighted values in Section 6.2, and the experimental results support that 0.5 is the most suitable value. For feature set d_i, a vector $S_{d_i} = (S_{d_{i1}}, S_{d_{i2}}, \ldots, S_{d_{iu}})$ is finally obtained, which contains the similarity values between d_i and the other feature sets. The set of vector $\left\{S_{d_i}\right\}, i = 1, 2, \ldots, u$ will be clustered for the detection of repackaged malware:

$$S_{d_{ik}} = q \times \mathrm{CS}_{d_{ik}} + (1 - q) \times \mathrm{BS}_{d_{ik}}. \tag{10}$$

To be specific, we use the density peak clustering method [58] to realize our objective. Density peak clustering is based on the following assumption: cluster centers are characterized by a higher density than their neighbors and by a relatively large distance from points that have higher densities. Hence, density peak clustering can automatically identify the correct number of clusters. This is important because we do not know whether there is repackaged malware or how many types of repackaged malware there are. In other words, we cannot determine the number of clusters in advance. Density peak clustering can help us overcome this challenge.

We assume that most mobile devices have installed the original apps, as discussed in Section 5.1. Therefore, after clustering, the cluster that has the largest number of voxels will be recognized as original, and the remaining clusters will be regarded as suspicious. However, suspicious apps are not necessarily repackaged malware. For example, apps can be repackaged to show ads only. Repackaged apps of this type should not be simply judged as malware. Additionally, original apps can be suspicious due to different user behaviors. To further reduce the frequency of the false alarms, we can check the server hostnames that are contained in the repackaged clusters using VirusTotal (the URLs are included in the HTTP traffic contents). If all of the servers' hostnames for a suspicious cluster C_r are judged as clean, it will be recognized as a cluster of abnormal but harmless apps. Otherwise, it will be identified as repackaged malware. Via this approach, we can accurately detect repackaged Android malware. In this step, we only use VirusTotal as a white list to exclude false alarms. We do not use it as a backlist (detection signatures) to detect repackaged malware. Malware detection is still realized by clustering the similarity values.

6. Evaluation

6.1. System Implementation and Dataset. Figure 11 illustrates our experimental setup. The Android phones were connected to the edge server via WiFi, and each phone had installed the edge-client app, which sends the flow metainformation to the edge server using Socket (java.net.Socket). The edge is a Dell PowerEdge R730 server (with 32 processors, 16 GB RAM, and 12 TB hard drives) that is configured as an access point. In the edge server, we used *tcpdump* (http://www.tcpdump.org/) to collect network traffic. The similarity calculations of traffic contents and behaviors and the clustering of similarity values were implemented on the Windows Azure cloud computing platform. The URL checking was conducted using *Virustotal* (https://www.virustotal.com/).

In the implementation, we used *aria2c* (https://aria2.github.io/) to download apps from Androzoo (https://androzoo.uni.lu/). aria2c was encapsulated as commands in Python code for parallel downloading. After downloading the apps, we automatically installed them on phones using the *adb* shell command. Each phone had 40 tested apps installed. We have implemented the edge-client app based on Android VPN service. In our implementations, the edge-client app correlated network flows and apps by resolving the ports and PIDs (Process IDs) send the newly collected flow metainformation to the edge server every 60 seconds. The edge server used *tcpdump* to collect network traffic, which was saved as pcap files. To extract traffic features, we used *Splitcap* (https://www.netresec.com/?page=SplitCap) to restore tcp flows from pcap files. Then, for HTTP traffic, we extracted the contents using tshark (https://www.wireshark.org/docs/man-pages/tshark.html) commands *-z follow*, *tcp*, and *ascii*. The packet sizes and packet interval times were also obtained by tshark with commands *-e frame.len –e frame.time_relative*. In the cloud, $\mathrm{CS}_{d_{ik}}$ and $\mathrm{BS}_{d_{ik}}$ were calculated using scikit-learn (http://scikit-learn.org/) and Numpy (https://plot.ly/numpy/norm/). The code of density peak clustering is available on GitHub (https://github.com/jasonwbw/DensityPeakCluster), and we have fixed various bugs that were present in the original code.

We have downloaded 200 pairs of original and repackaged apps from Androzoo for testing. However, not all of these downloaded repackaged apps are malicious. Some apps are only repackaged to show extra ads. To screen out Android malware from the downloaded repackaged apps, we further checked the 200 repackaged apps using Virustotal. If an app was judged as safe by all the antivirus tools, it was labelled as *repackaged_safe* in our experiments. Otherwise, it was labelled as *repackaged_malware*. In total, 143 repackaged apps were detected as malware by Virustotal. Nevertheless, in our experiments, we evaluated the proposed clustering method on all 200 repackaged apps to obtain more realistic performance results.

The downloaded apps were run in 10 rooted Android phones in parallel to accelerate the experimental process. For all apps, we used Droidbot [59] to send input events to them and to generate network traffic automatically. We ran each repackaged app (repackaged_safe and repackaged_malware apps) 10 times and the corresponding original app 30 times to generate sufficient network traffic. The running times of the original apps exceeded those of the repackaged apps because we assume that most of the apps are original, as discussed in Section 5.1. Consequently, we obtained 40 traffic sets for each repackaged-original app pair. Note that, from the perspectives of the mobile devices and the edge server, there are only 200 distinct apps because the repackaged app has the same name as the corresponding original app. Therefore, our experiments have simulated $40 * 200 = 8000$ mobile devices. For each repackaged-original app pair, there were 30 devices with the original app and 10 devices with the repackaged version.

In total, we have gathered almost 89 GB network traffic. The datasets that are used in our experiments are summarized in Table 3. In the table, the traffic dataset sizes are the sizes of the captured network traffic, and the feature dataset sizes are the sizes of the extracted features. In our experiments, the HTTP traffic contents and the flow statistical features, which are listed in Table 2, were both saved as text files. As shown in Table 3, the feature datasets are significantly smaller than the traffic datasets (25G vs 63G and 9G vs 26G). Thus, the edge computing can reduce the data size and increase the data acquisition speed for cloud computing [24]. Since the feature dataset sizes are still large (25G + 9G), we did not send all these feature data to the cloud simultaneously. Instead, we sent the feature data one by one. When the feature data of an app were successfully received by the cloud, its traffic similarities were calculated and clustered immediately. After that the feature data of another app would be sent and analyzed. The experimental results will be presented in the following.

6.2. Experimental Results. In this section, we mainly evaluate the detection efficiency of Android malware by clustering similarity values. The performance evaluation of the proposed method in terms of power, memory, and CPU consumptions is elaborated in the following Section 6.3.

Similar to previous studies [8, 14], to measure the performance of the proposed method, Accuracy and *F*-measure metrics are utilized, which are defined in equations (11) and (12). In equation (11), TP, FN, FP, and TN are defined in Table 4. In Table 4, *repackaged_x* represents *repackaged_safe* or *repackaged_malware*. The precision and recall in equation (12) are calculated via equations (13) and (14):

$$\text{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FN} + \mathrm{FP} + \mathrm{TN}}, \tag{11}$$

$$F - \text{measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}, \tag{12}$$

$$\text{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \tag{13}$$

$$\text{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}. \tag{14}$$

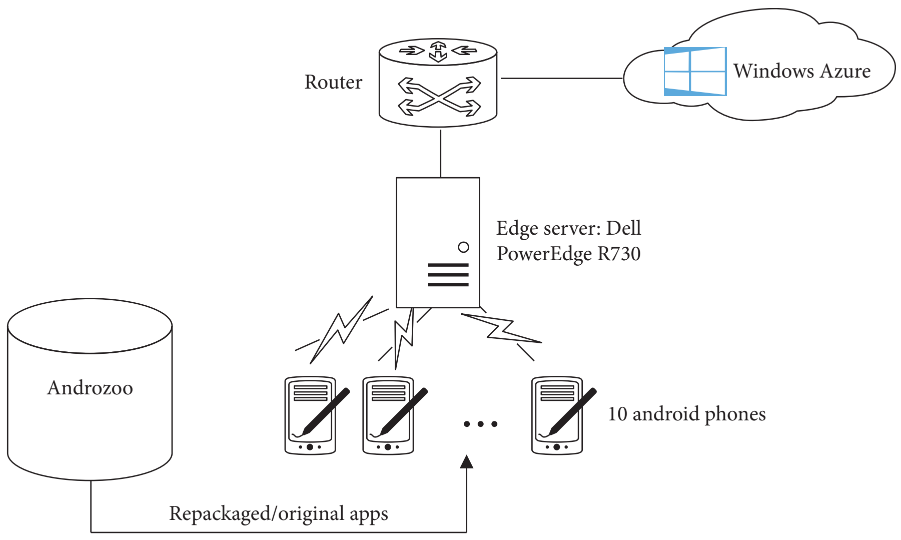


Figure 11: Experimental setup.

Table 3: Datasets used in the experiments.

	# of apps	# of runs	# of HTTP flows	# of encrypted flows	Traffic dataset sizes	Feature dataset sizes
Original apps	200	30	340,015	183,452	≈63 G bytes	≈25 G bytes
Repackaged apps	200	10	110,670	93,041	≈26 G bytes	≈9 G bytes

Table 4: Confusion matrix.

Observed	Detected	
	Repackaged_x app	Original app
Repackaged_x app	TP (# of TPs)	FN (# of FNs)
Original app	FP (# of FPs)	TN (# of TNs)

We have evaluated the Accuracy and *F*-measure for each repackaged app. Recall that we ran each repackaged app 10 times and the corresponding original app 30 times. Therefore, for each app, the number of the original app traffic sets is 30, and the number of the repackaged app traffic sets is 10, namely, TP + FN = 10 and TN + FP = 30 for each app. We have randomly chosen 20 apps from repackage_malware, and the experimental results for these apps are presented in Figures 12 and 13. Detailed descriptions of the chosen 20 apps are listed in Table 5. In the table, the main function of each app is determined by analyzing the package name from the AndroidManifest.xml and from the Google search results of the package name. As depicted in Figure 12, 100% accuracy was realized on a total of 6 apps (app3, app5, app9, app10, app13, and app18). The minimum accuracy is 85% (app2, 34/40) and the average accuracy is 95.2%. Figure 13 presents the *F*-measure values. In our experiments, the maximum *F*-measure value is 1, and the minimum value is 0.737. The average *F*-measure value is 0.888. These results demonstrate the satisfactory performance of our proposed clustering method.

We have also calculated the average accuracy and *F*-measure values for all repackaged_safe and repackaged_malware apps. In these calculations, TP + FN = 57 ∗ 10 = 570 and TN + FP = 57 ∗ 30 = 1710 for

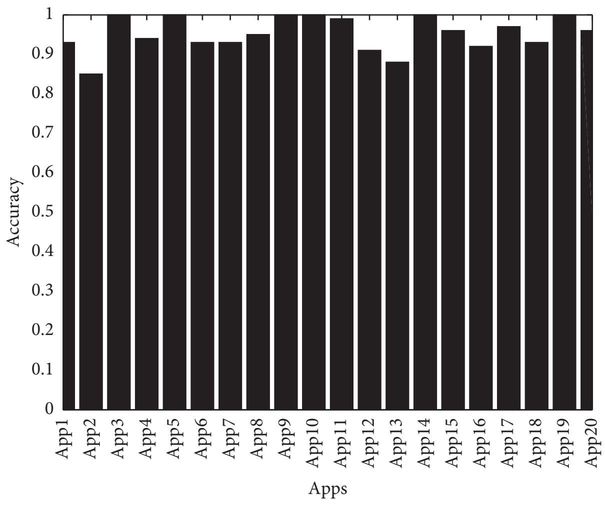


Figure 12: Detection accuracies for the 20 selected apps.

repackaged_safe apps. For repackaged_malware apps, TP + FN = 143 ∗ 10 = 1430 and TN + FP = 143 ∗ 30 = 4290. This is because in our experiments, 143 apps were identified as repackaged malware and 57 apps as repackaged safe by Virustotal. Thus, the parameter values differ. The experimental results are listed in Table 6.

According to Table 6, the average accuracy values for repackaged_safe and repackaged_malware apps are both high. However, the average *F*-measure for repackaged_safe apps is only 0.78, which is lower than that of repackaged malware. This indicates that repackaged safe apps are

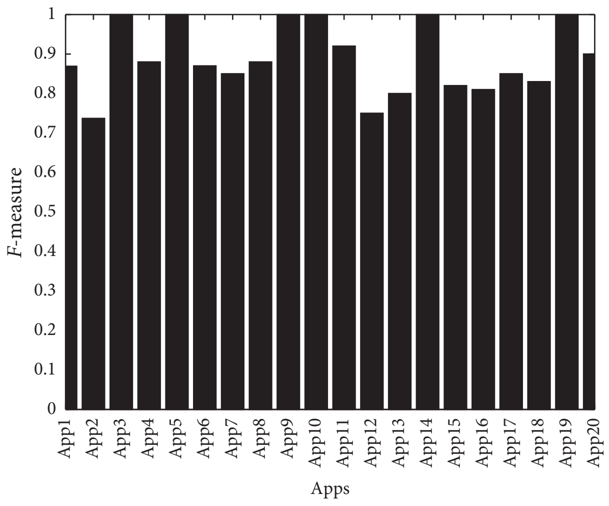


Figure 13: Detection *F*-measure for the 20 selected apps.

Table 5: Detailed descriptions of the chosen 20 apps.

	Main function of app	# of HTTP flows	# of encrypted flows
App1	VideoPlayer	453	51
App2	Game	657	1455
App3	Browser	1531	1478
App4	MusicPlayer	341	25
App5	Game	612	289
App6	Game	788	123
App7	Game	1336	524
App8	Fitness	357	892
App9	Email	122	431
App10	Game	963	790
App11	MusicPlayer	1128	547
App12	Game	1598	426
App13	News	2460	1178
App14	News	2891	1543
App15	Chat	171	49
App16	Email	143	368
App17	WeatherForecast	397	120
App18	Game	1832	921
App19	Fitness	732	362
App20	OnlineLearning	2378	834

difficult to distinguish from their original apps. This is because repackaged_safe apps only generate small amounts of additional network flows, and their network traffic features are not clearly distinguishable from those of normal traffic. Fortunately, the average *F*-measure of repackaged malware apps is outstanding. This may be due to the fact that malware typically send stolen data to or receive commands from remote servers. Hence, their network traffic features are special and distinguishable. Figure 14 presents an example of app clustering and illustrates the distinguishability of the repackaged malware and the original app. These results demonstrate that repackaged Android malware can be effectively detected by comparing their network traffic with that of original apps.

Table 6: Average accuracy and *F*-measure.

	Average accuracy (%)	Average *F*-measure
Repackaged_safe	90.3	0.78
Repackaged_malware	96.9	0.94

For repackaged malware apps, the accuracy results with different weight values (q in equation (10)) are presented in Figure 15. As depicted in the figure, when $q = 0.5$, we realize the highest accuracy rate, 96.9%. If $q = 0$, namely, only the features of encrypted traffic are used, the accuracy result is 75.3%. The result is 61.7% when $q = 1$, namely, only the features of plaintext traffic are considered. These results demonstrate that both the encrypted and plaintext traffic contribute to the differentiation of the original apps and the repackaged malware. However, the encrypted traffic contributes more than the plaintext traffic. This may because malicious apps usually connect to their servers by security protocols to evade IDS detection.

In the previous experiments, we checked the servers' hostnames using VirusTotal to reduce the frequency of the false positives. For various clusters that contained less than 20 elements, if all the servers' hostnames were judged as clean, the cluster was identified as an original app rather than a repackaged one. As argued in Section 5.5, this process only reduces the frequency of false alarms. To support this argument, we list the numbers of true positives and false positives of the 143 repackaged malware in Table 7 for comparison. When conducting the hostname checking, the number of true positives is 1400, and the result is the same when the hostname checking is turned off. This finding supports that the malware is detected by clustering the similarity values. However, the number of false positives increased from 47 to 213 when the hostname checking was turned off. Therefore, we suggest checking the servers' hostnames after clustering to increase the practicality of the method. Note that false positives still occur when we check the hostnames by VirusTotal. Hence, there may be malware in the original apps. In our dataset, the typical examples are *com.mobo.video.player.pro* and *cn.cf.shop ele1.taoxiaosan*. We further analyze these two apps by reading their decompiled codes, and we determine that they indeed establish connections with suspicious servers. In detail, in various runs, app *com.mobo.video.player.pro* connects with http://www.topappsquare.com,which is judged as malicious by *CyRadar* in VirusTotal. App *cn.cf.shop ele1.taoxiaosan* visits suspicious URL http://955.cc/jSn9. Since these apps are the original apps, they are identified as false positives in the experiments.

In our experiments, we chose density peak clustering as the clustering method because density peak clustering can automatically determine the correct number of clusters [58]. Hence, our method can detect various versions of malware that are repackaged from the same original app. Unfortunately, in the apps that were downloaded from AndroZoo, no malware was repackaged from the same original app. To evaluate the efficiency of detecting versions of malware, we have created two malware instances from a popular game

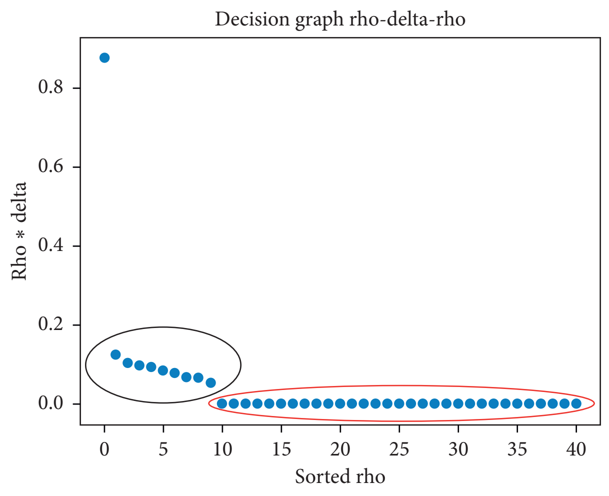


Figure 14: Example of app clustering. Obviously, two clusters are identified. The dot in the top left corner is an outlier. The outlier is detected as an abnormal but harmless app by our method. This figure was generated by the density peak clustering tool.

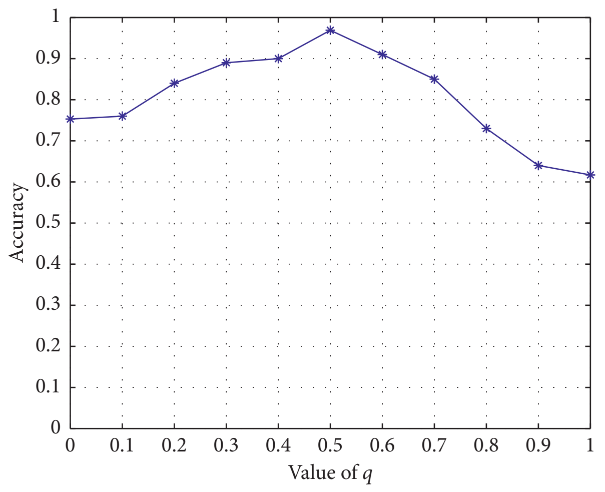


Figure 15: Accuracies of various weight values for traffic content similarity and traffic behavior similarity.

Table 7: True positives and false positives with and without hostname checking.

	# of true positives	# of false positives
With hostname checking	1400	47
Without hostname checking	1400	213

app, namely, *air.com.aceviral.motox*3. One is produced by grafting AnserverBot's source code into *air.com.aceviral.motox*3, and the other is created by adding code for stealing users' contacts and short messages and sending them to a remote server. Similarly, each malware is run 10 times and the original app *air.com.aceviral.motox*3 is also run 10 times for validation. The clustering result is presented in Figure 16. Apparently, there are 3 clusters. The experimental results show that the value of *F*-measure for each malware is 1. Hence, our method can detect various malware that were repackaged from the same original app.

6.3. Comparison. In this section, we compare our method with the typical methods that are based on network or cloud, as illustrated in Figure 5. The comparative indicators are the detection accuracy, *F*-measure, average power, CPU, and memory consumptions of the mobile device. First, we compare the proposed method with AppFA [14], which is implemented completely at the network level. Since AppFA also must analyze app traffic, we directly use the traffic sets of the repackaged malware apps for comparison. While testing AppFA, we deleted the label information that was contained in the traffic sets and mixed the network flows as a whole. Then, we used the signature matching and the constrained *K*-means clustering algorithm proposed in [14] to restore app sessions, namely, the label for each flow. The similar apps used in AppFA were chosen in Google Play via the method introduced in [14], and for each repackaged malware, 10 similar apps were selected as the peer group. The peer group apps were run once by Droidbot to generate network traffic. The comparison results are listed in Table 8.

Since AppFA need not to install some special apps on mobile devices, the extra power, memory, and CPU consumptions are 0. However, the detection accuracy is much lower (86.3% vs. 96.9%) than that of the proposed method, as compared in Table 8. In the proposed method, we use the edge-client app to record flow metainformation; thus, network flows can be accurately labeled. However, in AppFA, flow labeling is realized via traffic clustering, and the clustering accuracy cannot be 100%. Meanwhile, AppFA compares the apps' network behaviors with those of their similar apps, which also results in errors. In the proposed method, we directly compare the repackaged malware with its original app, and the detection accuracy is increased. The average power, CPU, and memory consumptions of the proposed method are also low and acceptable.

In Table 8, the power, memory, and CPU consumptions for our edge-client app were obtained with the help of app GT (https://github.com/Tencent/GT). GT is a portable tool that runs on smartphones for debugging the APP internal parameters and the code time-consuming statistics. The average values for the power, memory, and CPU consumptions are calculated as follows. Before running an app with Droidbot, we started the edge-client app by GT. When the Droidbot was stopped, we recoded the power, CPU, and memory consumptions. We repeated the process until all apps (including 143 repackaged malware apps and their original versions) had been tested. Via this way, we obtained the average power, CPU, and memory consumptions for the edge-client app. According to Table 8, the edge-client app is lightweight and has minimal impact on the performances of the mobile devices.

Then, we compare the proposed method with SAMA-Droid [44]. SAMADroid monitors Android system calls and sends the calls to a centralized server for the detection of Android malware. SAMADroid is a typical cloud-based on-

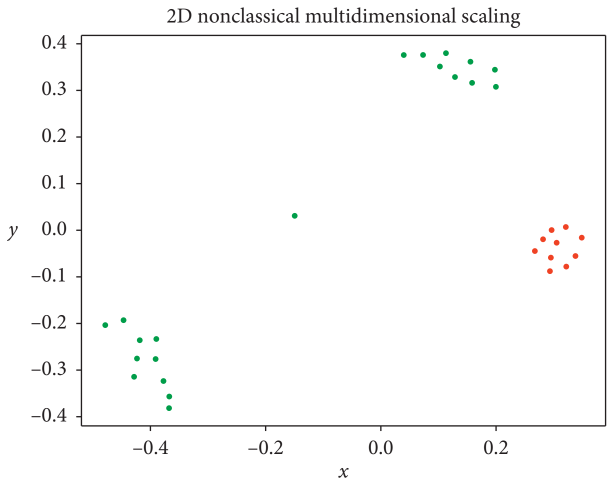


Figure 16: Clustering results of various malware that were repackaged from *air.com.aceviral.motox3*. The outlier in the middle is detected as an abnormal but harmless app. This figure was generated by the density peak clustering tool.

Table 8: Comparison of the proposed method with AppFA.

	Proposed method	AppFA
Accuracy	96.9%	86.3%
F-measure	0.94	0.75
Average power consumption	1.2%	0
Average CPU consumption	0.6%	0
Average memory consumption	6.54 MB	0

device detection method. We have implemented the dynamic detection according to the description in [44]. In detail, the APIs that are invoked by the running apps are monitored by the tool *Sensitive API Monitor* (https://github.com/cyruliu/Sensitive-API-Monitor). As done in [44], the frequency values of 10 system calls (open, ioctl, brk, read, write, close, sendto, sendmsg, recvfrom, and recvmsg) are recorded. These recorded frequency values are sent to Windows Azure for malware detection. We evaluate the dynamic detection performance of SAMADroid on our 143 repackaged_malware apps and their original versions. In the evaluation, the frequency values of the original apps are used to create the normality model, and the frequency values of the repackaged malware apps are used for testing. We also use the app GT to record the power, memory, and CPU consumptions of SAMADroid. Note that our phones were all rooted, and *Sensitive_API_Monitor* and GT can be used. The experimental results are presented in Table 9.

Last, we compare our method with Shabtai's method [8]. Shabtai's method detects Android malware at the mobile devices, and it considers the app's network traffic patterns only; the HTTP traffic contents are not considered. This gives us an opportunity to compare our method with methods that have been specially designed for encrypted traffic. We have implemented Shabtai's method and chosen the feature subset #2 as the detection features. Feature subset #2 includes Avg. Sent Bytes, Avg. Rcvd. Bytes, and Pct. of Avg. Rcvd. Bytes, among other values, and it is proved to be the best feature subset for malware detection [8]. The Decision/Regression tree is chosen as the classification algorithm. The testing dataset is the traffic that was generated by the 143 repackaged malware instances, and the training set is the traffic that was generated by the corresponding original apps. The classification results are listed in Table 10. According to the table, the detection accuracy and *F*-measure values of our method are significantly higher than those of Shabtai's method. This indicates that the HTTP traffic provides useful information for Android malware detection. Since Shabtai's method is conducted at the mobile devices, it consumes more resources than our method, as compared in Table 10.

7. Discussion

The results of our extensive experiments demonstrate that repackaged Android malware can be efficiently and effectively detected by clustering network traffic at the edge devices. However, there are still limitations for our method. First, we assume the most of the devices have installed the original apps ($r < (1/2)u$, as described in Section 5.1). This is reasonable in most cases since mobile users typically download apps from the official markets. However, in areas where official markets such as the Google play and Amazon are banned, it is possible that the most mobile devices have installed the repackaged apps. This may cause false positives. Second, if the malware does not generate sufficient traffic, our method cannot detect it. For example, some malware only sends fraudulent premium SMS messages. This type of malware cannot be detected by monitoring its network traffic. Third, we rely on third-party tools such as Virustotal for the identification of malicious URLs and to reduce the number of false alarms. If a malicious app rents legal platforms such as the public cloud as the remote control server, our method will judge it as a normal app. Hence, false negatives will be generated.

To overcome these limitations, we could combine the proposed method with available methods such as SAMADroid. In such cases, the edge-client app will not only record flow metainformation but also monitor app system behaviors. However, the system behavior monitoring does not need to be conducted all the time. If abnormal but harmless or lower traffic apps are detected, the edge-client app will be informed to record the corresponding apps' system behaviors. Similarly, the edge-client app can send the app system behaviors to the edge server for further processing. The performance of this combined method will be evaluated in our future work. One can also combine the proposed framework with AppFA [14] for the detection of other types of Android malware. For example, the edge extracts app traffic behavior features and the cloud can compare apps' behaviors with their peer groups to identify abnormalities. Furthermore, one could first use clustering techniques [60] to group repackaged apps according to the same malicious modules and then analyze the behaviors of the representative repackaged apps in each group to obtain more accurate behavior features.

Table 9: Comparison of the proposed method with SAMADroid.

	Proposed method	SAMADroid
Accuracy	96.9%	83.1%
F-measure	0.940	0.793
Average power consumption	1.2%	2.3%
Average CPU consumption	0.6%	0.8%
Average memory consumption	6.54 MB	7.73 MB

Table 10: Comparison of the proposed method with Shabtai's method.

	Proposed method	Shabtai's method
Accuracy	96.9%	77.1%
F-measure	0.940	0.695
Average power consumption	1.2%	7.7%
Average CPU consumption	0.6%	2.3%
Average memory consumption	6.54 MB	9.37 MB

The paradigms of edge computing are used to detect Android malware in this study. However, several challenges would be encountered when deploying it in practice: First, the mobile devices must install a specified app for the collection of essential information, which may cause concerns and mobile users may refuse to cooperate. Second, in edge computing, the data should be preprocessed by the edge to better protect the users' privacy. However, the edge server can observe all the data (including the sensitive information), and it threatens the privacy of users. For example, if the edge is deployed by the cloud provider, the users' privacy information is also available to the cloud. There is no difference with the user-cloud model for privacy protection unless the edge server is deployed by the user. Third, the edge server should be well protected from network attacks. Otherwise, it will be convenient for hackers to steal users' private information.

8. Conclusion

In this paper, we propose a novel method for detecting Android malware based on edge computing. To the best of our knowledge, the detection of Android malware by utilizing edge computing and traffic clustering has not been considered previously. By introducing the edge layer, the main tasks of the mobile device are offloaded to the edge server, and the mobile device only needs to record flow metainformation, which substantially conserves the resources of the mobile device. The edge server extracts apps' network traffic features and sends these features to the cloud platform for further analysis. Thus, the users' privacy can be better protected. With the received features, the cloud calculates the similarities between apps and clusters these similarity values to separate the original apps and the malware automatically. We evaluated our methods on 400 Android apps, and the experimental results demonstrate that the detection accuracy can reach 100% for several apps, and the average accuracy is 96.9%. We also tested the performance of the proposed method and compared it with the typical approaches. The comparison results show that our method can detect repackaged Android malware with both higher detection accuracy and lower power and resource consumptions. In the future, we will continue to work to overcome the discussed limitations and to evaluate the proposed method in real world scenarios.

Data Availability

The downloaded apps used to support the findings of this study have been deposited in https://pan.baidu.com/s/1b8vMceIdtdd1gBRC5Gbzvw. Its extraction code is vd5r. The clustering code is also in the repository. Since app network traffic contains users' sensitive information, it cannot be publicly shared. However, we have explained all the steps to generate network traffic automatically. These steps are also described in the instructions.pdf in the repository.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by National Key Technologies Research and Development Program of China under Grant 2018YFD0401404, National Natural Science Foundation of China under Grants 61702282, 61802192, and 71801123, Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grants 18KJB520024 and 17KJB520023, Nanjing Forestry University under Grants GXL016 and CX2016026, and NUPTSF under Grant NY217143.

References

[1] Statista, *Mobile Operating Systems' Market Share Worldwide from January 2012 to December 2019*, Statista, Hamburg, Germany, 2009, https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[2] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[3] https://www.appbrain.com/stats/number-of-android-apps, 2020.

[4] 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.

[5] M. Fan, J. Liu, X. Luo et al., "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[6] J. Zhang, Z. Qin, K. Zhang, H. Yin, and J. Zou, "Dalvik opcode graph based android malware variants detection using global topology features," *IEEE Access*, vol. 6, pp. 51964–51974, 2018.

[7] S. Garg, S. K. Peddoju, and A. K. Sarje, "Network-based detection of android malicious apps," *International Journal of Information Security*, vol. 16, no. 4, pp. 385–400, 2016.

[8] 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.

[9] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "pBMDS: a behavior-based malware detection system for cellphone devices," in *Proceedings of the Third ACM Conference on Wireless Network Security*, ACM, New York, NY, USA, pp. 37–48, 2010.

[10] 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*, ACM, Chicago, IL, USA, pp. 15–26, October 2011.

[11] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "MADAM: a multi-level anomaly detector for android malware," in *Proceedings of the International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, vol. 12, Springer, St. Petersburg, Russia, pp. 240–253, 2012.

[12] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ACM, Berlin, Germany, pp. 611–622, November 2013.

[13] S. Wang, Z. Chen, X. Li, L. Wang, K. Ji, and C. Zhao, "Android malware clustering analysis on network-level behavior," in *Proceedings of the International Conference on Intelligent Computing*, Springer, Liverpool, UK, pp. 796–807, August 2017.

[14] G. He, B. Xu, and H. Zhu, "AppFA: a novel approach to detect malicious android applications on the network," in *Security and Communication Networks*, Wiley, Hoboken, NJ, USA, 2018.

[15] X. Wang, Y. Yang, and Y. Zeng, "Accurate mobile malware detection and classification in the cloud," *SpringerPlus*, vol. 4, no. 1, p. 583, 2015.

[16] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 76, 2017.

[17] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: a user-oriented behavior-based malware variants detection system for android," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1103–1112, 2017.

[18] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pp. 447–458, Kyoto, Japan, June 2014.

[19] Y. Duan, M. Zhang, A. V. Bhaskar et al., "Things you may not know about android (Un) packers: a systematic study based on whole-system emulation," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, pp. 1–15, San Diego, CA, USA, February 2018.

[20] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, Buenos Aires Argentina, pp. 358–369, May 2017.

[21] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE, San Francisco, CA, USA, pp. 95–109, 2012.

[22] N. W. Lo, S. K. Lu, and Y. H. Chuang, "A framework for third party android marketplaces to identify repackaged apps," in *Proceedings of the 2016 IEEE 14th International Conference on Dependable, Autonomic and Secure Computing, 14th International Conference on Pervasive Intelligence and Computing, 2nd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, pp. 475–482, Auckland, New Zealand, August 2016.

[23] L. Li, J. Gao, M. Hurier et al., "Androzoo++: collecting millions of android apps and their metadata for the research community," 2017, https://arxiv.org/abs/1709.05281.

[24] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[25] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smart-phone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ACM, Antonio, TX, USA, pp. 317–326, February 2012.

[26] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACM, New Orleans, LA, USA, pp. 56–65, December 2014.

[27] K. Chen, P. Wang, Y. Lee et al., "Finding unknown malice in 10 seconds: mass vetting for new threats at the google-play scale," in *Proceedings of the USENIX Security Symposium*, vol. 15, Washington, DC, USA, August 2015.

[28] Z. Wang, C. Li, Y. Guan, and Y. Xue, "Droidchain: a novel malware detection method for android based on behavior chain," in *Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS)*, IEEE, Florence, Italy, pp. 727-728, September 2015.

[29] K. Tian, D. D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 64–77, 2017.

[30] A. De Lorenzo, F. Martinelli, E. Medvet, F. Mercaldo, and A. Santone, "Visualizing the outcome of dynamic analysis of android malware with vizmal," *Journal of Information Security and Applications*, vol. 50, Article ID 102423, 2020.

[31] M. Lin, D. Zhang, X. Su, and T. Yu, "Effective and scalable repackaged application detection based on user interface," in *Proceedings of the 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, Atlanta, GA, USA, May 2017.

[32] G. Meng, M. Patrick, Y. Xue, Y. Liu, and J. Zhang, "Securing android app markets via modelling and predicting malware spread between markets," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 7, pp. 1944–1959, 2018.

[33] A. Arora and S. K. Peddoju, "Minimizing network traffic features for android mobile malware detection," in *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ACM, Hyderabad, India, January 2017.

[34] X. Wu, D. Zhang, X. Su, and W. Li, "Detect repackaged Android application based on HTTP traffic similarity http traffic similarity," *Security and Communication Networks*, vol. 8, no. 13, pp. 2257–2266, 2015.

[35] J. Malik and R. Kaushal, "Credroid: android malware detection by network traffic analysis," in *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, ACM, Paderborn, Germany, pp. 28–36, July 2016.

[36] A. Zulkifli, I. R. A. Hamid, W. M. Shah, and Z. Abdullah, "Android malware detection based on network traffic using decision tree algorithm," in *Proceedings of the International Conference on Soft Computing and Data Mining*, Springer, Senai, Malaysia, pp. 485–494, January 2018.

[37] Z. Chen, Q. Yan, H. Han et al., "Machine learning based mobile malware detection using highly imbalanced network traffic," *Information Sciences*, vol. 433-434, pp. 346–364, 2018.

[38] G. He, B. Xu, L. Zhang, and H. Zhu, "Mobile app identification for encrypted network flows by traffic correlation," *International Journal of Distributed Sensor Networks*, vol. 14, no. 12, pp. 1–17, 2018.

[39] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: towards on-device non-invasive mobile malware analysis for ART," in *Proceedings of the 26th USENIX Security Symposium*, pp. 289–306, Vancouver, Canada, August 2017.

[40] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, Article ID 101663, 2020.

[41] 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)*, IEEE, Istanbul, Turkey, pp. 4348–4351, August 2010.

[42] M. Zhao, T. Zhang, F. Ge, and Z. Yuan, "Robotdroid: a lightweight malware detection framework on smartphones," *Journal of Networks*, vol. 7, no. 4, p. 715, 2012.

[43] K. A. Talha, D. I. Alper, and C. Aydin, "APK auditor: permission-based android malware detection system," *Digital Investigation*, vol. 13, pp. 1–14, 2015.

[44] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu, "Samadroid: a novel 3-level hybrid malware detection model for android operating system," *IEEE Access*, vol. 6, pp. 4321–4339, 2018.

[45] G. He, L. Zhang, B. Xu, and H. Zhu, "Detecting repackaged android malware based on mobile edge computing," in *Proceedings of the 2018 Sixth International Conference on Advanced Cloud and Big Data (CBD)*, IEEE, Lanzhou, China, pp. 360–365, August 2018.

[46] R. Roman, J. Lopez, and M. Mambo, "Mobile edge computing, Fog et al.: a survey and analysis of security threats and challenges," *Future Generation Computer Systems*, vol. 78, pp. 680–698, 2018.

[47] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ACM, Helsinki, Finland, pp. 13–16, August 2012.

[48] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: a taxonomy," in *Proceedings of the Sixth International Conference on Advances in Future Internet*, pp. 48–55, Lisbon, Portugal, November 2014.

[49] P. Mach and Z. Becvar, "Mobile edge computing: a survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[50] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[51] X. Lyu, H. Tian, L. Jiang et al., "Selective offloading in mobile edge computing for the green internet of things," *IEEE Network*, vol. 32, no. 1, pp. 54–60, 2018.

[52] S. Agrawal and J. Agrawal, "Survey on anomaly detection using data mining techniques," *Procedia Computer Science*, vol. 60, pp. 708–713, 2015.

[53] A. Tongaonkar, "A look at the mobile app identification landscape," *IEEE Internet Computing*, vol. 20, no. 4, pp. 9–15, 2016.

[54] G. He, B. Xu, and H. Zhu, "Identifying mobile applications for encrypted network traffic," in *Proceedings of the 2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, IEEE, Shanghai, China, pp. 279–284, August 2017.

[55] G. Ranjan, A. Tongaonkar, and R. Torres, "Approximate matching of persistent lexicon using search-engines for classifying mobile app traffic," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*, IIEEE, San Francisco, CA, USA, April 2016.

[56] L. Li and S. Qu, "Short text classification based on improved ITC," *Journal of Computer and Communications*, vol. 1, no. 4, pp. 22–27, 2013.

[57] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark android malware datasets and classification," in *Proceedings of the 2018 International Carnahan Conference on Security Technology (ICCST)*, IEEE, Montreal, Canada, October 2018.

[58] A. Rodriguez and A. Laio, "Clustering by fast search and find of density peaks," *Science*, vol. 344, no. 6191, pp. 1492–1496, 2014.

[59] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight UI-guided test input generator for android," in *Proceedings of the Software Engineering Companion (ICSE-C)*, pp. 23–26, Buenos Aires, Argentina, May 2017.

[60] M. Fan, X. Luo, J. Liu et al., "Graph embedding based familial analysis of android malware using unsupervised learning," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, Piscataway, NJ, USA, pp. 771–782, May 2019.