Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Mobile malware detection through analysis of deviations in application network behavior

A. Shabtai*, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, Y. Elovici

Department of Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva, Israel

ARTICLE INFO

Article history:
Received 1 November 2013
Received in revised form
17 February 2014
Accepted 21 February 2014

Keywords:
Mobile applications
Network traffic
Machine learning
Android malware
Anomaly detection
Smartphone security

ABSTRACT

In this paper we present a new behavior-based anomaly detection system for detecting meaningful deviations in a mobile application's network behavior. The main goal of the proposed system is to protect mobile device users and cellular infrastructure companies from malicious applications by: (1) identification of malicious attacks or masquerading applications installed on a mobile device, and (2) identification of republished popular applications injected with a malicious code (i.e., repackaging). More specifically, we attempt to detect a new type of mobile malware with self-updating capabilities that were recently found on the official Google Android marketplace. Malware of this type cannot be detected using the standard signatures approach or by applying regular static or dynamic analysis methods. The detection is performed based on the application's network traffic patterns only. For each application, a model representing its specific traffic pattern is learned locally (i.e., on the device). Semi-supervised machine-learning methods are used for learning the normal behavioral patterns and for detecting deviations from the application's expected behavior. These methods were implemented and evaluated on Android devices. The evaluation experiments demonstrate that: (1) various applications have specific network traffic patterns and certain application categories can be distinguished by their network patterns; (2) different levels of deviation from normal behavior can be detected accurately; (3) in the case of self-updating malware, original (benign) and infected versions of an application have different and distinguishable network traffic patterns that in most cases, can be detected within a few minutes after the malware is executed while presenting very low false alarms rate; and (4) local learning is feasible and has a low performance overhead on mobile devices.

Introduction

Along with the significant growth in the popularity of smartphones and the number of available mobile applications, the amount of malware that harm users or compromise their privacy has also dramatically increased (Mylonas et al., 2013). Furthermore, the significant growth in social networks and "always-connected" applications has caused a dramatically increasing influence on the traffic and signaling loads on the mobile network infrastructure, potentially leading to network congestion incidents.

Network overloads can be caused by either intentional attacks or unintentionally by poorly designed benign "network

* Corresponding author. Tel.: +972 8 64728121.
E-mail address: shabtaia@bgu.ac.il (A. Shabtai).
http://dx.doi.org/10.1016/j.cose.2014.02.009
0167-4048/© 2014 Elsevier Ltd. All rights reserved.

unfriendly" applications. Both malware activities and "network unfriendly" applications regularly affect network behavior patterns and can be detected by monitoring an application's network behavior. Thus, the monitoring and the analysis of the traffic patterns of network-active applications are essential for developing effective solutions for the prevention of network overloads.

Recently, a new type of mobile malware hosted on the official Google Android marketplace, the Google Play Store (formerly known as the Android Market), was detected (Symantec blog). The main feature which distinguishes this Trojan horse (named Android.Dropdialer) from earlier known malware is its self-updating capabilities. Applications infected by this Trojan and hosted on the Play Store were initially absolutely benign and did not contain any malicious functionality. The package containing malicious payload was downloaded from the Internet sometime after the original application was installed on the device. This allowed the malware to remain undiscovered on the market for several weeks and generate tens of thousands of downloads. Such an attack is easy to launch because as shown by (Mylonas et al., 2012), smartphone users believe that applications in the application repository (i.e., Play Store) are risk-free which makes them more vulnerable to social engineering.

In general, any malware can be downloaded and executed on a device using such a "remote payload" technique. Specifically, in the case of Dropdialer Trojan, the downloaded malicious package sent SMS messages to premium-rate numbers. The download action can be scheduled for both specific and random time in the future, or may even be initiated remotely by sending a command message to the devices, using, for instance, Google's push notification service (GCM or former C2DM). Several different techniques allowing Android application update from a remote location exist. These techniques are elaborated in Self-updating malware section.

Such self-updating malware cannot be detected by standard static techniques as the original version of the application is absolutely benign and does not contain any malicious code. Transforming a benign application into a malicious one using dynamic code loading makes the static analysis of the host application virtually irrelevant. Detection by dynamic analysis techniques can be simply avoided by using a time delayed or filtered deployment of the malicious payload. For example, on days 1–5, an attacker may offer benign update, and on day 6 switch to the malicious one or offer the update to a certain geo-location, filter by carrier, device type, OS version, etc. Furthermore, a sophisticated attacker can use server-side polymorphism employing obfuscation and encryption to completely deter and complicate the dynamic/static analysis efforts. It is also difficult to identify this type of emerging malware since the self-updating technique is often used by legitimate applications for benign purposes as well (e.g., to upgrade already installed games with new levels, bug fixes, update the detection engine of anti-virus applications, etc.)

A recent survey shows that 70% of known mobile malware steals user information or credentials (Felt et al., 2011). Therefore, in this paper we aim to detect self-updating malware which steals user data or allows spying on users. For that, we present a behavior anomaly detection system for monitoring and detecting meaningful deviations in a mobile application's network behavior. The proposed system serves two main purposes. First, it allows protection of mobile device users from malware that steals user data or spies on users. Second, it allows aggregation and analysis of an application's network traffic patterns to be used for protecting cellular infrastructure from malicious and "network unfriendly" applications.

Our method is based on monitoring applications running on a device, learning the models which represent their "normal" network behavior, and detecting any deviations from the learned patterns. Such continuous monitoring of an application's network behavior could offer suggestions about certain unexplained changes in the application's network behavior any time they occur.

The new system supports two main use cases. The first case relates to applications already installed on a device and the second, to newly downloaded and installed applications. In the first case, the network traffic pattern of an application can be changed due to: (1) a change in the user's behavior; (2) an application update to a new benign version; or (3) a malicious attack. In this case the system's purpose is to detect the deviation in the application's traffic pattern and correctly classify it to one of the three above mentioned reasons. In the second case, the system's purpose is to identify whether the new application is actually a modification of another application with some new (possibly malicious) behavior. This is done by comparing the network behavior patterns with the behavior of the same application as observed on other devices that are already running the application or are characterized with the behavior of a similar application.

To meet the above requirements, the system follows the hybrid Intrusion Detection Systems (IDS) approach and is designed in the client-server architecture. The responsibility of the *client-side* software is to monitor the applications that are *already* installed and running on a device, learn their user-specific *local* models and detect any deviations from the observed "normal" behavior. The *local* models that are derived from the client-side software are designed to detect changes in the application's traffic patterns as a result of a change in the user's behavior, an update to a newer (benign) version or as a result of a malicious attack. The responsibility of the *server-side* software is to compare the behavior of a newly installed application against the application's known traffic patterns in order to identify whether the new application is a modification of a known application with some new (possibly malicious) behavior. This is done by aggregating data reported by various mobile devices and deriving *collaborative* models, which represent the common traffic patterns of numerous users for each application. The *local* models are used for detection of deviations in traffic patterns of installed applications; the *collaborative* models are used to verify of newly installed applications vs. the known traffic patterns.

This paper focuses on local learning and detection. We present the client-side of the whole system, i.e., the sub-system developed for the extraction of per-application traffic features, learning of *local* models and detection of deviations from the normal user's behavior. This sub-system (from now on referred to as the *system*) is implemented as a regular Android application running on the OS user space, and evaluated on regular (un-rooted) Android devices. The collaborative detection and server-side modules are beyond the scope of this paper and are to be discussed in future work.

We evaluated the system using a wide range of different applications, their versions, and several self-developed as well as real malware applications. The results demonstrate that applications have very specific network traffic patterns and that certain application categories can be distinguished by their traffic patterns. In addition, deviations from an application's normal behavior can be detected quickly and accurately. The conducted experiment also shows the feasibility of the proposed implementation and analysis performance overhead on mobile devices. The server-side of the system is currently under development and thus the collaborative models learning and detection is part of our future research.

The rest of the paper is organized as follows: Self-updating malware section presents the self-updating malware. Related work section discusses related work. System overview section describes the client-side system components and the proposed detection methods. Evaluation section presents the results of the evaluation experiment. In Resources overhead section we present the resource overhead analysis of our system, and in Discussion section we discuss the achieved results. Lastly, Summary and conclusions section concludes the paper and outlines future research.

Self-updating malware

Four main techniques can be used to create self-updating applications for Android that can download new pieces of software stored remotely. These techniques are especially attractive for malware developers as they permit the addition of malicious behavior to initially benign applications, thus making malware identification much more difficult. The four techniques are:

(1) **Offer the user an update (i.e., complete replacement) to the original application**

This update method is the simplest of the four. The initial version of the application, which can be distributed via different application markets, downloads from a remote server another version of the same application that is signed by the same developer key and may potentially contain malware of any type. It then presents the user with an "update available" notification. Once the user agrees to install the update, the original application is replaced with the new malicious version which can be as different from the original one as it wishes.

(2) **Dynamic loading of a compiled Android code (i.e., executable dex files) using Android's DexClassLoader class**

The second type of behavior differs from the first one in that it downloads a package with a compiled Android code (i.e., dex file) using the Android's DexClassLoader class, and continues on to dynamically load and execute the downloaded code without any notification to the user. In this type of attack, abilities of the downloaded code are constrained by the permissions of the original application.

(3) **Dynamic loading of a binary shared object file (also called .so library) or an executable file containing native code which can be executed using Java's Runtime class**

The third method is essentially similar to the second one, only this time the payload contains a native library or an executable file of a program coded in C or C++ (binary shared object file also called .so library) which can be executed using Java's Runtime class.

The three behavior types list above can be triggered by an external event, such as a phone call or an incoming text message.

(4) **Dynamic loading of a certain file (e.g., mp3, jpg, flash, and pdf) containing a malicious payload (i.e., shellcode)**

The last behavior type exploits system libraries or third party application vulnerabilities such that a user clicking on a certain file (e.g., mp3, jpg, avi, wmv, pdf) actually causes the execution of the malicious payload code. This behavior type is currently only theoretical and no attacks of such type have been detected yet. However, some of the recently fixed Flash Player vulnerabilities (Adobe blogs), for instance, could be exploited for this purpose.

Such a self-updating capability makes it possible for malware developers to simultaneously penetrate new threats into numerous devices. The new threats can even exploit system vulnerabilities which were unknown at the time of the development of the initial application version. Thus, developing new methods which allow for identification and defense against this new emerging malware type is of great importance.

Related work

In order to overcome the known limitations of traditional signature-based and dynamic/static analysis solutions, and due to the dramatic increase in the number of malware applications targeting smartphones, various behavioral-based Intrusion Detection Systems (IDS) have recently been proposed in the mobile environment. In this section we review these techniques and specify the uniqueness of our approach in light of previously proposed methods.

Traditionally intrusion detection systems are classified according to the protected system type as either host-based (HIDS) or network-based (NIDS) (Amer and Hamilton, 2010). A network-based IDS analyzes network-related data and events in order to detect intrusion attempts. The data can appear in the form of traffic volume, IP addresses, service ports and protocol usage. A host-based IDS resides on and monitors a single host machine. Its performance is based mainly on an analysis of events related to OS information, such as file system, process identifiers, and system calls (Garcia-Teodoro et al., 2009).

Some very informative reviews of several methods for intrusion detection on mobile devices are presented in

Burguera et al. (2011) and Shabtai et al. (2012). Most of the IDSs for mobile devices have focused on host-based intrusion detection systems, applying either anomaly or rule-based methods on the set of features which indicate the state of the device (Shabtai et al., 2010). However, in most cases, the data interpretation processes are performed on remote servers motivated by limited computational resources of the mobile phone. Only a few of the proposed systems perform the learning or data analysis directly on the device (Enck et al., 2010; Li et al., 2010; Shamili et al., 2010) and some have applied statistical or machine-learning techniques (Li et al., 2010; Shamili et al., 2010; Dini et al., 2012; Damopoulos et al., 2012) that are very popular and have been successfully used in traditional anomaly detection systems (Garcia-Teodoro et al., 2009; Shamili et al., 2010). Most of the systems either send the observed data to the server for centralized analysis that utilized the multiple data sources (Burguera et al., 2011; Cheng et al., 2007; Moreau et al., 1997; Portokalidis et al., 2010; Schmidt et al., 2009; Qian et al., 2011; Portokalidi et al., 2010) and the unlimited computation resources of the cloud (Khune & Thangakumar, 2012), or perform the learning process offline on the server and plant the learned models back into the devices for the detection process (Shabtai et al., 2010, 2012; Schmidt et al., 2008). Contrary to the earlier proposed methods, the current one performs anomaly detection using only application-level network traffic features while both the learning and the detection processes utilize the machine-learning algorithms that are performed locally on the device.

Consider the earlier proposed systems where learning is performed on the mobile devices (as this is the closest to our work). The system proposed by Shamili et al. in Shamili et al. (2010) utilizes a distributed Support Vector Machine algorithm for malware detection on a network of mobile devices. The phone calls, SMSs, and data communication related features are used for detection. During the training phase support vectors (SV) are learned locally on each device and then sent to the server where SVs from all of the client devices are aggregated. Finally, the server distributes the whole set of SVs to all the clients, and each client updates his own SVs. Thus, although a part of the learning is performed on the device, the server and communication infrastructure, along with additional bandwidth load, are required. Our approach, though planned as a part of the wider client-server system, can perform and be utilized as a stand-alone solution running independently on each mobile device. The authors evaluated their methods using the MIT Reality dataset (Eagle and Pentland, 2006) with manually injected symptoms of malware behavior, and no estimation of resource overhead was carried out.

Li et al. (2010) presented an approach for behavioral-based multi-level profiling IDS considering telephone calls, device usage, and Bluetooth scans. They proposed a host-based system which collects and monitors user's behavioral features on a mobile device. A Radial Basis Network technique was used for learning profiles and detecting intrusions. However, the system capabilities were also tested offline only using the MIT Reality dataset (Eagle and Pentland, 2006) while its feasibility on mobile devices was not tested or verified.

Dai et al. (2010) presented a malware detection system for the Windows Mobile platform. They used API interception techniques for monitoring and analyzing the application's behavior and compared it to the patterns within the predefined library of malicious behavior characteristics. A similar approach for monitoring system events and API calls and mapping them to the behavioral signatures was proposed by Bose et al. (2008) and evaluated using the Symbian OS emulator. However, its detection is limited to the set of predefined malware behavior patterns and thus is incapable of identification of new malware types.

Another recently proposed behavior-based malware detection system for cellular phones is the pBMDS (Xie et al., 2010). The pBMDS is based on correlating user inputs with system calls to detect anomalous activities. A Hidden Markov Model (HMM) is used to learn application and user behaviors from two major aspects: process state transitions and user operational patterns. Built upon these two aspects, the pBMDS identifies behavioral differences between user initiated applications and malware compromised ones. The proposed system needs to be integrated into the smartphone hardware and it works in the kernel level of the mobile OS. Additionally, the resource consumption of the proposed method, which appears to be significant for mobile devices, was not estimated in their paper. Contrary to the pBMDS, our system is a stand-alone application which works from the regular user space and can be simply installed on a device at any time.

Some of the related works monitored the Linux kernel system calls usage as a basis for malware classifications. Burguera et al. (2011) have developed an Android framework named "Crowdroid" that includes a client application that is installed on the device. The application monitors Linux kernel system calls and sends them after a preprocessing to a centralized server. On the server a dataset is built from the list of the system calls, the list of running applications and the device information. *k*-Means algorithm is then used for clustering the applications into two groups – benign and malware applications. Reina et al. (Alessandro et al., 2013) inspected not only the system calls but also the interactions between the Android components (Binder) and the underlying Linux system to characterize low-level OS-specific and high-level Android-specific behaviors. They ran the code on a simulator and used a special tool to simulate the user actions.

Zhao et al. (2012) recorded for each process the access requests to system resources. Each request is represented by a call to Android's service manager. Each process is represented by a sequence of time stamped calls. An SVM algorithm combined with active learning is then applied in order to classify the applications as benign or malicious. Zhang et al. (2013) combined dynamically tracing of the permission requests for resources usage by applications, with tracking sensitive operations on the granted resources (using taint tracking). This combination enabled them to understand how applications utilize the permissions to access sensitive system resources.

Rastogi et al. (2013) widened the scope. Their "AppsPlayground" framework uses three types of dynamic analyses on different Android platform levels: Linux system call tracing to detect known patterns of malware behavior; API calls tracing; taint tracing. The "AppsPlayground" developers used special sandbox emulation with automatic exploration strategies.

The current study is unique in the IDS field in the sense that it proposes a host-based system whose performance is based solely on application-level network events. The proposed system is capable of identifying new malware types with self-updating capabilities. The anomalous behavior of an application is detected in real time on the device and is based on the observed network traffic patterns. This approach is justified by the fact that many malware applications use network communication for their needs, such as sending a malicious payload or a command to a compromised device or getting user data from the device (Felt et al., 2011). Such types of behavior influence the regular network traffic patterns of the application and can be identified by learning the application's "normal" patterns and further monitoring network events. Specifically, recent mobile malware analysis done by anti-virus vendors such as Sophos 2014 Security Threat Report (Sophos, 2014) indicate the following groups of threats: surveillance (i.e., collecting private information such as pictures, location, files, call logs, recording video and sound), data theft (e.g., contacts, account details), botnet activity (e.g., for denial of service), impersonation (sending infected emails or MMS, posting in social media), financial (sending SMS or calling to premium numbers, stealing bank accounts or credit card information), intrusive advertising (collecting massive amount of data on the user for targeted advertising) and self-updating Trojans (i.e., malware that downloads from remote the malicious components and installs it on the device). When analyzing these groups of threats we can conclude that the method and network features will be able to mitigate the surveillance, intrusive advertising and self-updating malware threats which have a high network footprint. It will partially mitigate the data theft and botnet activity type of threats but will not be able to mitigate the impersonation and financial threats which are characterized by very small network footprint.

Additionally, our work is one of the first practical implementations of the machine-learning induction algorithm for mobile OS in general and for the Android platform specifically. The proposed system is capable of continuous monitoring and real time detection of both known and new types of malware applications. Moreover, we have analyzed the resource consumption of the proposed method and demonstrated its feasibility and low resource overhead on mobile devices.

System overview

The proposed system's architecture is presented in Fig. 1 and consists of the following main components:

Graphical User Interface (GUI) – responsible for the communication with the user; presents relevant information and alerts, receives the desirable parameter's configuration, starts and stops the monitoring, etc.

Alerts Handler – responsible for generating alerts and processing user's response to alerts.

Features Extraction – performs the measurements of the defined features at the specified time periods.

Features Aggregation – computes the defined aggregations over all the extracted measurements for the specified time period.

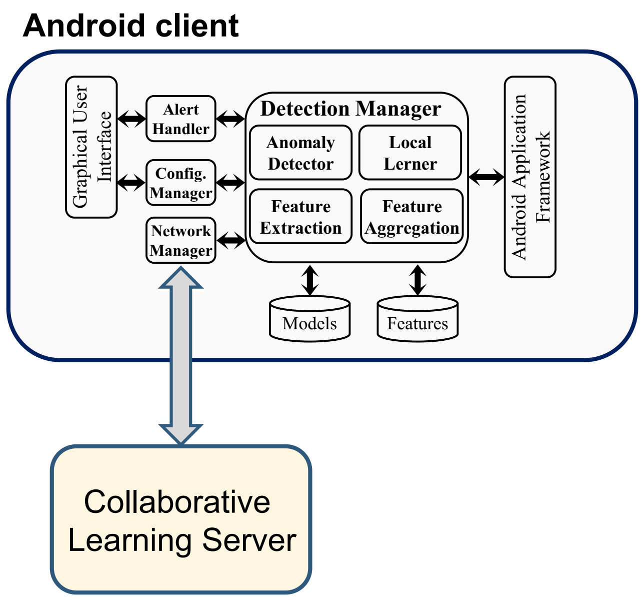


Fig. 1 – The general system architecture consists of a client component installed on the mobile (Android) device and a server component. The responsibility of the client-side software is to monitor the applications that are already installed and running on the device, learn their user-specific local models and detect any deviations from the observed "normal" behavior. The Collaborative Learning Server is responsible for collecting and aggregating data reported by various mobile devices and deriving collaborative models, which represent the common traffic patterns of numerous users for each application.

Local Learner – induces the local models representing an application's traffic patterns specific to the user.

Anomaly Detector – is responsible for the online analysis of an application's network behavior and detection of deviation from its normal patterns.

Below we describe the four main logical components: Feature Extractor, Feature Aggregator, Local Learner, and the Anomaly Detector.

Feature extractor

The Feature Extractor is responsible for extraction (i.e., measuring and capturing) of the defined list of features for each running application at each defined time period. For this purpose it uses the APIs provided by the Android Software Development Kit (SDK). Below is a list of features currently supported by our Feature Extractor:

- sent\received data in bytes and percent out of total amount of the transmitted data;
- network state (Cellular, WiFi or "no network");
- time (in seconds) since application's last send\receive data;
- send\receive mode (eventual\continuous) – derived from "since-last-send\receive-seconds"; i.e., if the last send or

receive data event was detected less than a specified number of second ago, the corresponding (send or receive) mode is continuous, otherwise it is eventual;
- two application states – the first specifies whether the application is in foreground or background and the second specifies whether the application is among the active or non-active tasks at the time of the measurement;
- time in fore\background (in seconds and percent) – total time that an application has been in fore\background since the last monitoring of this application was started;
- minutes since application's last active\modified time;

The extraction time period is a configurable parameter. In our experiments this parameter was set to 5 s.

Feature aggregator

The purpose of the Feature Aggregator is to provide a concise representation of the extracted application's traffic data. For this purpose, a list of various aggregation functions was defined. The instances of the aggregated data are used to induce machine-learning models representing an application's behavior and for further anomaly detection. We defined and evaluated an extended list of possible aggregated features in order to determine a notion of the usefulness of the features. According to the evaluation results, a list of the most useful features was determined (see the Evaluation section). Below is a list of all the currently supported aggregated features:

- Average, standard deviation, minimum, and maximum of sent\received data in bytes;
- Average, standard deviation, minimum, and maximum of sent\received data in percent out of total amount of the transmitted data;
- Percent of sent\received bytes;
- Time intervals between sent\received events – the sent\received events that occurred within the time interval of less than 30 s from the previous corresponding event contribute to the calculation of the *inner average send\receive time interval*. The events that occurred within the time frame above or equal to 30 s from the previous corresponding event contribute to the calculation of the *outer average sent\received time interval*. Additionally, two types of intervals: *local* – for each specific aggregation time period, and *global* – averaged over the whole monitoring process, were calculated. The *local* time intervals describe an application's behavior at certain monitoring time points, while the *global* time intervals describe the application's general behavior observed up until the current point in time;
- Network state – Cellular, WiFi, none or mixed. The mixed state was determined in the case where several different states (i.e., Cellular and WiFi) were observed during the same aggregation period;
- Minutes past since application's last sent\received data event;
- Application state 1 – foreground, background or mixed. Mixed state was determined in the case where several different states were observed during the same aggregation period;
- Application state 2 – active, non-active or mixed;
- Total and local time (in seconds) for which the application was in the fore\background state. Local time may vary from 0 to 60 s and it may represents the value specific for the current aggregation interval, while the total time is aggregated over the whole application's active time period;
- Minutes past since the application's last active time;
- Days past since application's last modified time determined according to the application's installer file (i.e., ".apk" for Android) modification time.

Similar to the extraction, the aggregation time period is a configurable parameter. For the current experiment it was set to 1 min.

Local models learning

Our main goal in this paper is to learn user-specific network traffic patterns for each application and determine if meaningful changes occur in the application's network behavior on the monitored device (i.e., user-specific). This task relates to the family of semi-supervised anomaly detection problems, which assumes that the training data includes samples for "normal" data only. These types of problems can be solved, for example with one-class support vector machines (SVMs), the local outlier factor (LOF) method or with clustering based techniques (Chandola et al., 2009; Noto et al., 2010).

In this paper we decided to convert the semi-supervised learning problem to a set of supervised problems for which numerous well established and quick algorithms exist. For this purpose we follow the *cross-feature analysis* approach presented in Huang et al. (2003) and then further analyzed by Noto et al. (2010). Both of these works have found this approach successful and useful for anomaly detection. However, Huang et al. (2003) consider only features with discrete values, and Noto et al. (2010) mainly focus on methods for combining the results of multiple feature predictors to make the final decision about instance normality. In our problem most of the features are numerical, and an efficient implementation is desired in order to run on a limited resource mobile phones. Thus, in this work we follow the general idea of the above approach; however, our implementation differs in several details from the aforementioned methods. Both the general idea and our current implementation are presented below.

The main assumption underlying the cross-feature analysis approach is that in normal behavior patterns, strong correlations between features exist and can be used to detect deviations caused by abnormal activities. The basic idea of a cross-feature analysis method is to explore the correlation between one feature and all the other features. A pseudo-code for the local models learning algorithm is given in Fig. 2. Formally, it tries to solve the classification problems C_i: $\{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_L\} \rightarrow \{f_i\}$, where $\{f_1, f_2, \ldots, f_L\}$ is the features vector and L is the total number of features. Such a classifier is learned for each feature i, where $i = 1, \ldots, L$ (lines 1–4 in Fig. 2). Thus, an ensemble of learners for each one of the features represents the model through which each features' vector will

```
Local Learning
input: data - application's training set with $L$ features
output: $C$ - an array of $L$ classification models (one for each feature)

1. FOR EACH feature $f_i$ WHERE $i=0,...,L$
2.     data.setClassAttribute($f_i$)
3.     $C_i$ = learnModel(data)
4. END FOR
5. RETURN $C$

Anomaly Detection
input: $x$ – vector of $L$ attributes to verify for normality; $C$; threshold
output: 0 - normal event; 1 - anomaly
1.  finalProb = 1; maxProb = 0.999
2.  FOR EACH feature $f_i$ WHERE $i=0,...,L$
3.      $x$.setClassAttribute($f_i$)
4.      predictedVal = $C_i$.classify($x$)
5.      realVal = $x$.getValue($f_i$)
6.      IF ($f_i$ is NOMINAL)
7.          IF (predictedVal== realVal) distance=maxProb
8.          ELSE distance = 0
9.      ELSE
10.         diff=|realVal-predictedVal|/$C_i$.getMean
11.         distance =MIN(maxProb, diff)
12.     prob=1-distance
13. END FOR
14. finalProb= finalProb * prob
15. IF (finalProb> threshold) RETURN 1
```

Fig. 2 – A pseudo-code for the "local models learning" and "anomaly detection" algorithms. During the learning phase a classifier is trained for each feature f_i, where $i = 1, \ldots, L$. When a feature's vector representing a normal event is tested against the models generated during the learning phase, there is a higher probability for the predicted value to match (for discrete features) or be very similar (for numeric features) to the observed value. The more the predictions differ from the true values of the corresponding features, the more likely it is that the observed vector comes from a different distribution than the training set (i.e., represents an anomaly event).

be tested for "normality". The C4.5 Decision Tree algorithm was used for learning the classification model when the target class consists of categorical values. For learning the classification model for the numeric target attribute, several methods that are capable with numeric classes were evaluated. The evaluated methods and their results are presented in the Evaluation section.

Anomaly detection

This module is responsible for the online analysis of an application's network behavior and the detection of deviations from normal patterns. The procedure utilized for testing each individual instance is further described.

When a feature's vector representing a normal event is tested against C_i, there is a higher probability for the predicted value to match (for discrete features) or be very similar (for numeric features) to the observed value. However, in the case of a vector representing abnormal behavior, the probability of such a match or similarity is much lower. Thus, by applying all the features models to a tested vector and combining their result, a decision about vector normality can be derived. The more the predictions differ from the true values of the corresponding features, the more likely it is that the observed vector comes from a different distribution than the training set (i.e., represents an anomaly event).

A pseudo-code for the anomaly detection algorithm is given in Fig. 2. For each predictor C_i we compute the probability of the corresponding feature value of a vector x to come from an abnormal event (lines 2–4). This probability, noted $P(f_i(x)$ *is abnormal*$)$, is calculated as $1 - distance(C_i(x), f_i(x))$, where $C_i(x)$ is the predicted value and $f_i(x)$ is the actually observed value (lines 6–12). The *distance* between two values for a single continuous feature is the difference in values divided by the observed range of that feature's value (i.e., the maximum distance is 1.0), and the distance for a discrete feature is the Hamming distance (i.e., 1 if the feature values are different and 0 if they are identical). To get the total probability of a vector x to represent an abnormal event, we make a naïve assumption that all sub-models are independent and multiply all the individual probabilities computed for each one of the feature values (line 14). We utilize this method due to its simplicity and computational efficiency, despite the known incorrectness of the underlying independence assumption. Utilization and analysis of more sophisticated methods is one of our future tasks. A threshold that distinguishes between normal and anomalous vectors is learned during the algorithm calibration phase conducted on the datasets with labeled samples (lines 15–16).

However, detection of abnormality in a single observed instance is not sufficient to determine whether that application's behavior has been meaningfully changed. Such point anomalies can be caused by changes or noise in a user's behavior. In order to reduce the false alarm rate and improve the effectiveness of the proposed system in general, one of our goals is to define a procedure which considers the consequent observations and derives a decision comprised of the individual predictions for each one of these observations. For example, an alarm can be dispatched only when an anomaly has been detected in a certain number of consequent instances (e.g., 3 consequent instances were detected as anomalous) or when an anomaly has been detected in a certain percentage of instances during a specified time period (e.g., 3 or more anomalies during a 10 min interval). The exact procedure has been determined according to our observations during the evaluation experiments and is described in the next section.

Evaluation

This section presents the initial analysis of applications' traffic patterns and an evaluation of the proposed detection system. The research questions that we attempt to answer are described in Research questions section. Data collection section describes the data collected for the experiments. In Visual traffic pattern analysis of various applications section we present a visual traffic pattern analysis observed for several popular applications. In Calibration experiments section the experiments conducted for system calibration are presented.

Finally, the results observed for the three types of the tested software, regular (benign) applications, self-written malware and real malware, are presented in Classifying different versions of the same application section and Detecting self-updating malware section.

Research questions

The performed evaluation experiments aimed to answer the following research questions:

1. Is it possible to model an application's network behavior so that any deviation from its normal behavior would be detected?
2. Which network application-level features are most useful for modeling an application's network traffic patterns?
3. Which classification algorithm is the most effective for learning the models and detection of anomalous behavior when most of the features are numeric values?
4. What level of detection accuracy (and false alarms) could be reached using application-level network-behavioral features only?
5. How much overhead on mobile phone resources is caused by applying machine-learning and detection algorithms directly on the device?

Data collection

For the initial analysis of applications' traffic patterns and system calibration, the Features Extractor module was installed and ran on the personal Android devices of eight volunteer users. During this period the applications' features were extracted and aggregated, as described in Feature extractor section and Feature aggregator section. We collected data from each user's device for varying periods of time starting from two weeks to three months.

Additionally, we experimented with ten self-written and five real malware. Each evaluated malware has two versions: the original application asking for network access permission for benign purposes (such as displaying advertisements, best scores updates, etc.) and the repackaged version of the original application with injected malware code performing network connections for malicious purposes. Experiments with the malware applications were conducted on the dedicated devices. The detection was performed and evaluated on the same device the models were learned on.

Real malware

We used five infected applications and their benign versions for the experiments with the real malware. The infected applications and the corresponding versions of the benign application were retrieved from a repository collected by crawling the official and various alternative Android markets for over a year and a half. In total, our current repository contains over 500,000 various Android applications. We used two applications injected with PJApps (Symantec: Android.Pjapps) Trojan; Fling and CrazyFish, two applications injected with Geinimi (Symantec: Android.Geinimi) Trojan; Squibble Lite and ShotGun, and one sample of DroidKungFu-B (McAfee: Android/DroidKungFu.B) malware found within the OpenSudoku game.

The PJApps Trojan which was discovered in applications from unofficial Android marketplaces creates a service that runs in the background, sends sensitive information containing the IMEI, Device ID, Line Number, Subscriber ID, and SIM serial number to a Web server, and retrieves commands from a remote command and control server.

The Geinimi Trojan is installed on the device as part of repackaged version of legitimate applications. Applications repackaged with Geinimi Trojan have been found in a variety of locations, including unofficial marketplaces, file-share sites, and miscellaneous websites. When installed, the Trojan attempts to establish contact with a command and control server for instructions and once the contact is established, it transmits information from the device to the server and may be instructed to perform certain actions.

The DroidKungFu-B is a version of the DroidKungFu malware. The initial DroidKungFu malware is known for its capability of rooting Android phones with OS 2.2 or below. The infected applications were found among alternative Android markets targeting the Chinese audience. The DroidKungFu-B version targets already rooted phones and requests for the root privilege. Both versions collect and steal phone information (e.g., IMEI, phone model, etc.).

Note that although the above mentioned real Trojans are not of the self-updating type, they can easily be upgraded to this type in the future. Moreover, following the successful debut of the Android.Dropdialer on the official Google's Android marketplace, new Trojans with functionalities similar to those described above as well as other self-updating capabilities are expected to appear very soon. Thus, we use these real malware applications for the evaluation of our system's detection abilities and manually simulate the self-updating process.

The malware applications and their benign counterparts were executed on a specially designated device and their behavior has been collected and analyzed.

Self-developed malware

Malware applications with the advanced self-updating capabilities have just started to appear and there are not yet enough known real malware samples of this type. Thus, for the purposes of this paper, we have created the self-update malware using methods 1 and 2 described in Self-updating malware section, and infected several open-source applications with these packages. The utilized open-source applications are: APG (APG application), K-9 Mail (K-9 Mail), Open WordSearch (Open WordSearch application), Rattlesnake Free (Rattlesnake Free application) and Ringdroid (Ringdroid application). The APG, the Android Privacy Guard application, provides OpenPGP functionalities, such as encryption and signing of emails. It uses network connections for public and secret keys management. K-9 Mail is an open-source email client for Android. Open WordSearch is a game application that uses network connections to synchronize global high scores. Rattlesnake Free is also a game application utilizing network connections for online advertisements. Ringdroid is an application for recording and editing sounds and creating ringtones directly on the Android phone. It uses network connections to share ringtones and other sounds created by users. Each one of these applications was infected and evaluated using the created malware of both types. To simulate malicious behavior within the created malware, we chose to implement some simple malicious behavior patterns of known malware such as stealing a user's contacts list, recent calls details and user's GPS location which are sent out to a remote server.

An application infected with the malware component of method 1 will present an "update is available" notification to the user when the corresponding command is received by the device. When a user agrees to install the update, it will download a malicious version of the same application from a remote server and replace the benign version with the malicious one. At this stage the user is presented with a list of permissions to be granted to the new application version, which actually could differ from those granted to the original application. Once installed, the new malicious version will wait for an external command. When the command is received, it steals the user's contacts list and sends it to a remote server.

An application infected with the malware component of method 2 will silently download a precompiled malicious payload, when the corresponding command is received by the device and then continue to load and execute malicious code without any notification to the user. The malicious payload will first steal the user's contact list and send it to a remote server and then continue to report the user's location and recent calls details to the server at every specified time period (set to 2 min for our experiments).

Visual traffic pattern analysis of various applications

This section presents the traffic patterns observed while analyzing the collected data from several popular benign applications with heavy network usage such as Facebook, Skype, Gmail, and WhatsApp, as well as the traffic patterns observed from the benign and malicious versions for two of the evaluated applications. Fig. 3 presents the network behavior of the benign applications collected from the devices of different users. The data points of different users are plotted in different colors. Although the graphs are presented in two dimensions only, average sent vs. average received bytes, the distinguishable patterns of each application are clearly highlighted. Note also that all the dimensions (i.e., features) were considered during the model's learning and detection processes, as described in Local models learning section.

As can be seen from the graphs, each one of the analyzed applications has its own specific traffic pattern which is easily distinguishable from other applications (note that on each of the graphs, the axis value's range is different). In less certain cases, other features can be utilized for differentiation.

The two graphs presented in Fig. 4 depict the behavior of different applications of the same type. Fig. 4(a) depicts the traffic pattern of two email client applications: Gmail and Android's native email client. Fig. 4(b) depicts the traffic pattern of two Internet browsers: Mozilla Firefox and the device's native browser application. Data points of different applications are plotted in different colors.

It can be seen from the graphs that different applications from the same functionality type have very similar traffic patterns among them, while the traffic patterns of various application types are different.

Fig. 5 depicts the network traffic patterns of the ShotGun application, its version infected with Geinimi Trojan, the network traffic patterns of the K-9 Mail client and its version infected with our self-updatable malware of type 2. On all the graphs the data points of malicious and benign versions are plotted in different colors. The presented dimensions are: (1) average sent vs. average received bytes; (2) average sent vs. average received data in percent; (3) inner sent vs. inner received time intervals; and (4) outer sent vs. outer received time intervals. The data of both benign and malicious versions of the applications were taken from the same device.

As can be seen from the graphs, the distinguishable patterns of different application versions can be clearly identified on most of the compared dimensions. For example, the amount and percentage of sent\received data by malicious versions of the ShotGun application are very different from those of the original one. Similarly, the amount of sent\received data and outer time intervals of the infected K-9 Mail are very different from those of the original application. Such distinguishable patterns can be identified on several dimensions of all other evaluated applications. Although certain dimensions of malicious and benign versions may lie in overlapping areas, an anomaly could be detected based on deviations observed in the values of other dimensions. Generally, our algorithm is able to detect an anomaly if meaningful deviations from the learned features correlations in a few dimensions (i.e., one or two features) are observed.

The above observations lead us to the following conclusions:

1. It is possible to model a mobile application's network behavior based only on application-level features;
2. Applications seem to have characterizing patterns of normal behavior that can be learned so that any meaningful deviations from these patterns would be detected;
3. The observed network behavior of an application can be used in order to determine whether the application is what it claims to be, given that the normal behavioral patterns of this application are known;
4. Certain types of applications have similar network traffic patterns which can be used, for example, for traffic classification or hierarchical clustering of applications;
5. Repackaged applications containing malware functionality produce different network patterns that can be useful for the detection of self-updating malware.

Calibration experiments

The first set of experiments was conducted for calibrating the proposed system, and served the following purposes: (1) selection of optimal features subset; (2) evaluation of several machine-learning algorithms as the cross-feature analysis method's base learners; (3) determination of the minimal sufficient training set size; and (4) determination of the strategy for raising the *anomaly* alarm in case one or more

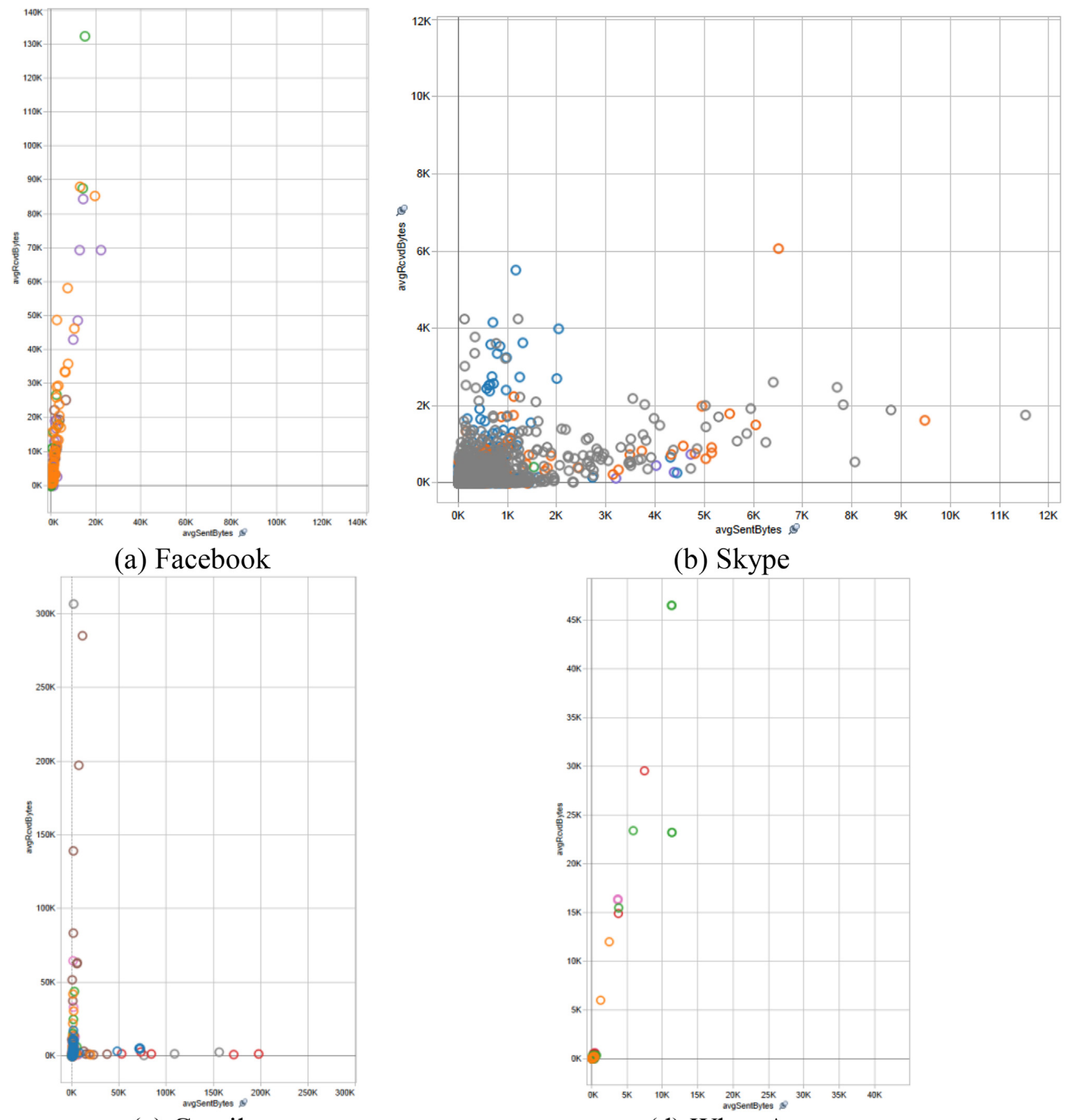


Fig. 3 – Comparing the network traffic behavior of different benign applications collected from the devices of different users. The data points of different users are plotted in different colors. Although the graphs are presented in two dimensions only, average sent vs. average received bytes, the distinguishable patterns of each application are clearly highlighted. It can be seen that each one of the analyzed applications has its own specific traffic pattern which is easily distinguished from other applications (note that on each of the graphs, the axis value's range is different). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

anomalous records are detected. Based on the results of the calibration phase in the second phase (the test phase) we evaluated the ability of the proposed system to distinguish between benign and malicious versions of the same application and between two different benign versions of the same application. For evaluation of different classification algorithms and features selection, the following standard measures were employed: True Positive Rate (TPR) measure (also known as Detection Rate), which determines the proportion of correctly detected changes from an application's normal

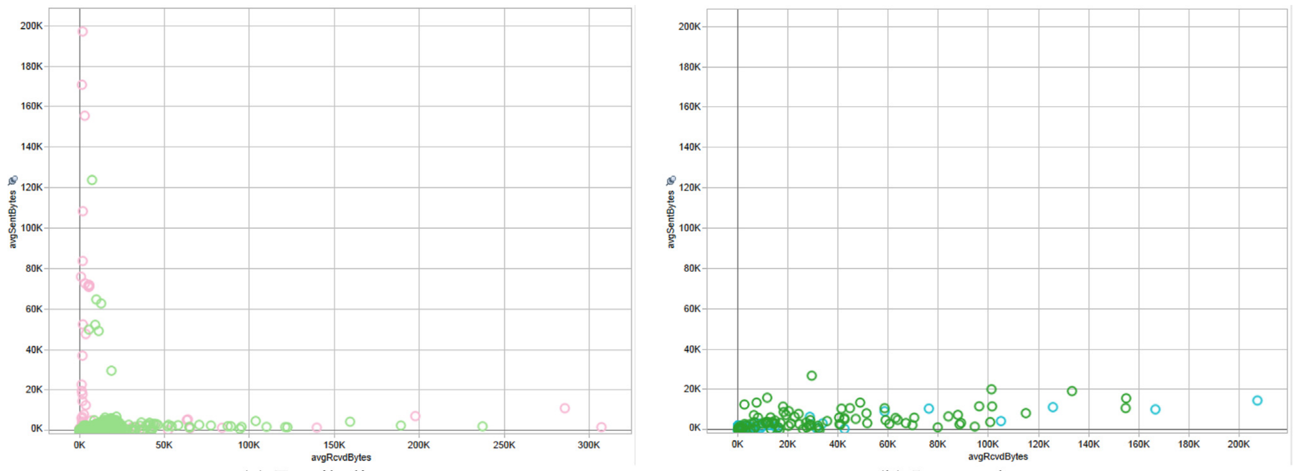


Fig. 4 – The two graphs compare the traffic of different applications of the same type: (a) traffic pattern of two e-mail client applications: Gmail and Android's native email client; (b) traffic pattern of two Internet browsers: Mozilla Firefox and the device's native browser application. Data points of different applications are plotted in different colors. It can be seen from the graphs that different applications from the same functionality type have very similar traffic patterns among them, while the traffic patterns of various application types are different. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

behavior; False Positive Rate (FPR) measure (also known as False Alarm Rate), which determines the proportion of mistakenly detected changes in an actually normal application behavior; and Total Accuracy, which measures the proportion of a correctly classified application behavior as either anomalous or normal.

In this section we first describe the calibration experiments and their results.

For the calibration experiments 16 datasets were extracted and prepared from the data collected for the following 8 applications: twitter, groupme, gmail, facebook, twitter, firefox, whatsapp, and linkedin. Each one of the 16 datasets consists of train and test records. The datasets were selected and prepared in the following way: in 8 datasets both the train and test records were taken from the *same version* of a certain application. These datasets were used to verify a low detection rate on the records of the same application and determine the deviation level in traffic patterns that can be attributed to the application diversity and changes in a user's behavior. In the other 8 datasets train and test records were taken from *different versions* of a certain application. These datasets were used to verify the higher detection rate seen in the cases with the same application version. However, in some cases, the low detection rate for the different application versions is acceptable as different application versions are not obligated to contain any network-related updates. For both the calibration and test experiments, the train size for all applications was limited to a *maximum* of 150 instances, and the test size to a *maximum* of 400 instances. On datasets with fewer available examples, the full train and test sets were utilized.

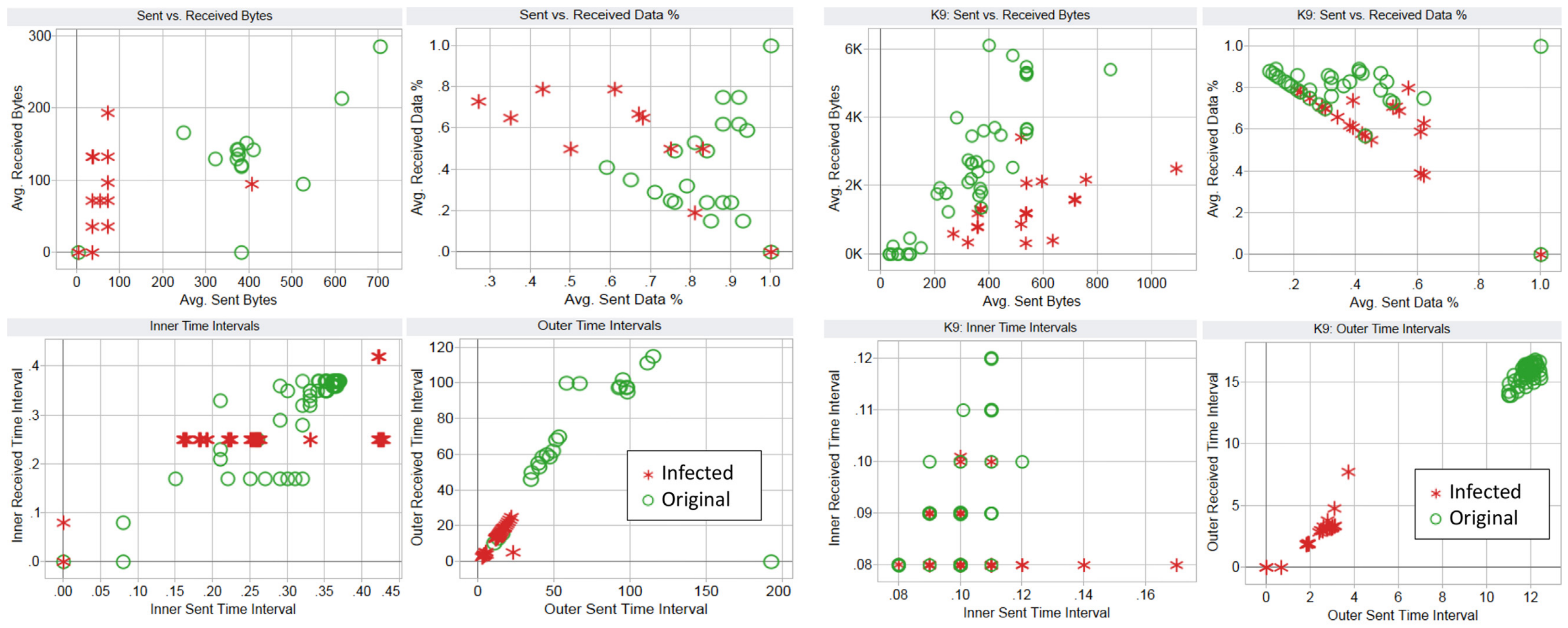
Features selection

A wide range of features has been defined and presented in System overview section. However, selecting a subset of features is essential for the following reasons: (1) extraction and aggregation of a large number of features on a mobile device is a very inefficient and resource wasting process; (2) a learning process involving a large number of features is much more computationally expensive; and (3) the presence of redundant or irrelevant features may decrease the accuracy of the learning algorithm.

Thus, our purpose in this task is to identify the minimal and most useful set of features. There are several groups of features among the defined list of aggregated features for which extraction and calculation can be performed together such that if one feature in a group is extracted, all other features can be extracted as well with minimal marginal effort (resources). Thus, removing one or a few features from a group from which at least one feature is calculated will not reduce the extraction and calculation overhead significantly. The standard approaches for features selection, such as Filter and Wrapper, are not applicable in this case, as they cannot consider these dependencies among the features. For this reason, 20 feature subsets of various sizes were manually defined; each subset containing different groups of features. These groups of features were defined such that if one feature in a group is extracted, all other features in the same group can be extracted as well with marginal minimal effort. The threshold distinguishing between the normal and anomalous vectors was defined separately for each one of the features subset in the preliminary calibration experiments, as it depends on the number and type of the involved features.

Evaluated base learners

Considering the prevalence of numerical attributes among the defined aggregated features, as well as the resource consumption parameter, we decided to evaluate the following classifiers as candidates for the cross-feature analysis method



(a) ShotGun (b) K-9 Mail client

Fig. 5 – Traffic patterns of the ShotGun and K-9 Mail client applications depict the network traffic patterns of the ShotGun application, its version infected with Geinimi Trojan, the network traffic patterns of the K-9 Mail client and its version infected by our self-updatable malware of type 2. On all the graphs the data points of malicious and benign versions are plotted in different colors. The presented dimensions are: (1) average sent vs. average received bytes; (2) average sent vs. average received data in percent; (3) inner sent vs. inner received time intervals; and (4) outer sent vs. outer received time intervals. The data of both benign and malicious versions of the applications were taken from the same device. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 1 – Selected features subsets.

Features subset #1	Features subset #2
Avg. Sent Bytes	Avg. Sent Bytes
Avg. Rcvd. Bytes	Avg. Rcvd. Bytes
Pct. of Avg. Rcvd. Bytes	Pct. of Avg. Rcvd. Bytes
Inner Avg. Send Interval	Inner Avg. Send Interval
Inner Avg. Rcvd. Interval	Inner Avg. Rcvd. Interval
Outer Avg. Send Interval	Outer Avg. Send Interval
Outer Avg. Rcvd. Interval	Outer Avg. Rcvd. Interval
–	Avg. Sent data Percent
–	Avg. Rcvd. data Percent

base learners: Linear Regression, Decision Table, Support Vector Machine for Regression, Gaussian Processes for Regression, Isotonic Regression, and Decision/Regression tree (REPTree). The Weka (Weka 3) open-source library was used for evaluation of these algorithms. All the defined feature subsets were tested with all of the evaluated base learning algorithms on the calibration datasets described above.

As was previously mentioned, sometimes abnormal instances are caused by either changes in the user's behavior or by diversity in an application's functionality. In order to determine the acceptable rate of such abnormal instances in a normal application's behavior, noted as '*anomaly acceptance*' rate, we have evaluated each combination of machine-learning algorithm and features subset, using five levels of acceptance rate – 5, 10, 15, 20, and 25 percent. For example, an acceptance rate of 5% means that we would expect to get up to 5% of abnormal instances while monitoring regular application behavior.

The results of the calibration experiments reveal the two best combinations of the base learning algorithms and two feature subsets. The two best base algorithms are the Decision Table and the REPTree. The two best feature subsets, presented in Table 1, are very similar to each other; one of the subsets includes all the features of the other set plus two additional features.

As can be seen from Table 1, there are 7 features included in both of the best subsets. Thus, we can conclude that these features are the most useful for modeling an application's network traffic.

As for the estimated algorithm accuracy performance, the Decision Table algorithm in conjunction with the features subset #1 and "anomaly acceptance" rate of 20 percent result in TPR = 0.8, FPR = 0, and Total Accuracy = 0.875, and the REPTree algorithm, in conjunction with the features subset #2 and "anomaly acceptance" rate 25 percent demonstrate exactly the same accuracy values.

For a better insight into the detection rate observed in the calibration datasets, the results of the Decision Table algorithm in conjunction with the features subset #1 and the REPTree algorithm in conjunction with the features subset #2 are presented in Table 2 (errors are marked in bold).

It can be seen that for most of the different application versions the detection rate is above the determined "anomaly acceptance" rate of 20–25 percent for both algorithms. At the same time, the detection rate on the test sets from the same application version is always below 20 percent. Thus, the detection strategy consisting of several steps can be defined as follows: (1) raise an alarm if at least 3 consequent abnormal instances are detected; (2) raise an alarm if at least 3 abnormal instances are detected among the 5 consecutive observations; (3) raise an alarm if at least 3 abnormal instances are detected among the 10 consecutive observations. According to this strategy, a system will raise an alert about any meaningful changes in an application's network patterns, including those caused by a version update. Furthermore, the version update can be verified within the mobile OS and the alert with the relevant information (including abnormal instances rate, whether a version update was detected and at what point in time) can be presented to the user.

Table 2 – Detection rate on calibration datasets.

User	Application name, version and number of instances		Detected anomalous records (%)	
	Train set	Test set	Decision table	REPTree
Different application versions				
U1	twitter v103 (50)	twitter v320 (31)	60.9	91.3
U3	groupme v2 (124)	groupme v4 (47)	74.5	82.9
U3	gmail v1 (508)	gmail v2 (202)	**5.0**	**11.9**
U2	facebook v1 (686)	facebook v2 (855)	25.8	**17.3**
U2	twitter v1 (456)	twitter v2 (450)	**1.6**	26.0
U2	firefox v2 (80)	firefox v3 (183)	26.8	32.8
U2	whatsapp v1 (89)	whatsapp v3 (778)	29.2	44.4
U2	linkedin v1 (35)	linkedin v2 (25)	32.0	48.0
Same application version				
U1	twitter v201 (30)	twitter v201 (16)	0.0	6.7
U3	groupme v2 (99)	groupme v2 (30)	6.7	10.0
U3	gmail v2 (152)	gmail v2 (50)	16.0	8.0
U3	facebook v2 (653)	facebook v2 (456)	1.3	3.9
U2	twitter v2 (451)	twitter v2 (100)	1.0	14.0
U2	firefox v2 (60)	firefox v2 (20)	20.0	20.0
U2	whatsapp v2 (578)	whatsapp v2 (200)	7.5	13.5
U2	whatsapp v3 (844)	whatsapp v3 (200)	10.5	6.5

Table 3 – Detection rate on test datasets.

User	Application name, version and number of instances		Detected anomalous records (%)	
	Train set	Test set	Decision table	REPTree
Different versions				
U1	twitter v103 (42)	twitter v201 (45)	57.8	62.2
U1	twitter v201 (45)	twitter v320 (23)	78.2	34.8
U3	**facebook v1 (529)**	**facebook v2 (1109)**	**0.5**	**3.3**
U3	groupme v1 (82)	groupme v4 (47)	80.9	87.2
U2	whatsapp v2 (778)	whatsapp v3 (1044)	**16.7**	28.9
Same version				
U3	groupme v1 (62)	groupme v1 (20)	0.0	0.0
U3	groupme v4 (27)	groupme v4 (20)	0.0	15.0
U3	gmail v1 (400)	gmail v1 (108)	14.8	22.2
U2	facebook v2 (450)	facebook v2 (405)	16.0	15.7
U2	firefox v3 (35)	firefox v3 (148)	20.0	22.8

Training set sizes analysis
An important question regarding the proposed detection system is how quickly the detection can be started (i.e., how many examples are sufficient for the learning of the network traffic patterns)? To answer this question we have evaluated the two winning algorithms using train sets of various sizes. This experiment was executed on all the calibration datasets, varying the train set size from 10 to 100 or the maximum of the available instances with step 10, and from 100 to 400 with step 25.

The results with both algorithms show that, in most cases, the train size of 30–50 examples is sufficient for learning a stable model which is able to determine the level of deviation between an application's traffic patterns correctly. However, it has been found that in several cases among diverse applications such as Facebook and Gmail, a greater number of instances (80–150 examples) is required for learning a stable model. Considering the fact that in the current experiments each data instance represents 1 min of an application's network usage, we conclude that a relatively short time, varying from 30 min to 2.5 h of network activity is required for our system to learn the patterns of a new application. Note that certain applications with rare network usage may actually require much longer periods of time while the required amount of network behavior data is aggregated.

Additionally, we hypothesize that the minimal size of the training set required for learning stable models can be automatically predicted based on the meta-features of the observed data (i.e., range of values, variance, etc.) Investigation in this direction will constitute a part of our future work.

Classifying different versions of the same application

The main goal of this experiment is to evaluate the ability of the proposed method to distinguish between different versions of the same application; i.e., identifying different, unique, traffic patterns of the different versions.

In this set of experiments 10 datasets were used (5 with train and test records from the same application version and 5 with train and test records from different application versions). These datasets are different from the datasets used in the calibration experiments presented in Calibration experiments section.

Table 3 presents the detection rate of the Decision Table and REPTree algorithms in conjunction with the features subset #1 and #2 correspondingly on the evaluated datasets (detection errors are marked in bold).

Deviations at various levels were detected in most cases when the learned models were tested with instances from a different application version. The undetected versions of Facebook and WhatsApp applications can be explained by very few or no network-related changes in the considered application versions. Additionally, the detection rate for all the cases when the learned models were tested with instances from the same application version are below the defined "anomaly acceptance" rate of 20 percent for the Decision Table algorithm and of 25 percent for the REPTree algorithm. Thus, using the Decision Table algorithm's "anomaly acceptance" rate of 20 percent, the estimated method's accuracy on the test data is the following: TPR = 0.82, FPR = 0.0 and Total Accuracy = 0.875. For the REPTree algorithm with the determined "anomaly acceptance" rate of 25 percent, the estimated accuracy on the test data is even higher: TPR = 0.91, FPR = 0.0, and Total Accuracy = 0.94.

Detecting self-updating malware

This set of experiments was conducted on an Android smartphone. Our Client application was installed on the device and both the learning and detection were executed on the device. We used the benign and malware samples of several Trojan types, as described earlier. Initially, a benign version of each evaluated application was installed and executed on a device for two days. For further analysis and comparison of benign and malware traffic patterns the benign version was replaced by the malicious one which was executed for at least 1 h, whether or not it was detected earlier. The general detection rate and the time of first raised anomaly alert were recorded and are presented in the Results section. Additionally, during the execution of the benign versions, the False Alarm Rate was recorded and it is also reported in this section.

The system was configured to the parameters selected in our earlier calibration experiments in Calibration experiments

Table 4 – Malware detection results.

Application name			TPR (%)	Detection time	FPR (%)	Wrong anomaly alerts
Train (benign)	Test (infected)	Test (benign)				
Real malware						
Fling (30)	Fling (184)	Fling (144)	67.9	30 min	3.5	0
OpenSudoku (30)	OpenSudoku (49)	OpenSudoku (25)	100.0	2 min	0.0	0
ShotGun (30)	ShotGun (134)	ShotGun (42)	100.0	5 min	3.3	0
Squibble (30)	Squibble (40)	Squibble (19)	95.0	9 min	10.5	1
Crazy Fish (30)	Crazy Fish (117)	Crazy Fish (26)	100.0	5 min	3.7	0
Self-updating malware – type 1						
APG (30)	APG type1 (13)	APG (23)	92.3	3 min	7.9	0
K-9 Mail (30)	K-9 Mail type1 (12)	K-9 Mail (35)	100.0	3 min	4.1	0
WordSearch (30)	WordSearch type1 (13)	WordSearch (6)	100.0	3 min	6.0	0
Rattlesnake (30)	Rattlesnake type1 (13)	Rattlesnake (288)	92.3	3 min	9.8	0
Ringdroid (30)	Ringdroid type1 (10)	Ringdroid (6)	92.6	3 min	0.0	0
Self-updating malware – type 2						
APG (30)	APG type2 (15)	APG (23)	45.0	16 min	5.3	0
K-9 Mail (30)	K-9 Mail type2 (12)	K-9 Mail (35)	91.6	4 min	5.7	0
WordSearch (30)	WordSearch type2 (12)	WordSearch (6)	100.0	4 min	8.3	0
Rattlesnake (30)	Rattlesnake type2 (12)	Rattlesnake (288)	83.3	6 min	12.0	1
Ringdroid (30)	Ringdroid type2 (13)	Ringdroid (6)	92.3	2 min	0.0	0

section. The utilized parameters are as follows. The feature set of 9 attributes (feature set #2 in Calibration experiments section) in conjunction with the Decision/Regression tree (REPTree Weka's implementation) algorithm and probability threshold equal to 0.001, was utilized for the models learning and detection. For learning, the first 30 normal instances (excluding bootstrapping records) of each application were used. The learned models were static and have not been updated during the experiment. Analysis and selection of a policy for the periodical models update is a part of our future work.

The results for all the evaluated benign\malware applications pairs are presented in Table 4. The TPR and FPR columns correspondingly represent the proportion of instances identified as anomalous, correctly or incorrectly. The detection time column presents the period of time passed prior to the malware's first detection (i.e., first correct anomaly alert). The last column specifies the number of wrongly fired anomaly alerts which indicate how many times a user was actually disturbed unnecessarily. Note that according to the selected strategy, an anomaly alert is raised if at least three abnormal instances are detected among the ten consecutive observations.

Note that since the applications infected with the real Trojans were replaced manually on the devices (the self-updating technique was not involved), the specified detection time indicates the time it took from the moment the data theft began, up until the first anomaly alert was fired (which means that at least three abnormal instances have been detected). For the evaluated self-developed Trojans utilizing one of the "remote payload" techniques, the specified detection time indicates the time period from the *infection moment* until the first anomaly alarm was fired.

It can be observed that for almost all malicious applications, a high level of deviation (80–100% of anomalous instances) from the original network behavior was detected. In most cases the threat was identified and reported (using the defined strategy for raising alerts) within the first 5 min after the infection occurred or the data theft began. Additionally, it can be seen that in most cases the false positive rate of the detection algorithm is below 10%. Consequently, a very low level of wrongly raised anomaly alerts to the user was achieved; only two false alerts were fired during the two evaluation days among all 15 of the applications.

Consider the relatively low detection rates, 67.9% and 45%, on Fling and APG (type 2) applications, correspondingly. Note that in the applications infected with the Trojans, the main application's functionality is preserved, and some new functionality is added. Certain traffic patterns related to the old functionality might be expected to remain unchanged. This is actually the case with the Fling and APG applications. For example, in the Fling game, online mobile advertisements are displayed while the application is in the phone's frontend in both versions. Thus, the records corresponding to the time when the game was actually played were not as much affected by the Trojan functionality, and thus, the observed detection rate is "only" 68%. The situation is similar for the APG application. Note that lower detection rates on these applications cause a longer detection time. However, even in these cases, the threats were detected relatively quickly; the latest detection has occurred on the Fling application 30 min after the data steal began.

Analysis of the data aggregated from the benign and malicious versions of the evaluated applications, presented in the previous section, shows that the significant differences are caused by a background process in malicious versions which is running even when an application is not active. This behavior has a significant effect on such features as average sent\received bytes, number of data sent\received events, outer and inner sent\received intervals, etc. Most of these features are utilized by our method and this explains the overall high detection rate.

Resources overhead

This section evaluates the overhead caused by the learning and detection processes on mobile phone resources in terms of memory consumption, CPU load, and time needed for a model's induction as well as vector testing processes. Experiments were performed on a Samsung Galaxy S GT-i9000 running Android OS version 2.2. One of the selected combinations, the REPTree algorithm in conjunction with the features subset #2 was used for the overhead evaluation experiments.

Note that online monitoring is performed only for network-active applications. Generally, there are no more than 2–3 such applications running simultaneously on a device. Additionally, we assume that during the time periods of a user's normal activity, the number of such applications may reach no more than 10–15 network-active concurrent processes. Thus, for performance estimation, we consider a scenario of 10 concurrently monitored applications. We estimate the memory and CPU load for learning the 10 application models and further constant monitoring of their network traffic. For a better estimation of memory consumption, the results were averaged through 10 distinct experiments.

Memory consumption

The memory consumption of the application changes in intervals from 7035 KB ± 8 before the learning process has started to 7272 KB ± 15 after storing the 10 learned models in memory (which is approximately 1.4% of the device's RAM). The storage of each additional model in memory consumes about 24 KB ± 0.7 on average. The memory consumption observed for several constantly running Android services and other popular applications is presented below: Android System – 24,375 KB; Phone Dialer – 8307 KB; Anti-virus – 7155 KB; TwLauncher – 22,279 KB; Email – 10,611 KB; and Gmail – 9427 KB. The detection process has no effect on the consumed memory.

CPU consumption

The CPU consumption peaks occurred at the times of the actual models learning and were in the interval of 13% ± 1.5. Note that models learning operations occur very rarely, either when a new application is installed or when a model's update is needed (due to a new application version or changes in user's behavior). The CPU consumption observed during the process's idle time was in the interval of 0.7% ± 1.02. Time needed to learn a model (using 50 training examples) varies in intervals of 249 ms ± 27.4.

The time needed for testing a single instance varies in intervals 3.6 ms ± 2.5. Recall that aggregated features vectors are tested once at the defined aggregation time interval (1 min for these experiments). The CPU consumed by testing 10 concurrent instances (1 for each one of the assumed active applications) varies in intervals of 1.8% ± 0.8.

Note that the results of this experiment depict the resource overhead caused during the user's high activity time periods. During the presumably much longer time periods of the user's normal activity there expected to be even lower overhead is expected.

Discussion

In this section the defined research questions are discussed in light of the experimental results. Considering the first question, it has been shown that modeling a mobile application's network behavior using application-level features only is possible and that the proposed approach is feasible; patterns of the normal application's behavior for an application can be learned and then any meaningful deviations from these patterns can be accurately detected.

In regard to the second question, 7 network application-level features have been found as the most useful for modeling an application's network traffic patterns. Some further fine-tuning of evaluations to add\remove certain features could have probably slightly improve the achieved results.

As for the most effective classification algorithm to be used as a base learner in the "cross-feature analysis" approach, the REPTree and Decision Table algorithms were found to be the most successful. The high detection accuracy utilizing these algorithms was confirmed on the test data as well.

Considering the forth question, it was shown that a high true positive rate, along with zero false alarms, could be achieved using the selected classification algorithms and features subset. An interesting question for future research is whether this accuracy could be improved using the network-level features in addition to the currently selected features.

We have also demonstrated that, compared to the resources consumed by other permanently running services, the proposed online learning and detection has relatively low overhead on the mobile phone resources. Thus, the proposed system is acceptable for running on smartphones. Unfortunately, we could not evaluate the Features Extractor and the aggregation processes impact on the mobile phone resources due to the fact that an extended list of features was observed and calculated. In our future work we plan to re-implement the Features Extractor components retaining only the most effective features. We expect that in this way the performance overhead of the whole system can be determined.

Summary and conclusions

In this paper we presented a novel system for detecting meaningful deviations in a mobile application's network traffic patterns that can be used for detecting an emerging type of malware with self-updating capabilities that allow for stealing user data or spying on users. The presented system, although initially planned as a host-based part of a client-server system, is a fully-functioning, stand-alone monitoring application for mobile devices, which can be used alone or in conjunction with other methods. One of the main capabilities of the proposed system is protection of mobile device users from malicious attacks on their phones. The detection is performed based only on the application's network traffic patterns.

Machine-learning methods were used for learning and detection purposes. This work represents one of the first attempts to run learning and detection processes on the device itself, which demonstrates the feasibility and acceptable resource overhead of the proposed method. Although the overhead of the Features Extractor process was not yet measured, this is an unavoidable part of a host-based system, and thus, those processes should be of maximal efficiency at all times. An estimation of the overhead of the whole system is one of our upcoming tasks.

Experimentally, a subset of a few network application-level features most useful for modeling traffic patterns was identified among the wide range of the extracted and aggregated features. Additionally, several classification algorithms suitable for handling numerical data were evaluated and the two most effective methods were selected. Moreover, the traffic patterns of several popular applications were presented and analyzed. It was shown that many applications have very specific network traffic patterns and that certain application categories can be distinguished by their network traffic patterns.

Results of the evaluation experiments conducted with different benign and malware applications (including self-updating versions) demonstrate that a high true positive rate, along with low false alarms, could be achieved using the proposed method. Specifically, it was shown that different levels of deviation from normal behavior can be detected accurately. Thus, deviations of up to 20–25 percent were observed in tests with the same application versions. These deviations can be explained by different user's behavior and the diverse functionality provided by certain applications. Deviations in various ranges from 0, up to almost 90 percent of instances might be observed in the different versions of an application. Such a wide range of deviations is explained by different levels of network-related changes among new versions. Deviations of 60 percent and more were observed in the applications containing an injected malware. Summarizing the results of the presented research, we conclude that the proposed method is feasible and effective for the detection of different deviation levels in application network patterns.

Advanced malware that attempt to hide their activity within legitimate activity in order to "stay below the radar" pose a great challenge to the security community. In case that a malware will try to hide by using encryption, our system will still be able to detect the anomaly since it analyzes the volume, rate and frequency of the traffic rather than the content. More advanced methods that attempt to behave like a legitimate application are difficult to detect. One method that can be used for making it more difficult for an attacker to evade the machine-learning detection mechanism is randomly choosing each time a different set of features to learn from. Each model on different devices will be trained using a different subset of features which eventually would make it more difficult for the malware to adapt. Finally, as mentioned before, methods employed in this paper for an application's network monitoring are generic and can be further utilized to monitor other aspects of smartphone usage (such as processor, memory or file system) for protection against other types of malware threats. In such a case, choosing different types of features will be more effective and will allow addressing additional types of malware.

Some of our main future research directions are: verifying whether the detection accuracy can be improved using other application- and network-level features; automatic prediction of the minimal training set size sufficient for learning stable models, based on the meta-features of the observed data; as well as development and analysis of methods for a collaborative model learning and detection.

REFERENCES

Adobe blogs. http://blogs.adobe.com/psirt?s=android.

Alessandro R, Fattori A, Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. Prague, Czech Republic: EUROSEC; 2013.

Amer S, Hamilton J. Intrusion detection systems (IDS) taxonomy – a short review. Def Cyber Secur 2010;13(2).

APG application. http://www.thialfihar.org/projects/apg/.

Bose A, Hu X, Shin KG, Park T. Behavioral detection of malware on mobile handsets. In: Proceedings of the 6th international conference on mobile systems, applications, and services; 2008. pp. 225–38.

Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st workshop on security and privacy in smartphones and mobile devices; 2011.

Chandola V, Banerjee A, Kumar V. Anomaly detection: a survey. ACM Comput Surv 2009;41(3):1–58.

Cheng J, Wong SH, Yang H, Lu S. SmartSiren: virus detection and alert for smartphones. In: Proceedings of the 5th international conference on mobile systems, applications and services; 2007.

Dai S, Liu Y, Wang T, Wei T, Zou W. Behavior-based malware detection on mobile phone. In: International conference on wireless communications networking and mobile computing; 2010. pp. 1–4.

Damopoulos D, Menesidou SA, Kambourakis G, Papadaki M, Clarke N, Gritzalis S. Evaluation of anomaly based IDS for mobile devices using machine learning classifiers. Secur Commun Netw 2012;5(1):3–14.

Dini G, Martinelli F, Saracino A, Sgandurra D. MADAM: a multi-level anomaly detector for Android malwareIn Computer network security. Berlin Heidelberg: Springer; 2012. pp. 240–53.

Eagle N, Pentland AS. Reality mining: sensing complex social systems. Pers Ubiquit Comput 2006;10(4):255–68.

Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information-flow tracking system for real time privacy monitoring on smartphones. In: Proceedings of the 9th USENIX conference on operating systems design and implementation, Berkeley, CA, USA; 2010.

Felt AP, Finifter M, Chin E, Hanna S, Wagner D. A survey of mobile malware in the wild. In: Proceedings of the 1st workshop on security and privacy in smartphones and mobile devices; 2011.

Garcia-Teodoro P, Diaz-Verdejo J, Macia-Fernandez G, Vazquez E. Anomaly-based network intrusion detection: techniques, systems and challenges. Comput Secur 2009;28(1):18–28.

Huang YA, Fan W, Lee W, Yu PS. Cross-feature analysis for detecting ad-hoc routing anomalies. In: Proceedings of the 23rd international conference on distributed computing systems. Washington, DC, USA: IEEE Computer Society; 2003.

K-9 Mail. http://code.google.com/p/k9mail/.

Khune RS, Thangakumar J. A cloud-based intrusion detection system for Android smartphones. In: International conference on radar, communication and computing, India; 2012. pp. 180–4.

Li F, Clarke NL, Papadaki M, Dowland PS. Behaviour profiling on mobile devices. In: International conference on emerging security technologies; 2010. pp. 77–82.
McAfee: Android/DroidKungFuB.http://home.mcafee.com/virusinfo/virusprofile.aspx?key=522721.
Moreau Y, Verrelst H, Vandewalle J. Detection of mobile phone fraud using supervised neural networks: a first prototype. In: Proceedings of the 7th international conference on artificial neural networks; 1997.
Mylonas A, Kastania A, Gritzalis D. Delegate the smartphone user? Security awareness in smartphone platforms. Comput Secur; 2012.
Mylonas A, Theoharidou M, Gritzalis D. Assessing privacy risks in Android: a user-centric approach; 2013.
Noto K, Brodley C, Slonim D. Anomaly detection using an ensemble of feature models. In: Proceedings of the 10th IEEE international conference on data mining; 2010.
Open WordSearch application. http://www.brendandahl.com/wordsearch.
Portokalidi G, Homburg P, Anagnostakis K, Bos H. Paranoid Android: versatile protection for smartphones. In: Annual computer security applications conference; 2010.
Portokalidis G, Homburg P, Anagnostakis K, Bos H. Paranoid Android: versatile protection for smartphones. In: Proceedings of the 26th annual computer security applications conference. ACM; 2010. pp. 347–56.
Qian F, Wang Z, Gerber A, Mao Z, Sen S, Spatscheck O. Profiling resource usage for mobile applications: a cross-layer approach. In: Proceedings of the 9th international conference on mobile systems, applications, and services, Maryland, USA; 2011.
Rastogi V, Chen Y, Enck W. AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the 3rd ACM conference on data and application security and privacy; 2013. pp. 209–20.
Rattlesnake Free application. http://www.jamesbecwar.com/rattlesnake/.
Ringdroid application. http://code.google.com/p/ringdroid/.
Schmidt AD, Schmidt HG, Clausen J, Yüksel KA, Kiraz O, Camtepe A, et al. Enhancing security of Linux-based Android devices. In: Proceedings of the 15th international Linux Kongress; 2008.
Schmidt AD, Peters F, Lamour F, Scheel C, Camtepe SA, Albayrak S. Monitoring smartphones for anomaly detection. Mob Netw Appl 2009;14(1):92–106.
Shabtai A, Kanonov U, Elovici Y. Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. J Syst Softw 2010;83(8):1524–37.
Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. Andromaly: a behavioral malware detection framework for Android devices. J Intell Inf Syst 2012;38(1):161–90.
Shamili AS, Bauckhage C, Alpcan T. Malware detection on mobile devices using distributed machine learning. In: Proceedings of the 20th international conference on pattern recognition. Washington, DC, USA: IEEE Computer Society; 2010. pp. 4348–51.
Sophos. Security threat report 2014. Technical report; 2014.
Symantec blog. http://www.symantec.com/connect/blogs/androiddropdialer-identified-google-play.
Symantec: AndroidGeinimi. http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99&tabid=2.
Symantec: android.Pjapps. http://www.symantec.com/security_response/writeup.jsp?docid=2011-022303-3344-99&tabid=2.
Weka 3: data mining software in java.http://www.cs.waikato.ac.nz/ml/weka/.
Xie L, Zhang X, Seifert J-P, Zhu S. pBMDS: a behavior-based malware detection system for cellphone devices. In: Proceedings of the 3rd ACM conference on wireless network security; 2010.
Zhang Y, Yang M, Xu B, Yang Z, Gu G, Ning P, et al. Vetting undesirable behaviors in Android apps with permission use analysi. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security. ACM; 2013. pp. 611–22.
Zhao M, Zhang T, Ge F, Yuan Z. RobotDroid: a lightweight malware detection framework on smartphones. J Netw 2012;7(4):715–22.

Dr. Asaf Shabtai is a lecturer at the Dept. of Information Systems Eng. at Ben-Gurion University. Asaf is a recognized expert in information systems security and has led several large-scale projects and researches in this field. His main areas of interests are computer and network security, machine learning, data mining and temporal reasoning. Asaf has published over 30 refereed papers. Asaf holds a B.Sc. in Mathematics and Computer Science (1998); B.Sc. in Information Systems Engineering (1998); an M.Sc. in Information Systems Engineering (2003) and Ph.D. in Information Systems Engineering (2011) all from Ben-Gurion University.

Lena Tenenboim-Chekina is a Data Scientist, Machine Learning and Data-Mining specialist. At the time of conducting the present research she was a post-doc researcher at Telecom Innovation Laboratories at Ben-Gurion University. Lena Tenenboim-Chekina holds a Ph.D. and M.Sc. in ISE from Ben-Gurion University. Her main research interests include classification and clustering algorithms, ensemble methods, time series analysis, information retrieval and text mining.

Dudu Mimran is the CTO of Deutsche Telekom Labs in Israel and has experience of more than 22 years in managing projects in the field of security, mobile and large-scale systems.

Lior Rokach Founder of the Machine Learning Research Laboratory at Ben-Gurion University, Prof. Lior Rokach promotes innovative adaptations of machine learning and data-mining methods to create the next generation of intelligent information systems. An active entrepreneur with several patents to his credit, Rokach has worked with several multinational companies and governmental agencies. He has made significant contributions to the field of cyber security by developing machine-learning algorithms that are capable of identifying malwares, preventing data leakage, detecting anomalies and protecting user data and privacy. He is an awardee of the 2012 BGU Toronto Prize for young researchers.

Bracha Shapira is an associate professor at the Department of Information Systems engineering at Ben-Gurion University (BGU) where she heads the department. Her research interests include text analysis, information retrieval, personalization, user profiling, and recommender systems, especially their application to security and privacy. She has established the IR Lab at BGU and leads funded research, specifically, innovative application of IR and personalization to provide solutions for information security and privacy. Prof. Shapira has published more than 80 papers and was the inventors of some patents. During the last 10 years she has guided more than 20 research students.

Prof. Yuval Elovici is the director of the Telekom Innovation Laboratories at Ben-Gurion University of the Negev (BGU), head of the Cyber Security Labs at BGU, and a Professor in the Department of Information Systems Engineering at BGU. He holds B.Sc. and M.Sc. degrees in Computer and Electrical Engineering from BGU and a Ph.D. in Information Systems from Tel-Aviv University. Prof. Elovici has published 56 articles in leading peer-reviewed journals. In addition, he has co-authored a book on social network security and a book on information leakage detection and prevention. His primary research interests are cyber security and machine learning.