Detecting Application Update Attack on Mobile Devices through Network Features

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

Department of Information Systems Engineering and Telekom Innovation Laboratories
Ben-Gurion University of the Negev, Beer-Sheva, Israel

***Abstract*— Recently, a new type of mobile malware applications with self-updating capabilities was found on the official Google Android marketplace. Malware applications of this type cannot be detected using the standard signatures approach or by applying regular static or dynamic analysis methods. In this paper we first describe and analyze this new type of mobile malware and then present a new network-based behavioral analysis for identifying such malware applications. For each application, a model representing its specific traffic pattern is learned locally on the device. Machine-learning methods are used for learning the normal patterns and detection of deviations from the normal application's behavior. These methods were implemented and evaluated on Android devices.**

***Index Terms*—network traffic, anomaly-detection, machine-learning, Android malware, smart-phones security.**

I. Introduction

Recently, a new type of mobile malware hosted on the official Google Android marketplace, the Google Play Store, was detected. The main feature which distinguishes this Trojan (named Android.Dropdialer) from earlier known malware, is its self-updating capabilities. Applications infected by this Trojan and hosted on the Google Play Store were absolutely benign by themselves and did not contain any malware. The package containing malicious payload was downloaded from the Internet sometime after the market application was installed on the device. This allowed the applications to stay undiscovered on the market for several weeks and to generate tens of thousands of installations. In general, any malware can be downloaded and executed on a device using such a "remote payload" technique. The download action can be scheduled for any specific or random time in the future, or even be initiated remotely by sending a command message to the devices.

Such self-updating malware cannot be detected by standard static code analysis techniques as the original version of the application is absolutely benign by itself. Detection by dynamic analysis techniques can be simply avoided by using a time delayed or filtered deployment of the malicious payload. It is also difficult to identify this type of emerging malware since the self updating technique is often used by legitimate applications for the benign purposes as well (upgrade installed games with new levels, bug fixes etc.) A recent survey shows that 70% of known mobile malware steals user information or credentials [1]. Therefore, in this paper we aim to detect the self-updating malware types which steal user data or allow spying on users, and thus we focus on monitoring applications network behavior.

II. Self-Updating Malware Types

Four main techniques can be used to create self-updating applications for Android that download new pieces of software stored remotely. These techniques are especially attractive for malware developers: (1) offer the user an update (i.e. complete replacement) to the original application; (2) dynamic loading of a compiled Android code (i.e., executable DEX files) using Android's DexClassLoader class and allowing the execution of code not installed as part of the application; (3) dynamic loading of a binary shared object file (also called .so library) or an executable file containing native (i.e., machine) code which can be executed using Java's Runtime class; and (4) dynamic loading of a certain file (e.g., mp3, jpg, flash, and pdf) containing a malicious payload (i.e., shellcode) and executing it by exploiting vulnerabilities in the system libraries or external applications handling the file type.

Contrary to the earlier proposed methods our method performs anomaly detection using only *application-level network traffic features*. In addition, *both the learning and the detection processes utilize the machine-learning algorithms that are performed locally on the device* by a stand-alone application which works from the regular user space and can be simply installed on a device at any time. 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 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. 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.

III. Proposed Method

The proposed method uses several network-based features that are collected during the application runtime. Feature measurements are performed at fixed time intervals (of 5 seconds) and then various aggregation functions are computed over these measurements at each specified time interval (set to 1 minute). Based on the results of our previous work [2] a subset of nine most useful features is used.

For the detection a model representing the monitored application specific network traffic patterns is derived using the *cross-feature analysis* approach [3]. The basic idea of the cross-feature analysis method is to explore the correlation between one feature and all the other features. It assumes that in normal behavior patterns strong correlations between features exist which can be used for detecting deviations caused by abnormal activities. Formally, cross-feature analysis approach tries to solve the classification problem 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 $i = 1, \ldots L$. Thus, an ensemble of learners for each one of the features represents the model through which each new vector of features is tested for "normality". For the online analysis, each one of the instance features is predicted by the corresponding classification\regression model C_i using the values of all other features. The more different the predictions are from the true values of the corresponding features, the more likely that the observed instance comes from a different distribution than the training set (i.e., represents an anomaly event). Thus, the probability of the value to come from a normal event is calculated as the following:

$$P(f_i(x)\ is\ normal) = 1 - distance(C_i(x), f_i(x)),$$

where $C_i(x)$ is the predicted value and $f_i(x)$ is the actual observed value. The *distance* between two values for a single numeric feature is the difference in actual and predicted values divided by the mean of the observed values for that feature. If the difference is higher than the mean value, the distance is assigned with a constant large value (such as 0.999). 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 calculate the total probability of a vector x to represent an abnormal event, we make a naïve assumption about the sub-model's independence and multiply all the individual probabilities computed for each one of the feature values. Note that we utilize this method, despite the known incorrectness of the underlying independence assumption, as it has demonstrated sufficient performance in our previous experiments [2]. A threshold distinguishing between normal and anomalous vectors is learned during algorithm calibration on the data sets with labeled samples.

IV. Evaluation

For the evaluation of the proposed system we experimented with 5 real and 10 self-written Trojan malware. Each evaluated application has two versions: the original benign application and a repackaged version of the original application with injected malware code. These settings represent a regular infection case where malware developers use existing popular applications for quick spread of their malware. All real applications used: Fling, CrazyFish (PJApps Trojan), Squibble Lite, ShotGun (Geinimi Trojan), and OpenSudoku (DroidKungFu-B), exploit network communication for various purposes. Additionally we have created the malware packages using types 1 and 2 of the "self-updating" behavior described above and infected several open-source applications with these packages. The utilized open-source applications are: APG, K-9 Mail, Open WordSearch, Rattlesnake Free and Ringdroid. To simulate malicious behavior we choose to implement malicious behavior of known malware: (1) an application infected with the malware component of type 1 steals the user's contacts list and sends it to a remote server; and (2) an application infected with the malware component of type 2 first steals the user's contacts list, sends it to a remote server and then continue to report the user's location and recent call details to the server every two minutes. The malware and their benign counterparts were executed on a specially designated device and their behavior was collected and analyzed. The traffic patterns observed from the benign and malicious versions of the ShotGun application are presented in Fig. 1. As can be seen from the graphs, the distinguishable patterns of different application versions can be clearly identified on most of the compared dimensions. Such distinguishable patterns were identified for all other evaluated applications.

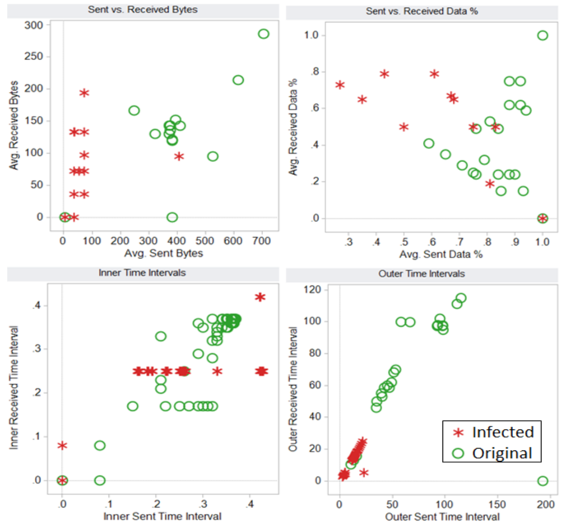

Fig. 1. Traffic patterns of the ShotGun application.

The results for all the evaluated benign\malware applications pairs are presented in Table 1. It can be seen that in most cases the threat was identified and reported within the first five minutes after the infection occurred (detection time column). Also a very low level of wrongly raised alerts (last column) to the user was achieved due to the selected alerting strategy (raise the anomaly alert if at least three abnormal instances are detected among the ten consecutive observations) - only two false alerts were fired during the two evaluation days among all 15 of the applications. In addition, the estimation of the consumed CPU and memory indicates low overhead comparable with other popular Android applications.

TABLE I. Malware Detection Results

Application	TPR (%)	Detection time	FPR (%)	Wrong alerts
Real malware				
Fling	64.7	30 min.	2.8	0
OpenSudoku	100.0	2 min.	0.0	0
ShotGun	100.0	5 min.	4.8	0
Squibble	95.0	9 min.	15.8	1
Crazy Fish	100.0	5 min.	7.7	0
Self-updating malware – type 1				
APG	84.6	3 min.	0.0	0
K-9 Mail	100.0	3 min.	4.1	0
WordSearch	100.0	3 min.	6.3	0
Rattlesnake	92.3	3 min.	9.8	0
Ringdroid	80.0	5 min.	0.0	0
Self-updating malware – type 2				
APG	46.7	16 min.	0.0	0
K-9 Mail	91.7	4 min.	5.7	0
WordSearch	100.0	4 min.	8.3	0
Rattlesnake	83.3	6 min.	12.0	1
Ringdroid	92.3	4 min.	0.0	0

[1] A.P. Felt, et al., "A Survey of Mobile Malware In The Wild," 1st Workshop on Sec. & Privacy in Smartphones and Mobile Devices, 2011.

[2] L. Chekina, et al., "Detection of Deviations in Mobile Applications Network Behavior," available on http://arxiv.org/corr/home

[3] Y.-A. Huang, et al., "Cross-feature analysis for detecting ad-hoc routing anomalies," Int. Conf. on Distributed Computing Systems, 2003.