Knowl Inf Syst (2018) 55:771–796
https://doi.org/10.1007/s10115-017-1111-8

REGULAR PAPER

Mobile Apps identification based on network flows

Georgi Ajaeiya[1] · Imad H. Elhajj[1] · Ali Chehab[1] · Ayman Kayssi[1] · Marc Kneppers[2]

Received: 26 August 2016 / Revised: 26 August 2016 / Accepted: 21 September 2017 /
Published online: 30 September 2017

Abstract Network operators and mobile carriers are facing serious security challenges caused by an increasing number of services provided by smartphone Apps. For example, Android OS has more than 1 million Apps in stores. Hence, network administrators tend to adopt strict policies to secure their infrastructure. The aim of this study is to propose an efficient framework that has a classification component based on traffic analysis of Android Apps. The framework differs from other proposed studies by focusing on identifying Apps traffic from a network perspective without introducing any overhead on subscribers smartphones. Additionally, it involves a technique for pre-processing network flows generated by Apps to acquire a set of features that are used to build an identification model using machine learning algorithms. The classification model is built using classification ensembles. A group of chosen users contribute in training the classification model, which learns the normal behavior of selected Apps. Eventually, the model should be able to detect abnormal behavior of similar Apps across the network. A 93.78% classification accuracy is achieved with a low false positive rate under 0.5%. In addition, the framework is able to detect abnormal flows of unknown classes by implementing an outlier detection mechanism and reported a 94% accuracy.

Keywords Android security · Traffic analysis · App profiling · Flow-based classification

1 Introduction

According to IDC [1], smartphones market share grew 13% over the second quarter of 2015. Android dominated the market with an 82.8% share. Android market share grew

✉ Georgi Ajaeiya
gaa39@mail.aub.edu

1 Department of Electrical and Computer Engineering, American University of Beirut, Beirut 1107 2020, Lebanon

2 TELUS Corp, Vancouver, Canada

from Q2 of 2012 till Q2 of 2015 by 13.5% indicating an increase in the number of users and Apps [1]. On the other hand, this growth presented new challenges to network operators. Furthermore, Bring Your Own Device (BYOD) concept raised serious challenges for network administrators [18] who are concerned about their networks ability to face new emerging attacks on their infrastructure by exploiting smartphones' vulnerabilities, such is the case with Distributed Denial of Service (DDoS) botnets' attacks [25].

Accordingly, many solutions have been proposed in order to protect Android platform against emerging attacks. For example, anti-virus and anti-malware Apps apply signature-based identification of known malware by examining files, memory, and system settings. These solutions turned out to be ineffective for mobile platforms because anti-malware Apps have the same access privileges as other Apps running on the phone. Since these solutions are signature based, they require continuous update of malware definitions, which consumes storage and processing resources. However, in a recent work, the authors of [22] suggested a framework which can be used to detect new malware types and update malware signatures using Active Learning (AL) and Support Vector Machines (SVM). The authors suggested deploying the framework in strategic network nodes such as Apps' market servers. Some proposed solutions focused on static and offline analysis of Apps' source code to detect plagiarism and changes in code such as what was proposed by [23] Other types of solutions such as access control solutions have been adopted in BYOD and implemented through Mobile Device Management (MDM). These solutions act at the App level by placing restrictions on Apps usage, or at the file level by using containers to limit the scope of data leakage. BYOD is implemented via tools installed on the device that limit users access to the phones resources in order to minimize the damage of exploited Apps. Data leakage prevention solutions depend on expecting the content of exchanged data traffic. One example is payload inspection by applying Deep Packet Inspection (DPI) [11]. Automated App analysis solutions have been proposed in the literature. For example, automated analysis of APK files extracts static information such as requested permissions then categorizes Apps based on the extracted info [14]. Comparing monitored security specifications of Android Apps against their manifest file is another example [31]. Other forms of automated App analysis use intrusion detection to statistically analyze Apps data and traffic flows, and compare it against the expected behavior [33].

In this paper, a classification component is introduced to identify Android Apps using traffic flow analysis. This component can be used in Cyber Threat Management (CTM) frameworks. The component can deliver valuable insights about Apps running across the network. Machine learning algorithms and classification ensembles are used to classify samples of extracted Apps' flows. This component can be more effective in defending against new emerging attacks since it can detect behavioral changes and operate in real time. The proposed framework has the following contributions:

- Employ new aggregated network features
- Pre-processing, analyzing, and classifying extracted feature vectors on the network side to save phone resources
- Using statistical analysis in classification and decision making, which allows for the handling of numeric network features and adapting to normal changes
- Does not require DPI nor rooting of the device

The rest of the paper is organized as follows. Section 2 includes a review of the related work in the literature. In Sect. 3, classification in machine learning is briefly viewed. Section 4 describes how the proposed framework works. Sections 5 and 6 show the experimental setup and results analysis. Finally, we conclude and discuss the future work in Sect. 7.

2 Related work

There are many proposed solutions in the literature related to Android security, but only few target App identification using network behavior profiling and traffic analysis. The authors of [37] presented a multilayer system for profiling Android Apps. The system includes four layers: Static, User interaction, OS, and Network layers. The main idea is taking advantage of cross-layer analysis to detect invisible abnormalities from a single-layer perspective. However, root privileges are required to analyze Apps events and network behavior. The authors of [11] proposed App identification based on DPI. It helps network operators to expect traffic loads, quality of service, and discover network abnormalities. However, their approach supposes that 70% of Apps do not use HTTPS which is not accurate for current Apps in stores. Additionally, applying DPI on the device consumes CPU and battery resources.

Behavioral analysis of Apps is another field of interest in App automated analysis. By concentrating on the analyzed behavior, we can split the proposed solutions to general behavior analysis and network behavior analysis. Andromaly [32] and Crowdroid [6] are solutions based on general behavior analysis. Andromaly is a behavioral malware detection framework that consists of real-time monitoring, collection, pre-processing, and analysis of Android system metrics (Features). These features are extracted from both kernel and App levels using a client running on the device. They capture aspects such as network activities, resource consumption (e.g., memory and CPU), and event occurrence (e.g., touch screen and keyboard). Crowdroid uses a crowdsourcing dynamic analysis method to detect Android-platform malware. The system collects samples of execution traces for the running Apps. These traces help in differentiating between benign and malicious Apps. However, both solutions require rooted devices to collect some of the features.

Network behavior analysis solutions concentrate more on features related to network activity. The authors of [33] presented a hybrid behavioral-based anomaly detection system, which has a client–server architecture. The system is designed to protect mobile users and network infrastructure by detecting deviations in Apps network behavior. Models are generated to represent the normal network behavior of each installed App. The models are based on network related features, which are collected using a client running on the smartphone. The authors of [26] proposed a system for network behavior detection of android malware, which consists of three parts: monitoring, anomaly analysis, and cloud storage model. The system monitors Apps network behavior in real time and does not need to parse the content of exchanged packets, which protects users privacy. It depends on network behavior features only such as received bytes, sent bytes, and connection length of the running processes. However, the reported results have shown low accuracy rates.

App traffic analysis solutions were adopted, but at a smaller scale compared to other solutions. The authors of [13] presented a detailed analysis of Android smartphones traffic to show the effect on power consumption and throughput. They analyzed transfer sizes of (Transmission Control Protocol) TCP Flows and Round Trip Times, in addition to retransmission rates. The authors of [35] concentrated on modeling network traffic produced by users behavior. They introduced a session concept that represents the needed flows to complete a single task. Session types were characterized by session variables such as session lengths in seconds or bytes. Each type of traffic, such as media streaming or browsing, has defined a set of values for these variables. However, the values were based on assumptions and collected statistics, which may not represent all traffic types accurately.

The authors of [2] investigated background traffic generated by Android Apps. They confirmed traffic characteristics diversity by performing a detailed experiment to analyze the

traffic. The authors also studied persistent TCP-based Apps, which require periodic message exchange to keep connections alive. However, the study was restricted to background traffic only. They suggested using DPI to identify running Apps, which is impractical due to traffic encryption. The authors of [38] compared URLs in HTTP requests made by an Android App against a URL table to detect malicious Apps. The table contains a list of malicious servers. This approach is very limited because it depends only on logging HTTP requests, while most of current requests use HTTPS. Additionally, comparing against a static or dynamic table of URLs is useless since attackers can fake URLs.

To identify Apps, the authors of [8] suggested an adaptive algorithm that automatically recognizes traffic by relying on machine learning classifiers. They introduced a traffic classifier as a collection of rules that defines each type of traffic. The system architecture is composed of three sections: data collection, a flow capturing mechanism, and a classifier generation algorithm. The generated information from flow capturing and payload of non-encrypted flows are used to generate the classifiers. However, the method had drawbacks because it depends on destination IPs and known ports of servers which are not specified in Peer to Peer (P2P) communication. In addition, the method used non-encrypted flows to build the classifiers, which is not an effective method in case of encrypted traffic.

The authors of [17] proposed traffic anomaly recognition using SVM classification algorithms [10]. A detection model is built using collected features from the phone. Afterward, they evaluated the detection model using real malware. The system used network features, and applied statistical classification to detect malicious Apps. However, the features were not aggregated which could have helped in improving the detection accuracy.

The authors of [9] focused on analyzing encrypted traffic to build usage profiles and understand users actions. The proposed framework analyzes TCP packets and extracts information about network flows. The authors used Dynamic Time Wrapping (DTW) algorithm [4] to find alignments between incoming and outgoing packets, which turned out to be unique for each App. Their approach handled encrypted traffic and identified users actions with high accuracy; it did not involve a client installation on the device, and it did not require any rooting privileges. However, it was affected by noisy packets such as TCP retransmission packets and required reading TCP flags to set the start and end of the flow. In another work [34], the same authors introduced an App scanner framework that inspects encrypted traffic. This study introduced multiple methods close to the proposed work in this paper. However, their experimental setup did not involve real traffic generated from real users.

The authors of [19] combined statistical-based and behavioral-based detection. Network traffic attributes are represented by graphlets and packet sizes as distributions. The authors studied Apps background and foreground traffic separately. The proposed work achieved high accuracy in detecting Apps traffic patterns using a feature vector of 59 features. However, the experiment was limited because the traffic was generated from a single testing device.

In this paper, we introduce a framework that handles encrypted traffic flows efficiently. The framework classifies each flow according to the originating App. It differs from previous work in data pre-processing and flow feature extraction. In addition, the experiments involve real users interacting with multiple Apps.

3 Background

Before the framework is presented, we briefly introduce the various techniques used in this work.

3.1 Classification in machine learning

Classification is a form of supervised machine learning implemented using data mining techniques [15]. In supervised learning there are two datasets, one for training and the other for testing or evaluation. Every instance in both datasets is represented by a set of features that may be continuous, categorical, or binary [15] and have known labels or classes. Before a classification model is built and the learning process starts, there are three important phases that have to be implemented. These phases guarantee high detection accuracy. Data preprocessing is the first phase, where training datasets may contain missing values that need to be filled, or noisy points that need to be cleaned [15]. The second phase is feature selection, which is the process of removing redundant and useless features that may affect learning either by increasing the learning period, or decreasing the accuracy of classification. In most cases, classifiers use aggregated features from a set of basic features. The final phase is choosing a suitable learning algorithm to meet the objectives of the study. The feature selection phase sometimes precedes training the classification model if the features ranking algorithm uses specific metric linked to the type of the classification model. Supervised classification has a large number of data mining models, e.g., Decision Trees [27], and some statistical approaches, e.g., Bayesian classifiers [29].

3.2 Bagging decision trees

Introduced by Quinlan [27], a Decision Tree (DT) is a tree structure classification model produced by a supervised machine learning algorithm. It maps input features to a label that represents the predicted type of the data or the class described by these features. DT classifies instances based on their feature values. The tree structure is built starting from a root node down to the end nodes (leafs) as a binary tree. Each split node in the tree depends on the gain value of a specific feature calculated using Information Gain algorithm [20]. Finally, the leaf nodes at the bottom of the tree represent the classes (Labels). Information gain is based on the change of the entropy value for a feature after splitting the dataset using one of the features. There are two kinds of entropies, the first is calculated using the frequency table of a class attribute c, where the frequency p_i is the count of the distinct values of that attribute, as shown in Eq. 1.

$$E(S) = \sum_{i=1}^{c} -p_i \log_2(p_i) \tag{1}$$

The second entropy value is calculated using the frequency table of a feature against the class attribute c, as shown in Eq. 2.

$$E(T, X) = \sum_{c \in X} P(c)E(c) \tag{2}$$

Finally, the gain of each feature is calculated using both entropies, and the feature with the largest gain is chosen to split the dataset based on its values, as shown in Eq. 3.

$$\text{Gain}(T, X) = E(T) - E(T, X) \tag{3}$$

The Bagging Algorithm was introduced by Breiman [5]. The idea is to create a single classifier from multiple weaker classifiers. These classifiers generate their votes from multiple Bootstrap samples [3]. Generating a Bootstrap sample is done by uniformly sampling n instances from a single training dataset and replacing them [12]. Afterward, the T Bootstrap

samples are used in building T classifiers in parallel. Each classifier, C_i, is learned by Bootstrap sample, B_i. The output of the final classifier C is the class that is most voted for by $C_1, C_2, \ldots, C_T$ as shown in Algorithm 1. Each instance in the training set has a probability of $1-(1-1/n)^n$ to be selected in one of the n times instances picked from the single training set [3]. This technique decreases the error probability since it is including most instances in different combinations to build the classifier. In addition, it prevents creating an over fitted model of the system.

Where S is the set of all training samples, Inducer I is the training algorithm, and T is the number of bootstrap samples. The algorithm generates T bootstrapped groups of training samples $S^{'}$, and for each group it creates a classifier C_i using inducer I.

Quinlan tried bagging the DT Algorithm [28] and evaluated the model using diverse collections of datasets. The results confirmed higher accuracy for multiclass datasets generated from multiple sources. Additionally, he stated that using a group of dissimilar leaners to build a strong classifier will contribute to increasing the accuracy while keeping a generalized model of the system. Based on that, and since it requires a non-stable learning environment, we decided to use the Bagging Algorithm. The small changes in the non-stable environment will create different sub-classifiers [28].

Algorithm 1: The Bagging Algorithm

Input : set S, Inducer I, integer T
Output: classifier C^*
for $i = 1$ **to** T **do**
  $S^{'}$ = bootstrap sample from S (i.i.d. sample with replacement);
  $C_i = I(S^{'})$;
end
$C^*(x) = \text{argmin}_{y \in Y} \sum_{i:C_i(x)=y} 1$ (the most often predicted variable y);

4 Flow-based application identification

Figure 1 shows the overall process of building the framework's classification model. It will be used to classify flow samples from multiple Apps. Classification decisions are based on the contribution of the chosen set of users to train the classifier. This can provide intelli-

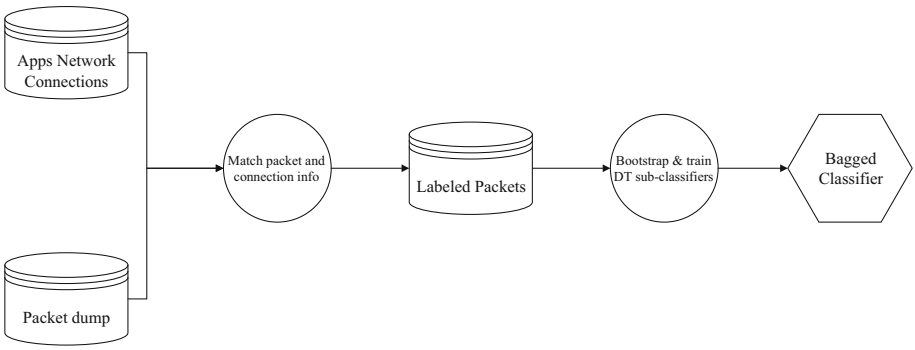

Fig. 1 Building framework's classifier

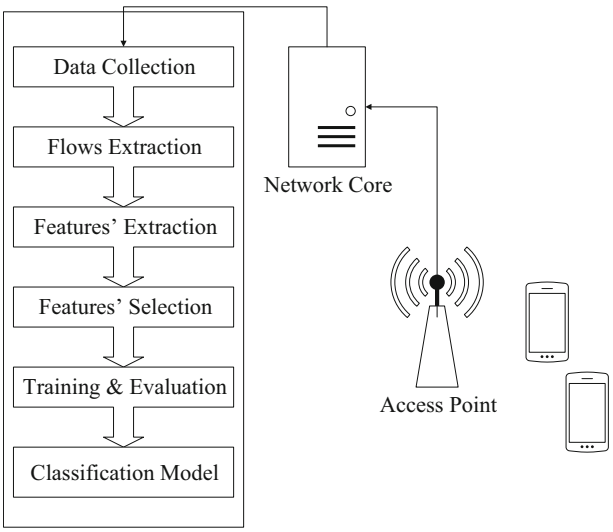

Fig. 2 App identification process

gence about Apps behavior. Setting a base profile for the network for multiple states can help administrators to optimize QoS. Additionally, it provides situational awareness, e.g., congestion durations and periods. The classifier is built using the Bagging Algorithm and includes multiple sub-classifiers which are DTs.

Figure 2 illustrates the overall process used in building the detection framework. Upon deployment, the App identification framework uses the classification model which is produced at the final stage. The process starts by extracting network features of flows generated by various Apps. A flow is the traffic exchanged between two IPs with the same ports and protocol during a time period. The flow ends when packet exchange turns idle for a certain amount of time and never fires back. Each flow consists of the exchanged packets during its lifetime. Feature extraction and aggregation is implemented over a specific time interval. During the time interval, the set of predefined features is measured and logged to be aggregated in a feature vector. Afterward, the set of aggregated feature vectors is filtered in a feature selection phase. Then, these feature vectors are used to build a classifier, which is the core of the system. The framework uses the classifier to identify Apps network behavior.

What distinguishes the proposed framework is the feature vector that can be extracted without acquiring special permissions or rooting. The framework has two main advantages. First, network users are not required to make any updates or install special software on their phones. Thus, customers will not lose phones warranty. Second, it will be easier for the network operator to distribute a light tool during the building phase on selected smartphones while having all the heavy processing and computations residing in the core.

Since feature extraction and aggregation is repeated at each time interval during the lifetime of the flow, the framework should be able to identify an App using just few exchanged packets. A flow can have multiple samples when applying a rapid sampling rate. The samples can be processed according to their arrival. Thus, the flow can be identified by analyzing headmost samples. Another advantage of this method is real-time identification, whereas the flow sample is classified at each time interval. Therefore, if any abnormal change occurs in the behavior, the system should be able to detect such changes. Identifying abnormalities in Apps

Table 1 Packet attributes

Packet number
Timestamp
Packet length
Packet inter-arrival time
Direction (in/out)
App label

behavior is based on outlier detection. The system depends on a confidence score, associated with each instance, to predict the outliers. It is expressed by the posterior probability of an instance belonging to a certain class. When the confidence score falls in a certain range, the system investigates the deviation for that instance. This method makes the system adapt smoothly to normal changes in Apps behavior while keeping the ability to detect abnormal changes.

The framework can be responsive to gradual changes by comparing the flow class at each time interval to older time intervals and track the changes in features values. Since the framework is running in the network core, real-time detection and adaptation to normal changes will not affect smartphones resources. It will not add any overhead that may affect users experience. The framework will even be transparent to the users because the classification is done on the network side without involving a detection client on the smartphone.

In the following subsections, we explain each phase in the framework and elaborate on the methods highlighting the contribution.

4.1 Flow extraction

This phase starts at the end of the data collection phase which will be shown in 5. Flows occurring during a certain time period are extracted from the packet dumps. A flow is represented by the exchanged packets between two IPs using same ports and protocol. The flow is terminated when traffic exchange is idle for a specified amount of time without firing back. Each flow packet is defined by a set of attributes as shown in Table 1. The inter-arrival time is the elapsed time between the current packet and last exchanged packet in the same flow. The idle time is a configurable parameter which is set as described in Sect. 6.1. Unlike other solutions, the flow extraction phase does not depend on reading TCP flags nor User Datagram Protocol (UDP) payload to identify the last exchanged packet. A flow in represents generated traffic when executing actions of an App. No payload information is used to calculate the features, which ensures users privacy.

4.2 Feature extraction and aggregation

The feature extraction and aggregation phase is implemented to get useful information that describe Apps traffic. Only statistically significant flows are taken into consideration. Such flows have a minimal length and amount of exchanged packets required to represent an App action. Several feature extraction methods were proposed in the literature, and they were reflected in this framework. The features are extracted for each flow at every time interval. Each flow may have more than one sample generated in this phase.

An advantage of this technique is having multiple measurements for the same flow at formal intervals. These measurements can be used in real-time detection of abnormalities in flow behavior. Additionally, setting a relatively short interval allows the process to adapt

smoothly to normal changes. Changes may occur during the lifetime of the same flow or other flows from the same App. Flow minimal length, and the time interval are all parameters of the framework. There are no standards that suggest values for these parameters. Therefore, they are set empirically in such a way to get best achievable results as we describe later on.

4.3 Building framework's classifier

The Bagged Trees classifier is a classification model that uses a voting approach to assign classes for evaluation and testing samples. It depends on creating an ensemble of classifiers using various mechanisms. One of the mechanisms is using multiple datasets for training with a single learning algorithm such as what was done in [16]. The Bagging Algorithm is used to create an ensemble of DTs. The classifier relies on the extracted feature vectors of the flow by examining features' values. Each time a new sample is classified; it represents a certain period of that flow since a sampling approach is used by taking multiple measurements for the flow.

4.4 Feature selection

Since Bagged Trees are adopted to build the framework's classifier, Information Gain algorithm is used to calculate the entropy of each feature. The entropy is used to measure the strength of the feature as shown in Sect. 3.2. The strategy is to choose features with the highest gain or entropy measured over all the feature vectors in the training set. Therefore, we can set a threshold for the gain and choose all the features that have a gain greater than the threshold. Afterward, the classifier can be re-evaluated using the selected features. Using a minimal set of features increases the training performance, it decreases the training time and the computational complexity while preserving a high classification rate.

4.5 Outlier detection

In multiclass classification models, the classifier tries to distinguish a sample by categorizing it into one of the known classes. However, if the sample is not classified normally into any of the classes, it is considered an outlier [36]. Generally, classifiers give a prediction score to each sample, and based on that score the sample will be classified. The score represents the confidence which a classifier has for that instance to be classified correctly. It is represented by the posterior probability of a sample belonging to a class. In Bagged Trees, the posterior probability of a class given a sample is a weighted average of the class posterior probabilities computed over selected trees in the ensemble (see Eq. 4).

$$\hat{P}_{bag}(c|x) = \frac{\sum_{t=1}^{T} \alpha_t \hat{P}_t(c|x)}{\sum_{t=1}^{T} \alpha_t} \tag{4}$$

where $\hat{P}_t(c|x)$ is the posterior probability of a class c given sample x using DT t and α_t is the weighted average factor for each tree. $\hat{P}_t(c|x)$ is the posterior probability of a class c given sample x in tree t as shown in Eq. 5.

$$\hat{P}_t(c|x) = \frac{P_t(x|c)P(c)}{P(x)} \tag{5}$$

where $P_t(x|c)$ is the number of samples that have ended at the same leaf node l and classified in class c divided by the total number of samples in class c, $P(c)$ is the prior of class c which

is the total number of samples from class c divided by the total number of samples, and $P(x)$ is defined as shown in Eq. 6, where K is the total number of classes.

$$P(x) = \sum_{c=1}^{K} P(c) P_t(x|c) \tag{6}$$

Accordingly, if the confidence score is low for all the classes, the instance is considered an outlier. Another technique adopted in Bagged Trees to detect outliers is the outlier score. The authors of [39] used the proximity measure in Random Forests (RF) to calculate the outlier of an instance. In general, RF is an ensemble of Trees. However, feature selection is done randomly at each level to build each tree. The proximity of an instance to a class is the average fraction of trees in the bagged ensemble for which the instance lands on the same leaf versus other instances in the same class as shown in Eq. 7.

$$P^{-}(n) = \sum_{k \in classj}^{max} (prox^2(n,k))/N \tag{7}$$

The outlier measure is identified as the inverse of the average proximity as shown in Eq. 8.

$$O^{-}(n) = \frac{N}{P^{-}(n)} \tag{8}$$

In the proposed framework, confidence and outlier scores are used to detect outliers. The outliers could be flow patterns generated from the same App or by other Apps. Therefore, the framework should be able to differentiate between both cases. Making the right decision about an outlier will drive the framework to either adapt to the normal changes in an App behavior, or detect the abnormal behavior and act accordingly. Both scores are configurable parameters in the framework and are set by experimental decisions. Calculating the proximity for all classified samples is computationally expensive. Therefore, the proximity is calculated only for samples with low confidence scores. Samples with high proximities and low confidence scores are considered outliers from a known class. On the other hand, instances with low proximities and low confidence scores are considered outliers from unknown classes that might be malicious. Using this technique the framework would be able to detect abnormalities in traffic flows. However, classifying these abnormalities as malicious or non-malicious is part of future work.

5 Experimental setup

Using the experimental setup shown in Fig. 3, we study the behavior of 6 mostly downloaded Apps as listed in Table 2. Android smartphones running Lollipop 5.1 are used in the experiment. Nine participants took part in the experiment and they interacted with each App for an hour using their personal accounts when needed. To collect traffic, a virtual access point is created on a PC connected to the Internet through a cable. The PC is configured to forward the traffic to the Internet through the wired network. Simultaneously, Wireshark is configured to run and collect the traffic into packet dumps. To label the packets, the connections that are generated by each App are logged using Network connections which is a tool that runs on Android and logs connections established or received by running Apps. The logs are used to filter packet dumps and extract Apps traffic.

First, the setup performs packet dumping to create a training dataset, which will be used to train the classifier. Traffic dumps are pre-processed to include flows that represent each

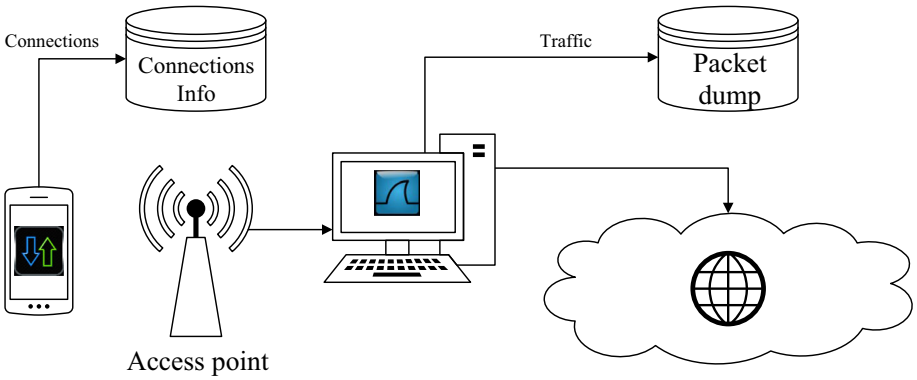

Fig. 3 Experimental setup

Table 2 Experimental apps

#	App name	Version	Category
1	Facebook	64.0.0	Interactive browsing
2	8 Ball Pool	3.5.0	Games
3	Skype	6.22.0	Video calling
4	Viber	5.8.0	VOIP
5	WhatsApp	2.12.45	Text & Multimedia messaging
6	YouTube	11.04.56	Video streaming

App. Established and received network connections of each App are logged using Network Connections, and the logs are sent to the network core to label the collected packets. This technique is used in the training phase only while the detection phase does not involve any tool installation. Privacy concerns may be raised during this phase. However, most of the traffic generated by the Apps is encrypted, and no payload information is used to extract the features.

Note that traffic dumps are collected under similar network conditions. However, changes in network QoS may affect some features values, e.g., having a router problem anywhere in the network core. Therefore, in a real network setup we can perform data collection on multiple periods to include different measurements for the same features. This will help in obtaining the normal ranges of these features. Furthermore, we can choose a selected group of known benign users to contribute in creating the initial dataset, which will be considered as a reference profile for the running Apps on the network.

The experiment resulted in a total of 18,051 flows. Each flow had different measurements according to its length using an idle time of 5 s. Each flow measurement is represented by a set of extracted features. Figure 4a shows flow count for every App based on a 2 s sampling rate. The 5 s idle time and 2 s sampling rate are explained later.

For some Apps, there is a noticeable difference between the number of flows and the number of measurements. This is due to the varying flow length among the Apps. Apps with long flows had more measurements than Apps with short flows. Figure 5 shows the difference in flow length among the Apps as a box plot.

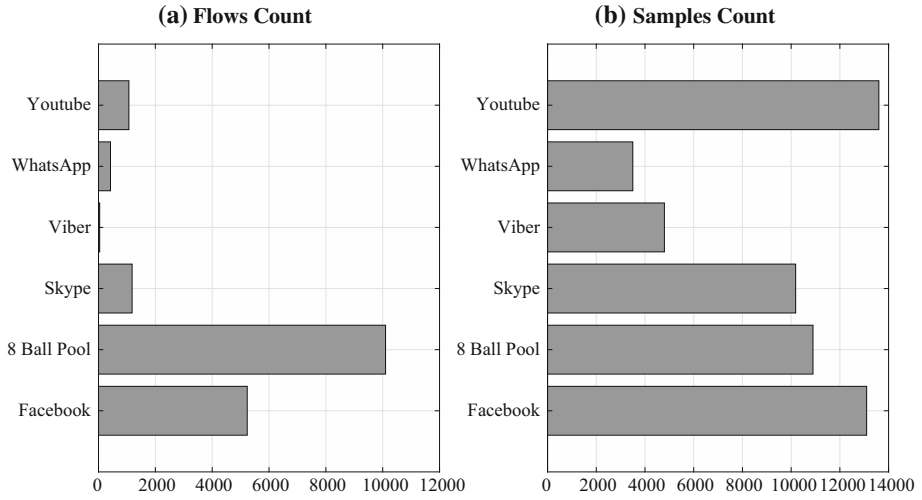

Fig. 4 Flow and samples count

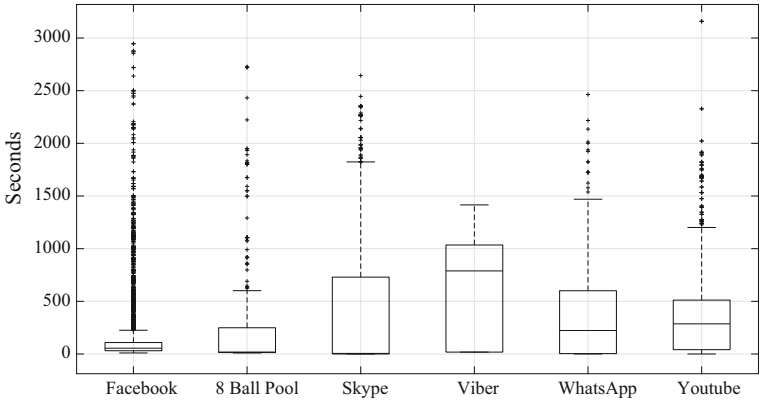

Fig. 5 Flow length in seconds

6 Results and analysis

6.1 Feature extraction

A 5 s idle time with a 2 s sampling rate are used to separate between the flows and take measurements. Since there is no criteria to set such values, we tried different options. For each value for the idle time and the sampling rate ranging from 1 to 10 by increments of 1, the 28 features listed in Table 3 are extracted for each sample of the flow. Using these samples, a DT classifier is trained. Then, the accuracy of the DT classifier is evaluated using fivefold cross-validation. At each evaluation iteration, the data which resulted from the feature extraction phase at specific values for the idle time and the sampling rate are used. Figure 6 shows the overall accuracy of the DT classifier for the chosen intervals. We can notice that a sampling interval of 2 s achieved high detection accuracy with a low false alarm

Table 3 Extracted and aggregated features

#	Feature
1	Packets Out Count
2	Packets In Count
3	Packets Out/Packets In ratio
4	Bytes Out Count
5	Bytes In Count
6	Bytes Out/Bytes In ratio
7	Average difference of Inter-arrival time of incoming packets
8	Average difference of Lengths of incoming packets
9	Average difference of Inter-arrival time of outgoing packets
10	Average difference of Lengths of outgoing packets
11	Median of Inter-arrival time of incoming packets
12	Median of Lengths of incoming packets
13	Median of Inter-arrival time of outgoing packets
14	Median of Lengths of outgoing packets
15	Variance of difference of Inter-arrival time of incoming packets
16	Variance of difference of Lengths of incoming packets
17	Variance of difference of Inter-arrival time of outgoing packets
18	Variance of difference of Lengths of outgoing packets
19	Average of Inter-arrival time of incoming packets
20	Average of Length of incoming packets
21	Average of Inter-arrival time of outgoing packets
22	Average of Length of outgoing packets
23	Variance of Inter-arrival time of incoming packets
24	Variance of Lengths of incoming packets
25	Variance of Inter-arrival time of outgoing packets
26	Variance of Lengths of outgoing packets
27	Inter-arrival Time between outgoing packets bursts
28	Inter-arrival Time between incoming packets bursts

rate compared to a 10 s sampling interval. Thus, it is chosen to extract flows' samples to train and evaluate the classification model.

The classification accuracy is evaluated using the True Positive Rate (TPR) versus the False Positive Rate (FPR) and the Receiver Operating Characteristics (ROC) space, where the aim is to increase the TPR and decrease the FPR. The accuracy at this stage refers to the accuracy of the DT classifier. Additionally, we computed the F1 score of the classification model, which is expressed in Eq. 9.

$$F1 = \frac{2TP}{(2TP + FP + FN)} \tag{9}$$

where TP is the True Positive count, FP and FN are the False Positive and False Negative counts, respectively. The F1 score is computed for each class; then, we take the average to express the overall score.

We can notice that this phase resulted a total of 56081 samples which forms an imbalanced dataset as shown in Fig. 4b. This is normal since some Apps generate more traffic

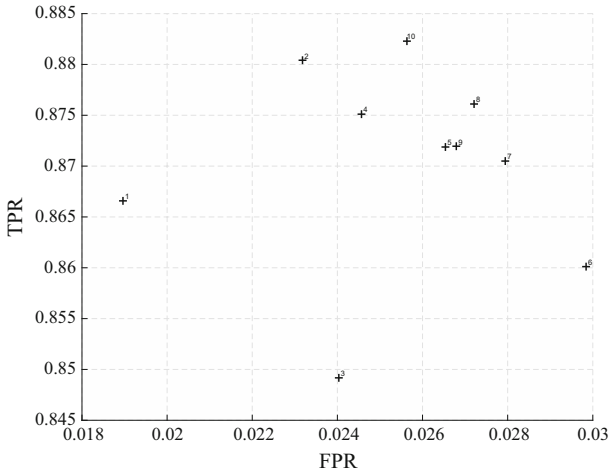

Fig. 6 Sampling interval accuracy

than others. For example, the messaging App, WhatsApp, had less traffic exchanged than the video streaming App YouTube. There are two general approaches to deal with such cases. The first approach is sensitive learning, and the second one is to use sampling techniques with classification ensembles [7]. SVM, for example, handles such datasets with class-weighted classification, which is a form of sensitive learning. It changes the misclassification penalty per class. Minority classes classification penalty would be chosen to be larger than majority classes [24]. Other studies on neural networks showed that the performance over an imbalanced dataset depends on how well the classes are separated [21]. Since the proposed approach depends on taking samples for each flow at a fixed interval, we follow the classification ensembles method. As such, we have to define a template model to be used in building the classification ensemble.

6.2 Training the classification model

6.2.1 A comparison among supervised classification models

To verify Bagged Trees selection as a classification model for the proposed framework a comparison is performed. The aim of the comparison is to show that Bagged Trees outperforms two of the most widely used machine learning algorithms which are SVM and artificial neural networks (ANN).

Table 4 shows the hyperparameters that were chosen for each classifier. The values of each model's hyperparameters are set to prevent overfitting the training data based on some experimental knowledge. The models are evaluated using fivefold cross-validation. Two approaches are followed to evaluate the classifiers. The first approach is using an imbalanced dataset. The second approach is extracting a balanced dataset by taking an equal number of samples for each class (App).

Table 5 shows the evaluation results for both imbalanced and balanced datasets. The Average Accuracy and F1 score for the 6 classes are reported. In the first dataset, samples' ratios are different among the classes. We can notice that the Bagged Trees model had the highest accuracy over the imbalanced dataset. Moving to the second approach, we can also

Table 4 Models' hyperparameters

Model	Parameters
SVM	Kernel = Polynomial
	Polynomial order = 2
	Kernel scale = auto
	Box constraint = 1
ANN	Hidden layers = 3
	Neurons count = 30
	Penalty function = cross entropy
	Learn function = gradient descent
	$\alpha = 0.1$
Bagged trees	T = 50
	Max. number of splits = 20

Table 5 Classification models evaluation results

Model	Imbalanced dataset		Balanced dataset	
	F1 (%)	Accuracy	F1 (%)	Accuracy (%)
SVM	86.02	87.85	86.65	87.28
ANN	79.59	80.52%	80.36	82.06
Bagged trees	93.78	94.02%	94.06	94.80

notice that SVM and ANN reported a better evaluation accuracy. However, Bagged Trees reported a higher accuracy outperforming both SVM and ANN.

As stated in Sect. 3.2, the Bagging Algorithm builds Trees from Bootstrap samples formed by uniformly distributing instances from the training set and replacing them. Thus, the grown Trees are based on small balanced datasets. Following this approach allowed to evade the complexity of sensitive learning. Additionally, the majority voting technique increases the prediction accuracy and prevents the model from over fitting the training data. This explains why Bagged Trees outperformed SVM and ANN in both evaluation approaches.

6.2.2 Evaluating bagged trees classification model

Bagged Trees model is evaluated using two approaches. The first approach is K-folds cross-validation, where K is chosen to be 5. All samples in the dataset are used to train and cross-validated a Bagged Trees classifier by fivefold using MATLAB Statistics and Machine Learning Toolbox. The hyperparameters are defined as shown in Table 4. Figure 7 shows the ROC curve for a fivefold cross-validation for all the classes (Apps). The reported average overall accuracy for the cross-validation is around 94%.

Table 6 shows the confusion matrix of the 6 Apps classes. We can notice a minimal ratio of misclassified samples for all classes. Some errors are due to a action between Apps types, e.g., browsing for video on Youtube, and browsing content on Facebook. Apps with common actions would generate similar flows which are classified in a single class randomly. Other errors are simply outliers generated by the background actions of Apps. These outliers are mostly filtered when picking flows that have statistically significant number of packets, but some of the outliers remain in the dataset. However, we can notice that Bagged Trees keeps a high detection accuracy of 93.78% measured by the $F1$ score.

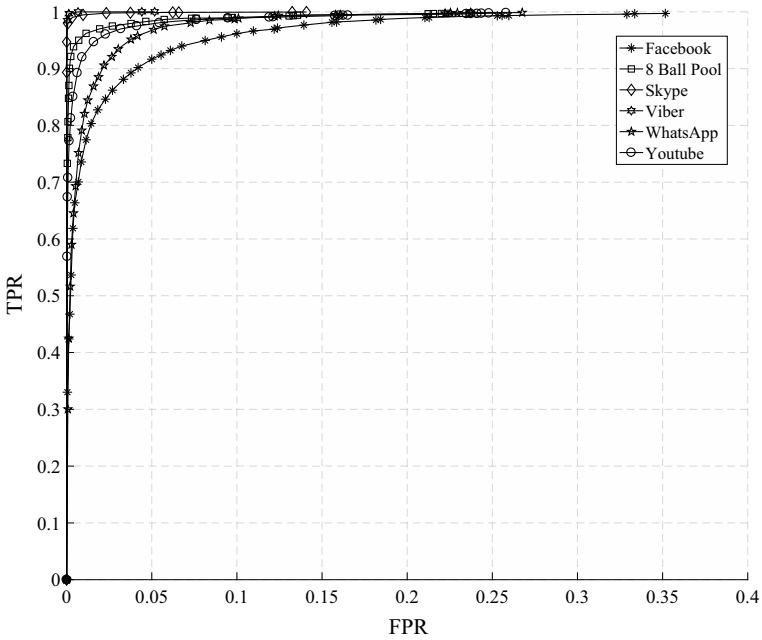

Fig. 7 ROC curve for fivefold cross-validation

Table 6 Confusion matrix (samples ratio %)

Class	Facebook	8-Ball	Skype	Viber	WhatsApp	Youtube
Facebook	87.10	1.42	0.04	0	7.69	3.75
8-Ball	2.79	94.85	0.02	0	0.90	1.44
Skype	0.35	0	98.25	0.46	0.92	0.02
Viber	0.06	0	0.27	99.65	0.02	0
WhatsApp	8.86	0.74	0.06	0.09	89.31	0.94
Youtube	4.88	0.96	0.02	0	0.71	93.43

The second evaluation approach is used to prove that the classification model is robust against introducing new flows which do not have samples in the training set. The classification model's accuracy is evaluated by folding over the experiment's users. The evaluation is implemented by iterating for 100 times, and at each iteration the users are split randomly into two groups. Using the first group we form the training set, and the second group forms the testing set. Then, using the training and testing sets, a Bagged Trees classifier is trained and evaluated using the hold out approach. In this approach, the training samples are picked from the training set only, and the testing samples are picked from the testing set to form a new dataset that has 70% of training samples and 30% of testing samples. Figure 8 shows the density function of the F1 score for all 100 iterations. We can notice that the F1 score averages around 91% which shows that the classification model is able to classify unknown flows. This evaluation approach shows that the classification model's accuracy is not associated with users count or type. At each iteration the users were split randomly into two groups with preserving the 70:30 ratio for training and testing samples.

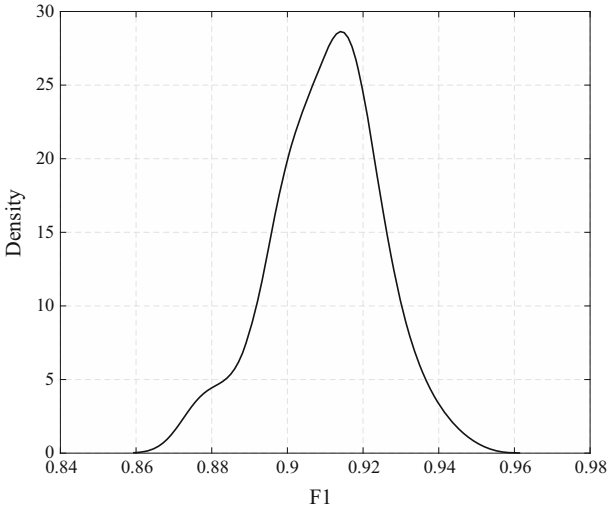

Fig. 8 F1 density for 100 evaluation iterations

To lower the computational complexity and classification time, the Bagged classifier is pruned and the number of trees is decreased. Using all samples in the dataset, we trained multiple Bagged classifiers while decreasing the number of trees. Figure 9a shows the decrease in the accuracy of the fivefolds while decreasing the number of used trees. We can notice that the overall accuracy slightly decreased even when we reached 20 Trees. This slight decrease in the accuracy is accompanied with a remarkable increase in the time performance. The classification model which is decreased 60% in size is able to achieve 99% of the accuracy using 61% less time as shown in Fig. 9b.

6.3 Feature selection

Each sample in the dataset is expressed using the feature vector shown in Table 3. However, the general trend in machine learning is to decrease the size of the feature vector for better performance and to eliminate possible noisy features. Therefore, in this phase, we target a reduced set of features, which increases time performance and at the same time preserves the accuracy. The Information Gain algorithm which discussed in 3.2 is used to rank the original set of features. To rank the features, we use a balanced dataset that includes an equal number of samples from each class. Then, we split the balanced dataset into a training and tuning set that forms 70% of the dataset, and a testing set that forms the rest. The tuning set is used to rank the features using Information Gain algorithm. Figure 10 shows the normalized rankings of all features in the original feature vector. We can notice that some features do not introduce any gain at training phase, e.g., Feature #17. Other features have a low impact on the classification decision, e.g., Features #25 and #27. Thus, to eliminate weak features, we sorted all features according to their rankings in a descending order. Then, starting from the weakest feature, we eliminate the last feature in the list and evaluate a Bagged Trees classifier with fivefold cross-validation using the tuning set.

Figure 11b shows the decrease in the classification accuracy measured by the $F1$ score. We can notice that when the feature vector size is decreased to 9 features, the Bagged Classifier preserves 98% of the reported classification accuracy before reduction at 91.56%. Figure 11a

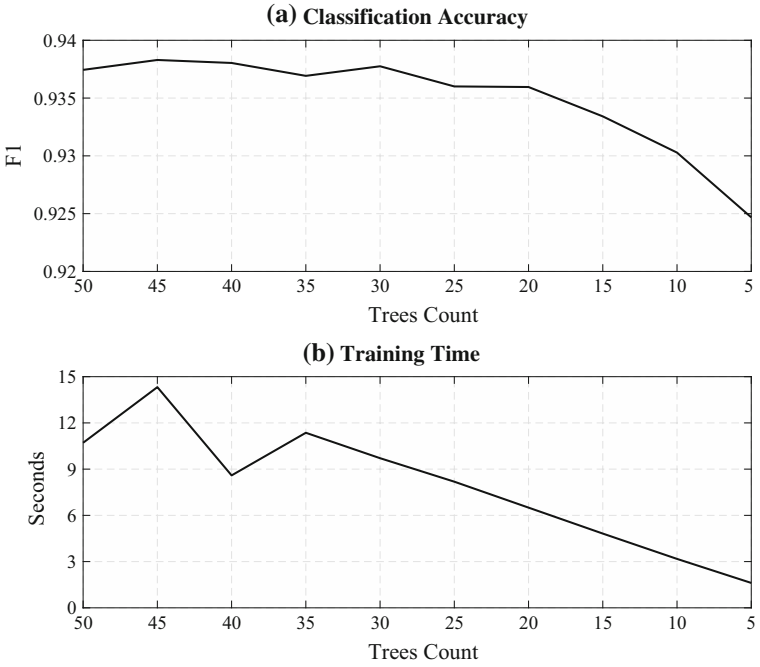

Fig. 9 Pruning the bagged classifier

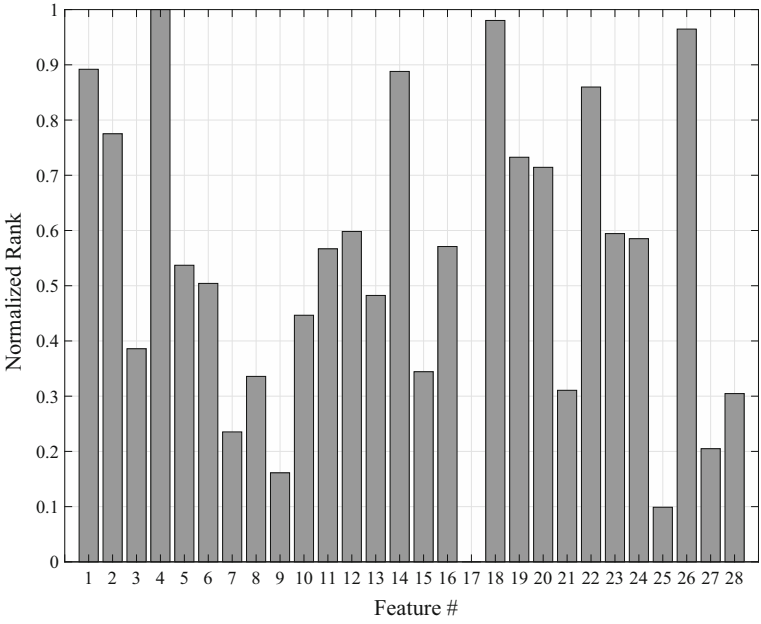

Fig. 10 Normalized features ranking

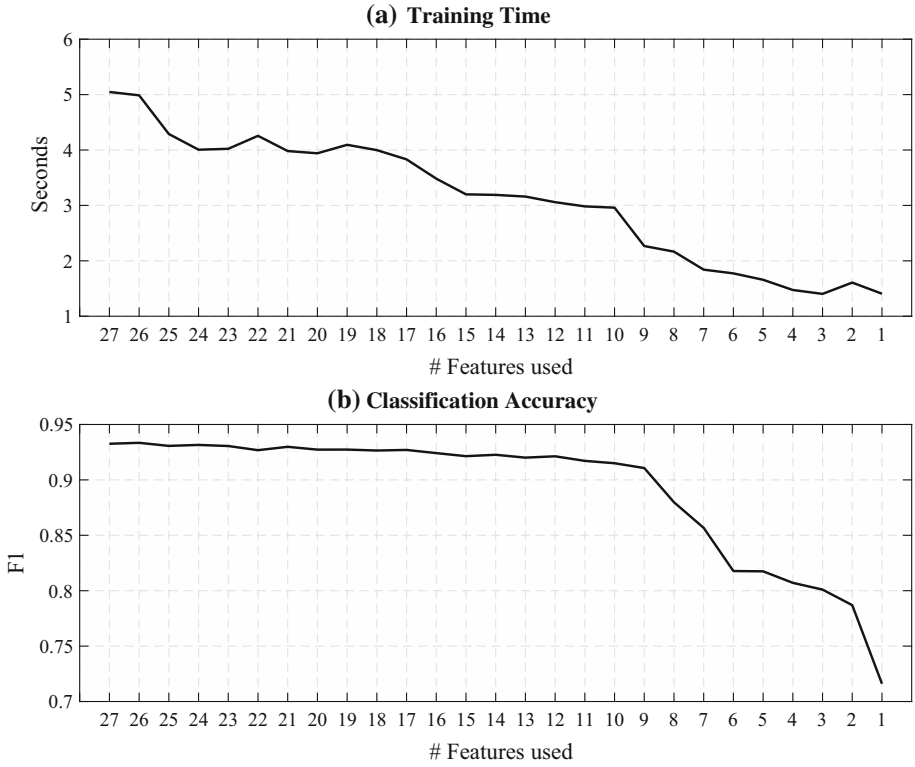

Fig. 11 Reducing feature vector size

Table 7 Classification accuracy of bagged classifier using the reduced feature vector

Measure/class	Facebook	8-Ball	Skype	Viber	WhatsApp	Youtube
FPR	0.042	0.014	0.0009	0.002	0.019	0.019
TPR	0.833	0.9135	0.970	0.996	0.851	0.917
F1	0.819	0.922	0.983	0.993	0.858	0.917

shows that using nearly one third of the feature vector size will improve time performance by 44%. To validate the new feature vector, we trained and evaluated the Bagged classifier once again using the training and testing sets with the Hold Out technique. Table 7 shows TPR, FPR, and F1 score for all the classes.

The aim in the feature reduction phase is to preserve at least 80% detection accuracy and above for all classes while achieving a significant increase in the performance. Thus, feature elimination stopped when the detection accuracy of Facebook dropped below 80% measured by the $F1$ score. Based on that, we decided to use the reduced feature vector expressed by the top 9 features in the outlier detection phase. Eliminating noisy features will help in increasing the detection accuracy of outlier-samples coming from unknown classes.

We can notice from features ranking (Fig. 10) that the packet length is an important feature. Top five features in the ranked set of features are derived from the packet length. Figure 12

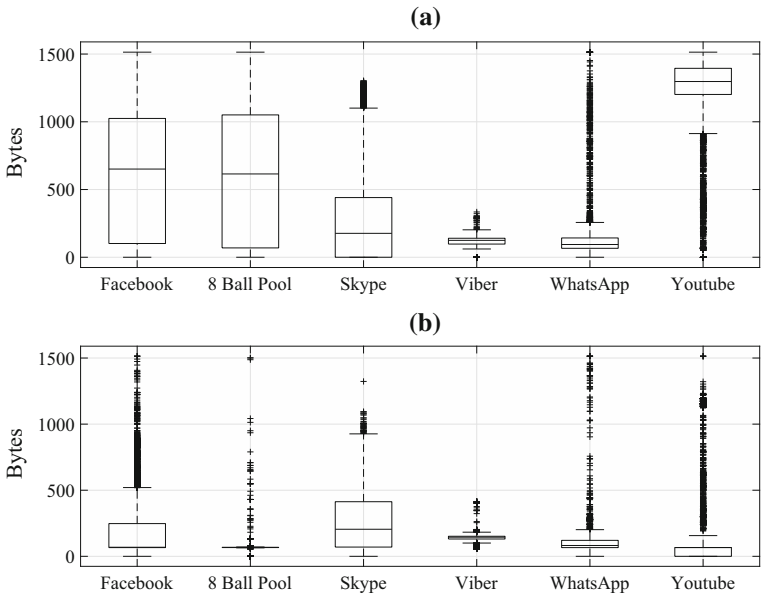

Fig. 12 Packet length stats. **a** Packets-in average length and **b** packets-out average length

shows the boxplot of outgoing and incoming packet lengths. We can see that each App differs in the incoming and outgoing packet lengths from other Apps.

6.4 Outlier detection

The proximity score of Bagged Trees is used to detect possible outliers. However, due to the complexity of such process, the confidence score is used to identify these possible outliers first and decide whether the proximity score should be computed or not. To evaluate the proposed outlier detection mechanism, we installed another famous social media App, Snapchat, on a single testing device. The App was used for 15 min to extract some labeled traffic. Then, the labeled packets are processed to get a total of 359 samples. To calculate the proximity, a Bagged Trees classifier is trained using a balanced dataset of samples extracted from known classes (Apps). Then, the proximity of Snapchat samples is calculated against each known App samples. Additionally, we calculate the proximity of some testing samples which are extracted from known Apps and form 30% of the training set. Next, the proximity and outlier score are compared between samples of known Apps and the newly extracted samples of Snapchat. The proximity and outlier score are calculated for all samples in the testing set regardless of the classification result.

Figure 13 shows the confidence score density plot of the known classes and Snapchat samples. We can notice that Snapchat samples scores average around 0.4. The majority of samples had their highest confidence score between 0.2 and 0.7. On the other hand, classified instances from known classes had a higher confidence score that averages around 0.95. Thus, using the confidence score can limit proximity calculations to instances having their score distributed outside the range defined by known classes' confidence score. This threshold is a tunable parameter which is controlled by the administrator. The threshold represents a tradeoff between complexity and detection accuracy. If the administrator selects a high

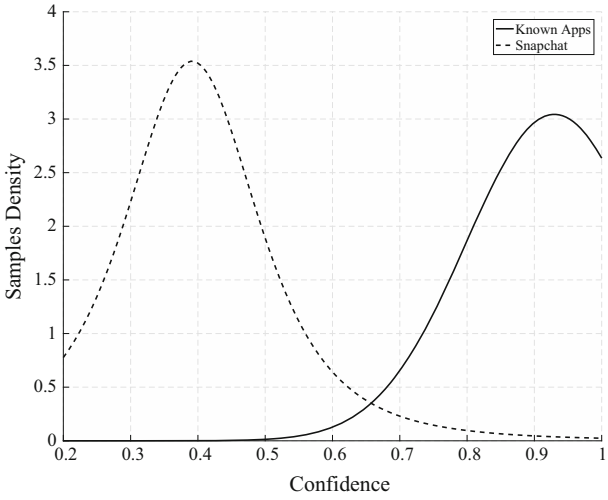

Fig. 13 Confidence score density distribution

value for the threshold, more samples will be candidates for the outlier score calculation since they fall outside the known classes' range. The framework will have a high detection accuracy since it calculates the outlier score for a wider range of samples. On the other hand, if the confidence score threshold is set to a low value, a narrower range of samples will be a candidate for outlier score calculation and the FN rate will increase. In this implementation, we did not set a value for this threshold since we needed to calculate the outlier score for all the samples and report the outlier detection accuracy. However, setting the threshold is presented as part of the analysis.

Figure 14 shows the difference in the normalized outlier score distribution for the classified samples. The outlier score is calculated for each class testing samples versus Snapchat samples. From the boxplot, we can notice that the outlier score of known samples averages around a lower value compared to Snapchat samples. To differentiate between known and unknown samples, we have to set a decision-making threshold based on the outlier score distribution. To detect the best threshold, we plot the FPR vs. TPR of each threshold value from 1 to zero with decrements of 0.05. Figure 15 shows the ROC curve resulting from all iterations on the threshold value. By examining the TPR and FPR of each decision threshold, the value is set to achieve best detection accuracy and lowest false alarm rate. The decision threshold is set differently for each App. The best reported detection accuracy is 94% with a false alarm rate of 5% at a normalized decision threshold of 0.017. This value is picked at the point where the detection rate started to converge slowly while having increased jumps in the false alarm rate. Eventually, the decision threshold value is a tunable parameter and can be optimized for new types of unknown flows.

The proposed classifier is able to detect Snapchat flows and differentiate them from samples of known classes. This would provide the proposed framework with the ability to make a decision, dynamically, whether to consider new outliers from known classes or to discard them. The framework can be adaptive to normal changes that may occur in Apps behavior, or the it can consider the detected flows as abnormalities caused by anonymous flows on the network. Since the detection mechanism succeeded in detecting new flows and distinguish them from flows of known classes, the framework should be able to detect

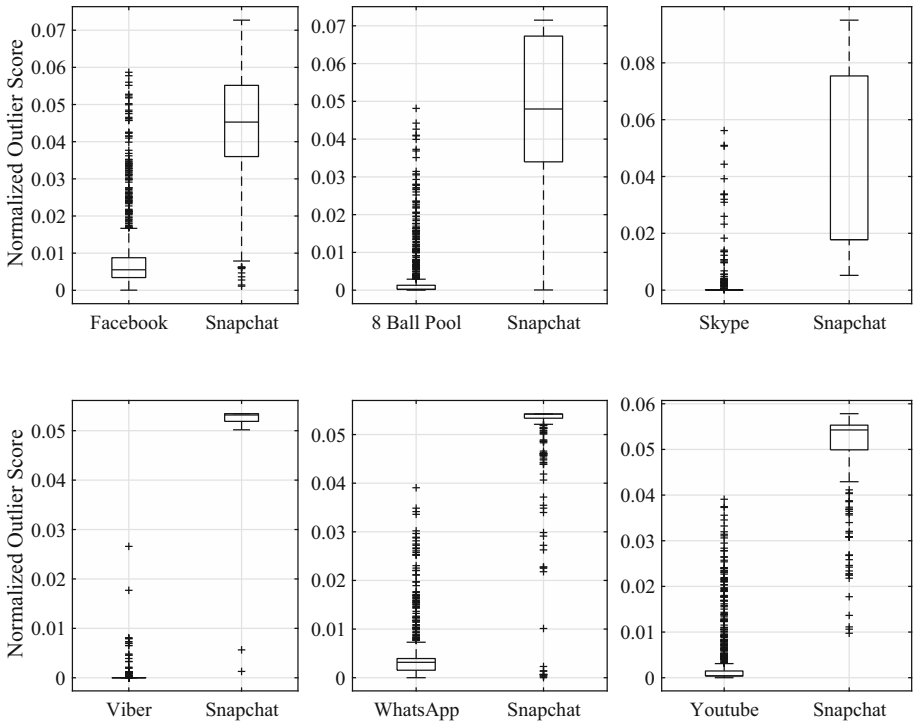

Fig. 14 Normalized outlier score stats

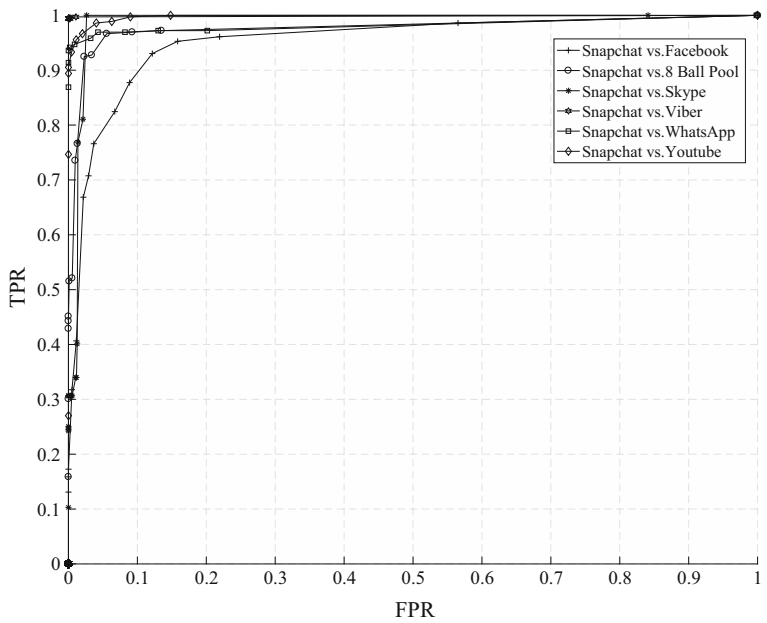

Fig. 15 Outlier detection ROC curve

changes in flows within the same class. The verification of this feature is left for future work.

7 Conclusion

In this work, we successfully designed and implemented a classification component that can be used in App identification. The proposed framework accuracy measured by the F1 is 93.78% with a false alarm rate less than 0.5%. The Bagged Classifier size is reduced by pruning the number of trees while preserving 99% of the classification accuracy. Additionally, we were able to reduce the feature vector by more than 50% of the original size while keeping nearly the same detection accuracy. The framework can be trained by a group of chosen benign participants to detect abnormal behavior of similar Apps across the network. The abnormal behavior is detected using an outlier detection mechanism with 94% accuracy. The outlier detection proved the ability to differentiate between anonymous patterns and known classes. The complexity of such a process is controlled by taking the confidence score into consideration. The proposed framework has many parameters to be set such as the idle time of flows, the sampling interval, the flow length, the decision making threshold, and other parameters such as the various network conditions to perform data collection, and the initial group of benign users who will create the base profile. As for future work, we are planning to devise an algorithm that defines all these parameters, which is considered as a security extension. The algorithm should also define an adaptive framework that can handle outlier detection more efficiently. An adaptive framework would consider normal changes in Apps behavior while preserving the ability of detecting anonymous flows that could be malicious. This approach needs crowd interaction to query suspicious Apps, which is implemented in [30]. Game theory is applied on crowd interactions on certain Apps and identified malicious activities by comparing users feedback. This way, the information provided by the crowd can be used to detect real malicious Apps on the network, or identify benign Apps that have been infected.

Acknowledgements This research is funded by TELUS Corp., Canada.

References

1. Smartphone os market share 2015, 2014, 2013, and 2012. http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed 2016
2. Baghel SK, Keshav K, Manepalli VR (2012). An investigation into traffic analysis for diverse data applications on smartphones. In: IEEE 2012 national conference on communications (NCC), pp 1–5
3. Bauer E, Kohavi R (1999) An empirical comparison of voting classification algorithms: bagging, boosting, and variants. Mach Learn 36(1–2):105–139
4. Berndt DJ, Clifford J (1994) Using dynamic time warping to find patterns in time series. KDD workshop, vol 10. Seattle, WA, pp 359–370
5. Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140
6. Burguera I, Zurutuza U, Nadjm-Tehrani S (2011). Crowdroid: behavior-based malware detection system for android. In Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices, pp 15–26. ACM. Chicago, IL, USA
7. Chen C, Liaw A, Breiman L (2004) Using random forest to learn imbalanced data. University of California, Berkeley, pp 1–12
8. Choi Y, Chung JY, Park B, Hong JW-K (2012) Automated classifier generation for application-level mobile traffic identification. In: 2012 IEEE network operations and management symposium. IEEE. MAUI, HAWAII, USA, pp 1075–1081

9. Conti M, Mancini LV, Spolaor R, Verde NV (2016) Analyzing android encrypted network traffic to identify user actions. IEEE Trans Inf Forensics Secur 11(1):114–125
10. Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297
11. Dai S, Tongaonkar A, Wang X, Nucci A, Song D (2013) Networkprofiler: towards automatic fingerprinting of android apps. In: INFOCOM, 2013 Proceedings IEEE. IEEE. Turin, Italy, pp 809–817
12. Efron B, Tibshirani RJ (1994) An introduction to the bootstrap. CRC Press, Boca Raton
13. Falaki H, Lymberopoulos D, Mahajan R, Kandula S, Estrin D (2010). A first look at traffic on smartphones. In: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pp 281–287. ACM. Melbourne, Australia
14. Johnson R, Wang Z, Gagnon C, Stavrou A (2012) Analysis of android applications' permissions. In: 2012 IEEE sixth international conference on software security and reliability companion (SERE-C). IEEE. Gaithersburg, MD, USA, pp 45–46
15. Kotsiantis SB (2007) Supervised machine learning: a review of classification techniques In: Proceedings of the 2007 conference on emerging artificial intelligence applications in computer engineering: real word AI systems with applications in eHealth, HCI, information retrieval and pervasive technologie. IOS Press, Netherlands, pp 3–24. http://dl.acm.org/citation.cfm?id=1566770.1566773
16. Kuncheva LI (2004). Classifier ensembles for changing environments. In: International workshop on multiple classifier systems, Springer, pp 1–15
17. Li J, Zhai L, Zhang X, Quan D (2014) Research of android malware detection based on network traffic monitoring. In: 2014 9th IEEE conference on industrial electronics and applications. IEEE. Hangzhou, China, pp 1739–1744
18. Miller KW, Voas JM, Hurlburt GF (2012) Byod: security and privacy considerations. It Prof 14(5):53–55
19. Mongkolluksamee S, Visoottiviseth V, Fukuda K (2016) Combining communication patterns and traffic patterns to enhance mobile traffic identification performance. J Inf Process 24(2):247–254
20. Moore AW (2001) Information gain. School of Computer Science, Carnegie Mellon University. http://www.cs.cmu.edu/~awm/tutorials
21. Murphey YL, Guo H, Feldkamp LA (2004) Neural learning from unbalanced data. Appl Intell 21(2):117–128
22. Nissim N, Moskovitch R, BarAd O, Rokach L, Elovici Y (2016) Aldroid: efficient update of android anti-virus software using designated active learning methods. Knowl Inf Syst 49(3):795–833
23. Oprişa C, Gavriluţ D, Cabău G (2016) A scalable approach for detecting plagiarized mobile applications. Knowl Inf Syst 49(1):143–169
24. Osuna E, Freund R, Girosi F (1997) Support vector machines: training and applications. Massachusetts Institute of Technology, USA. http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Amitai%3AMIT-AILab%2F%2FAIM-1602
25. Pieterse H, Olivier MS (2012) Android botnets on the rise: trends and characteristics. In: IEEE 2012 Information security for South Africa, pp 1–5
26. Qi Y, Cao M, Zhang C, Wu R (2014) A design of network behavior-based malware detection system for android. IN: International conference on algorithms and architectures for parallel processing. Springer. Dalian, China, pp 590–600
27. Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106
28. Quinlan JR (1996) Bagging, boosting, and c4. 5. AAAI/IAAI 1:725–730
29. Rish I (2001) An empirical study of the naive bayes classifier. In: IJCAI 2001 workshop on empirical methods in artificial intelligence, vol 3. IBM New York. Seattle, Washington, USA, pp 41–46
30. Saab F, Elhajj I, Kayssi A, Chehab A (2016). A crowdsourcing game-theoretic intrusion detection and rating system. In Proceedings of the 31st annual ACM symposium on applied computing, pp 622–625. ACM
31. Sanz B, Santos I, Laorden C, Ugarte-Pedrero X, Nieves J, Bringas PG, Álvarez Marañón G (2013) Mama: manifest analysis for malware detection in android. Cybern Syst 44(6–7):469–488
32. Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y (2012) andromaly: a behavioral malware detection framework for android devices. J Intell Inf Syst 38(1):161–190
33. Shabtai A, Tenenboim-Chekina L, Mimran D, Rokach L, Shapira B, Elovici Y (2014) Mobile malware detection through analysis of deviations in application network behavior. Comput Secur 43:1–18
34. Taylor VF, Spolaor R, Conti M, Martinovic I (2016) Appscanner: automatic fingerprinting of smartphone apps from encrypted network traffic. In: 2016 IEEE European symposium on security and privacy (EuroS&P). IEEE. Saarbrcken, GERMANY, pp 439–454
35. Tsompanidis I, Zahran AH, Sreenan CJ (2014) Mobile network traffic: a user behaviour model. In: 2014 7th IFIP wireless and mobile networking conference (WMNC). IEEE. Vilamoura, Algarve, Portugal, pp 1–8

36. Upadhyaya S, Singh K (2012) Classification based outlier detection techniques. Int J Comput Trends Technol 3(2):294–298
37. Wei X, Gomez L, Neamtiu I, Faloutsos M (2012). Profiledroid: multi-layer profiling of android applications. In: Proceedings of the 18th annual international conference on Mobile computing and networking, pp 137–148. ACM. Istanbul, Turkey
38. Zaman M, Siddiqui T, Amin MR, Hossain MS (2015) Malware detection in android by network traffic analysis. In: 2015 International conference on networking systems and security (NSysS). IEEE. Dhaka, Bangladesh, pp 1–5
39. Zhang J, Zulkernine M, Haque A (2008) Random-forests-based network intrusion detection systems. IEEE Trans Syst Man Cybern C Appl Rev 38(5):649–659

Georgi Ajaeiya received his Bachelor of Engineering in Information Technology from Aleppo University, Aleppo, Syria, in 2014 and an M.E. degree in Electrical and Computer Engineering from the American University of Beirut (AUB), Beirut, Lebanon, in 2017. From 2015 to 2017, he did research work in the Networking and Security research group at AUB which is advised by Professors Ali Chehab, Ayman Kayssi, and Imad Elhajj. He is currently a part-time researcher working with the same group. His research interests include Network and Information Security, Data mining and Pattern Recognition, and Android development. He is a member of IEEE.

Imad H. Elhajj received his Bachelor of Engineering in Computer and Communications Engineering, with distinction, from the American University of Beirut in 1997 and the M.S. and Ph.D. degrees in Electrical Engineering from Michigan State University in 1999 and 2002, respectively. He is currently an Associate Professor with the Department of Electrical and Computer Engineering at the American University of Beirut. In 2014, he co-founded, with two other professors in ECE, SAUGO 360 the first startup to be incubated at AUB. Dr. Elhajj is the past chair of IEEE Lebanon Section, senior member of IEEE and senior member of ACM. He is a member of the World Economic Forum Global Agenda Council on Artificial Intelligence and Robotics. He is an ABET program evaluator. His research interests include instrumentation and robotics, cyber security, sensor and computer networks, and multimedia networking. Imad received the Best Research Paper Award at the Third International Conference on Cognitive and Behavioral Psychology (CBP), the Best Paper award at the IEEE Electro Information Technology Conference in June 2003, and the Best Paper Award at the International Conference on Information Society in the twenty-first Century in November 2000. Dr. Elhajj is recipient of the Teaching Excellence Award at the American University of Beirut, June 2011, the Kamal Salibi Academic Freedom Award, 2014, and the most Outstanding Graduate Student Award from the Department of Electrical and Computer Engineering at Michigan State University in April 2001.

Ali Chehab received his Bachelor degree in EE from AUB in 1987, the Masters degree in EE from Syracuse University in 1989, and the Ph.D. degree in ECE from the University of North Carolina at Charlotte, in 2002. From 1989 to 1998, he was a lecturer in the ECE Department at AUB. He rejoined the ECE Department at AUB as an Assistant Professor in 2002 and became Full Professor in 2014. He received the AUB Teaching Excellence Award in 2007. He teaches courses in Programming, Electronics, Digital Systems Design, Computer Organization, Cryptography, and Digital Systems Testing. His research interests include: Wireless Communications Security, Cloud Computing Security, Trust in Distributed Computing, Low Energy VLSI Design, and VLSI Testing. He has more than 200 publications. He is a senior member of IEEE and a senior member of ACM.

Ayman Kayssi studied electrical engineering and received the BE degree, with distinction, in 1987 from the American University of Beirut (AUB), and the MSE and Ph.D. degrees from the University of Michigan, Ann Arbor, in 1989 and 1993, respectively. In 1993, he joined the Department of Electrical and Computer Engineering (ECE) at AUB, where he is currently a full professor. From 2004 to 2007, he served as chairman of the ECE Department at AUB and is currently associate dean of the Maroun Semaan Faculty of Engineering and Architecture. He teaches courses in electronics and in networking and has received AUB's Teaching Excellence Award in 2003. His research interests are in information security and networking, and in integrated circuit design and test. He has published more than 200 articles in the areas of security, networking, and VLSI. He is a senior member of IEEE, and a member of ACM, ISOC, and the Beirut OEA

Marc Kneppers MSc. Astronomy, University of Western Ontario (Western University) Chief Security Architect and TELUS Fellow TELUS Communications, 1997-present Member: ACM, NGMN SCT, ATIS, CTCP, CSTAC