Spotting the Malicious Moment: Characterizing Malware Behavior Using Dynamic Features

Alberto Ferrante*, Eric Medvet†, Francesco Mercaldo‡, Jelena Milosevic*, Corrado Aaron Visaggio‡
*Advanced Learning and Research Institute, Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
{alberto.ferrante, jelena.milosevic}@usi.ch
†Department of Engineering and Architecture, Università di Trieste, Trieste, Italy
emedvet@units.it
‡Department of Engineering, Università del Sannio, Benevento, Italy
{fmercaldo, visaggio}@unisannio.it

***Abstract*—While mobile devices have become more pervasive every day, the interest in them from attackers has also been increasing, making effective malware detection tools of ultimate importance for malware investigation and user protection.**

Most informative malware identification techniques are the ones that are able to identify where the malicious behavior is located in applications. In this way, better understanding of malware can be achieved and effective tools for its detection can be written. However, due to complexity of such a task, most of the current approaches just classify applications as malicious or benign, without giving any further insights.

In this work, we propose a technique for automatic analysis of mobile applications which allows its users to automatically identify the sub-sequences of execution traces where malicious activity happens, hence making further manual analysis and understanding of malware easier. Our technique is based on dynamic features concerning resources usage and system calls, which are jointly collected while the application is executed. An execution trace is then split in shorter chunks that are analyzed with machine learning techniques to detect local malicious behaviors. Obtained results on the analysis of 3,232 Android applications show that collected features contain enough information to identify suspicious execution traces that should be further analysed and investigated.

I. Introduction

Mobile devices have become of ultimate importance in our everyday life. We use them for communication with other individuals, for storing private data, and for performing financial transactions. Together with the increased usage of these devices, the interest of attackers to abuse them has been increasing as well. This can be observed in the expansion of reported threats and attacks, in particular of malware. By malware we consider any malicious software that gains access to a device for the purpose of stealing data, damaging the device, or annoying the user [1].

During the first quarter of 2015, G DATA security experts found 440,267 new malware files. This represents an increase of 6.4% compared to the fourth quarter of 2014 (413,871). Consequently, the number of new malware applications has risen by 21% compared to the first quarter of 2014 (363,153) [2], i.e., in average the analysts identified a new malware sample every 18 seconds, which means approximately 200 new Android malware samples per hour. In Q2 2015, Kaspersky Lab mobile security products detected 291,887 new malicious mobile applications; this represents a 2.8-fold increase on Q1 2015. The number of malicious installation packages detected was 1,048,129, which is seven times more than in the previous quarter [3]. Additionally, according to the 2016 Trend Micro Security Predictions [4], 3 out of 4 applications currently used in China contain malware.

To protect our privacy and enable secure usage of devices, effective malware detection methods are needed. Static detection, that is commonly used, is based on the investigation of static features that are observed before running the application. These techniques are efficient on mobile devices, but alone they may not be sufficiently effective in detecting malware, due to malware obfuscation techniques that are being adopted [5]. In order to be resistant to a variety of malware that exists on the market, dynamic detection that is based on the investigation of dynamic features is needed.

Most of the currently proposed dynamic detection methods provide the ability to classify applications as malicious or benign, without providing any insight on which parts of the application executions are actually malicious. We propose a dynamic detection method that can identify those malicious parts (we call them *sub-traces*) during the applications executions. Our proposed method jointly uses several dynamic features related to system calls, memory usage, and CPU usage, to identify those traces in which the application behave maliciously. We use system timestamps to connect our dynamic features, thus having knowledge about the behavior of different parts of the system at any given time.

In a nutshell, our solution consists of three steps. First, we split the execution trace under analysis in fixed-length sub-traces. Second, we use the features obtained from CPU and memory observations in each sub-trace to classify the sub-trace in one among a set of possible local behavior. Finally, we analyze the features derived from the system call sequence corresponding to the sub-trace, with another classifier tailored to sub-traces with similar local behavior. Later, during execution of applications, we classify sub-traces based on learned information, and we mark those that are potentially malicious. Being able to identify sub-traces in which a malicious behavior is detected may greatly foster the malware analysis by security experts by helping in training the detection algorithms and in better understanding malware behavior in general.

The rest of the paper is organized as follows. In Section II we discuss the related work. In Section III we present our method in detail. In Section IV we describe our experimental evaluation, along with the way we collected the data. In Section V we describe the results and discuss about possible improvements of our work. Finally, in Section VI, we draw the concluding remarks.

II. Related work

The existing malware detection methods make use either static or dynamic features. In the remaining part of this section, we survey recent works in the two aforementioned categories of detection. A deeper and broader analysis about issues and techniques for securing the Android platform may be found in two extensive and recent surveys [6, 7].

A. Static Detection Methods

An effective approach to static malware detection is proposed in [8] where high detection accuracy is achieved by using features from the manifest file and feature sets from disassembled code. The analysis of disassembled code is used also in [9, 10, 11, 12]. The cited papers rely on the analysis of opcodes. In the former, the occurrences of opcode n-grams are used, by means of machine learning, to classify apps as benign or malicious and the effectiveness of the proposed method is evaluated also with respect to the specific malware family. In [10, 11, 12] the features are given by opcode frequencies, which are also used to determine if an app was retrieved from the official market or third-party one.

A static feature which has been often used to characterize malware is app permissions. In [13], the authors consider sets of required permissions: their security rules classify applications based on sets of permissions rather than individual permissions to reduce the number of false positives. In [1], sending SMS messages without confirmation or accessing unique phone identifiers like the IMEI are identified as promising features for malware detection as legitimate applications ask for these permissions less often [14]. Still, using only asked permissions, as it is done in [1, 14], is subject to high false positive rates. For example, nearly one third of applications request access to user location but far fewer request access to user location and to launch at boot time. The authors state that more sophisticated rules and classification features are required in the future.

Recently, the possibility to identify the malicious payload in Android malware using a model checking based approach has been explored in [15, 16]. Starting from the payload behavior definition the authors formulate logic rules and then test them by using a real world dataset composed by real-world malicious samples.

A different feature has been considered in [17]: the authors analyze the set of identifiers representing the applications that are being executed on the device. The assumption is that such information may give an indication of the likelihood of the device being classified as infected in the future. Nevertheless, a conclusion of the paper is that observing just this feature is not enough to give a precise answer about the device being infected or not.

The main disadvantage of approaches based on static detection is that they are prone to obfuscation, and as such they alone are not enough to detect malware anymore [5]. Additionally, they are not able to detect malware at run-time, during its execution. Dynamic detection methods appear as promising approaches that can cope with these problems.

B. Dynamic Detection Methods

Several detection approaches based on dynamic analysis of resources have been proposed for mobile devices: some of them consider several resources while others focus on few or a single resource (e.g., power consumption). From a broader point of view, several techniques which are not specific to mobile device can be used for malware detection on such devices: a comprehensive and detailed description of those techniques and tools is provided in [18].

1) Approaches Based on Power Consumption: Power consumption, monitored through battery usage, has been found to be a promising feature for malware detection. Among the different solutions, VirusMeter [19], monitors and audits power consumption on mobile devices with a power model that accurately characterizes power consumption of normal user behaviors. However, to what extent malware can be detected on phones by monitoring just the battery power remains an open research question [20].

2) Approaches Considering CPU and Memory Features: Several resources are monitored by the tool proposed in [21], Andromaly, which considers: touch screen, keyboard, scheduler, CPU load, messaging, power, memory, calls, operating system, network, hardware, binder, and leds. The cited work compares different machine learning techniques for analyzing the features obtained from the monitored resources: Random Forest and Logistic Regression appear to be the most effective. The results are obtained using 40 benign applications and four malicious samples developed by the authors.

A similar approach is proposed in [22], where feature selection is performed on a set of run-time features related to network, SMS, CPU, power, process information, memory, and Virtual memory. Features importance is estimated using Information Gain and four different classification algorithms evaluated, Random Forest being the one that provides the best performance. Results are obtained by considering only 30 benign and 5 malicious applications, with a limited coverage of the high variety of malware available to date. This may limit the applicability of results, both in terms of algorithm and of features selected.

A quite large set of features related to CPU, memory, storage, and network are used in [23]: the focus of the cited paper is more on a precisely defined acquisition procedure than on the analysis itself, which is performed using machine learning techniques. The authors obtain an accuracy detection of 99% on a dataset of 2,000 apps.

Other recent works, presented in [24] and in [25], also take into account memory and CPU features for malware detection. In [24], most significant CPU and memory related features are extracted for each considered malware family; some features appear as good candidates for malware detection in general, some features appear as good candidates for detection of specific malware families, and some others are simply irrelevant. In [25], instead, importance of memory and CPU features for identification of malicious execution traces is investigated. Different detection algorithms of low complexity are tested and Logistic Regression is identified as the best one in distinguishing between benign and malicious execution records.

3) Approaches Based On System Calls: Many detection approaches have been developed which work on features deriving from system calls, or sequences of system calls, occurrences or frequencies. CopperDroid [26] recognizes malware samples through a system calls analysis with a customized version of the Android emulator in order to track system calls. In Wang et al. [27], an emulator is used to perform a similar task: the method is assessed on a dataset composed of 1,600 malicious apps; 60% of the malicious apps belonging to the Genoma Project as well as the 73% of the malicious apps included in the Contagio dataset are identified correctly.

Canfora et al. [28] propose a method to detect Android malware based on three metrics, which evaluate the occurrences of a reduced subset of system calls, a weighted sum of a subset of permissions required by applications, and a set of combinations of permissions. In their experiment a sample of 200 malicious apps and 200 benign apps are considered; a 74% precision in the identification of malware is obtained.

In [29], system calls generated in background when apps are stimulated through their user interface are monitored. The Android emulator is used for running experiments and their evaluation is based on two malicious samples of DroidDream family ("Super History Eraser" and "Task Killer Pro").

Beyond system calls, other OS-related features have been largely investigated for malware detection. The detection method presented in [30] uses data gathered by an application log and a recorder of a set of system calls related to management of file, I/O and processes. A physical device with a modified Android 2.1 is used for the experiments and 230 applications, in greater part downloaded from Google Play, were considered; among them, the method is able to detect 37 applications which steal some kind of personal data, 14 applications which execute exploit code, and 13 destructive applications.

Researchers in [31] hook system calls to create, read/write operations on files to detect malicious behavior. Their technique hooks system calls and a binder driver function in the Android kernel. The authors used a customized kernel on a real device, and the evaluated dataset includes two malicious apps developed ad-hoc. Similarly, Schmidt et al. [32] used the view from Linux-kernel such as network traffic, system calls, and file system logs to detect anomalies in the Android system.

Monitoring of system calls in Android is also discussed by Blasing et al. [33]. They perform both static and dynamic analysis, disposing a module which monitors system calls and logs the return value of each system call.

Authors of [34] propose a method to perform automatic classification based on tracking system calls while applications are executed in a sandbox environment, obtaining an accuracy of 93% with a 5% benign application classification error.

Armando et al. [35] built a kernel module able to retrieve system calls invoked by application framework layer. Their module re-executes the tracked calls: the collected information may indicate that little control is enforced between the Android and the Linux layers, thereby suggesting that the attack surface of the Android platform is wider than expected.

In [36] an accurate approach to malware detection that also uses system calls is presented. The approach uses machine learning to discover connections between malicious behavior (e.g., sending high premium rate SMS or ciphering data for ransom) and their execution traces and then exploit obtained knowledge to detect malware. As opposed to other systems, where a limited set of system calls is taken into account, in this work, all system calls are considered so as their sequences. The approach is tested with data coming from a real device, with a dataset consisting of 20,000 execution traces and 2,000 applications, and obtains a detection accuracy of 97%.

4) Approaches Considering CPU and Memory Features Along With System Calls: MADAM [37] is an Android malware detector that concurrently monitors Android at kernel and at user level; machine learning techniques are used to distinguish between standard and malicious behaviors. At kernel-level, features related to the following system parameters are extracted: system calls, running processes, free RAM and CPU usage. At user-level, features related to the following system parameters are considered: idle/active, key-stroke, called numbers, sent/received SMS and Bluetooth/Wi-Fi analysis. The developed prototype is able to detect several real malware applications found in the wild with a low number of false positives.

Our approach takes into account only memory, CPU, and system calls. By combining them, we are able to provide a system that is more resistant to obfuscation. At the same time, the proposed method potentially allows for a simpler implementation, since the actual analysis of system calls is triggered only when memory- and CPU-related information suggest that a given sub-trace deserves further attention.

As opposed to MADAM, we use only memory and CPU kernel-level features. Additionally, we focus on malicious sub-traces identification instead of currently commonly performed malicious application identification.

III. Detection Method

The aim of the work is to propose a method that, based on dynamic features, identifies suspicious execution sub-traces of Android applications and enables their further classification into *benign* or *malicious*. Dynamic features taken into account are CPU and memory usage by the application under analysis, as well as statistics on system calls caused by the same application. During application executions we use timestamps for memory, CPU, and system calls related features in order to connect them: this allow us to have information about behavior of different parts of the system during the execution time. While these features in the literature were observed separately, to the best of our knowledge, their combined usage together with execution time information and in order to detect suspicious sub-traces, was not investigated yet.

In the state-of-the-art works it is shown that high detection accuracy can be achieved using system calls. However, using system calls for identification of malicious sub-traces is a challenging task due to complexity of their analysis and to the difficulty of identifying anomalous parts in sequences of system calls consisting of system call names and their attributes. In order to overcome this problem, and to identify which parts of system calls traces are actually anomalous, we connect them with memory and CPU features, that are real-valued numbers and can be easily grouped into clusters with similar behavior. In this way, by having clusters of similar memory and CPU behavior, we have connected also system calls of similar behavior, being able to further investigate them. Based on the information of collected system calls, we train classifiers in the way it was proposed in [36] and then, during the application execution, by observing their memory and CPU behavior, we classify the sub-traces with learned classifiers. The main steps of the proposed methodology consist of the application execution, the sub-traces collection, the sub-traces clustering based on similarity between memory and CPU information, and the learning of one classifier for each cluster; an overview of our methodology is outlined in Figure 1.

We assume that no labeled training data is available which specify exactly the sub-traces which are actually malicious; instead, we assume that only a collection of traces which are atomically malicious or not-malicious is available—the formers possibly and likely consisting of several non-malicious sub-traces. On one hand, this assumption enhances the practicality of our proposed solution, since it can be used on existing datasets in which a single binary label is available for each trace. On the other hand, it makes the problem more challenging because it is posed as a classification scenario in which the training data is only weakly labeled or, from another point of view, heavily noisy.

More in detail, in our method, given an execution of an application, a *trace* $t = (o_1, o_2, \dots)$ of observations is available. Each *observation* o is a tuple composed of:

- A timestamp,
- 3 values for CPU usage (total, user, kernel)
- 5 values for total memory usage (PSS, shared, private, heap allocation, free heap)
- 5 values for Dalvik memory usage (PSS, shared, private, heap allocation, free heap)
- 5 values for Native memory usage (PSS, shared, private, heap allocation, free heap)
- A sequence s of system calls generated in the time frame delimited by the current observation timestamp and the previous observation timestamp

We denote an observation as $o = (\vec{f}_{\text{CM}}, s)$, where $\vec{f}_{\text{CM}} \in \mathbb{R}^{19}$ is the numeric vector consisting of the 19 values concerning CPU and memory, which are extensively discussed in Section IV-B. The *system calls sequence* s is a string defined over an alphabet of $63+1$ symbols, each one corresponding to a system call but one symbol, which is used to represent all the other system calls. The set of the 63 system calls has been chosen statically by selecting the most occurring system calls performed by Android applications (as in [36], where they provide high detection accuracy). Names of considered system calls are listed in Section IV-B.

We focus on traces in which the frequency of observations is $0.5\,\text{Hz}$ (i.e., one observation every $2\,\text{s}$): the number of system calls generated during the corresponding time frame is in the order of some hundreds.

We assume that a *learning set* $\mathcal{L}$ of labeled traces is available to tune the classification method. The learning set is composed by pairs (t, l) where T is a trace and $l \in \{\text{benign}, \text{malicious}\}$ is a label which is benign if the application for which t has been obtained is benign, and malicious otherwise.

The goal of this work is to propose a method which, given a learning set, a trace t, and a time interval T, can for any subsequence t' of t corresponding to consecutive observations collected in a time interval lasting T, indicate if t' corresponds to a malicious or benign behavior of the application during the corresponding interval. We remark, however, that the learning set provides traces labeled atomically (at the level of execution observations), i.e., there is no information about where an actually malicious sub-trace is within each trace of the learning set: it follows that the learning information is only weakly labeled.

The basic idea is to divide a trace in sub-traces and to classify each of them by considering system calls. For this purpose, different classifiers are trained for sub-traces that exhibit specific CPU and memory behaviors.

The method itself consist of a *learning phase*, in which the learning set $\mathcal{L}$ is used to tune several inner parameters, and a *classification phase*, in which a single trace t is analyzed.

In both phases, a trace t is preprocessed in order to decompose it in *sub-traces* $t'_1, t'_2, \dots, t'_k$, each one corresponding to a time interval T. Then, for each sub-trace t', a tuple $(\vec{f}'_{\text{CM}}, s')$ is obtained where $\vec{f}'_{\text{CM}} \in \mathbb{R}^{19\times 4}$ is a vector composed of 4 aggregates computed from the vectors $\vec{f}_{\text{CM}}$ of the observations in t' (see below), and s' is the concatenation of the strings s of the observations in t'. More precisely, $\vec{f}'_{\text{CM}}$ is the concatenation of: the vector $\vec{f}^{\text{last}}_{\text{CM}} - \vec{f}^{\text{first}}_{\text{CM}}$ (i.e., the difference between the vector of the first and the last observation in the sub-trace); the vector $\text{pmax}_{t'} \vec{f}_{\text{CM}}$ of the element-wise max values of vectors in t'; the vector $\text{pmin}_{t'} \vec{f}_{\text{CM}}$ of the element-wise min values; and the vector $\text{pavg}_{t'} \vec{f}_{\text{CM}}$ of the element-wise average values.

A. Learning Phase

The aim of the learning phase is twofold. First, to cluster sub-traces of the labeled applications according to the corresponding behavior of the app in terms of CPU and memory

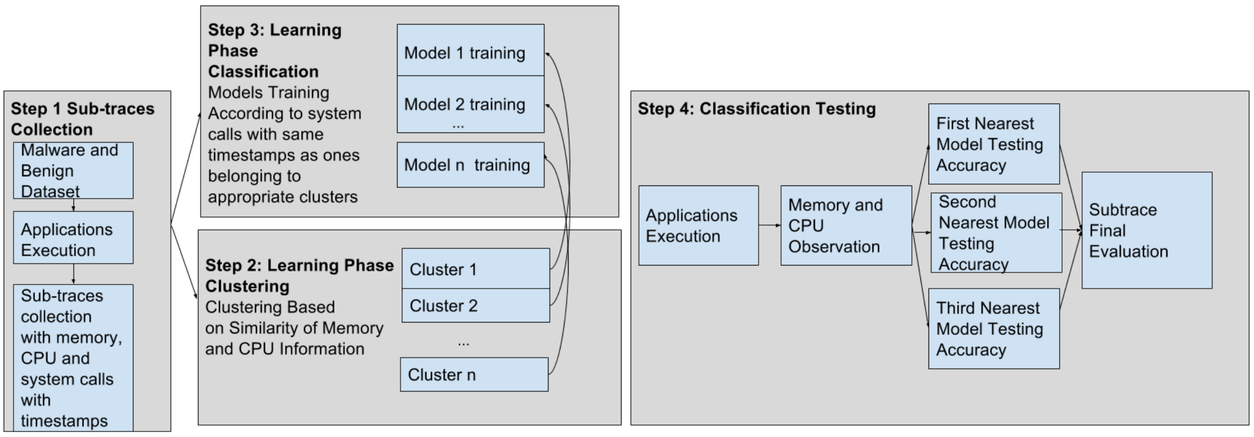
Step 1 Sub-traces Collection
Malware and Benign Dataset
Applications Execution
Sub-traces collection with memory, CPU and system calls with timestamps
Step 3: Learning Phase Classification
Models Training According to system calls with same timestamps as ones belonging to appropriate clusters
Model 1 training
Model 2 training
...
Model n training
Step 2: Learning Phase Clustering
Clustering Based on Similarity of Memory and CPU Information
Cluster 1
Cluster 2
...
Cluster n
Step 4: Classification Testing
Applications Execution
Memory and CPU Observation
First Nearest Model Testing Accuracy
Second Nearest Model Testing Accuracy
Third Nearest Model Testing Accuracy
Subtrace Final Evaluation

Figure 1. Main steps of the proposed detection system.

usage. Second, for each cluster, to train a classifier based on system calls; this classifier should be able to discriminate benign and malicious sub-traces.

Note that the latter task is somewhat hard, since the learning set does not provide labels at the level of the observation, nor of the sub-trace. Moreover, it can be expected that even a trace collected from a malicious application may consist of several sub-traces which do not correspond to any actually malicious behavior. Hence, and from another point of view, the sub-trace classifier has to be trained from a learning set which is intrinsically noisy.

The clustering step is performed using the KMeans++ algorithm [38] with a maximum of 10,000 iterations over all the vectors $\vec{f'_{\text{CM}}}$ in $\mathcal{L}$; note that labels are not used in the clustering step. KMeans++ partitions the observations into k clusters such that each observation belongs to the cluster with the nearest mean that is used as a prototype of the cluster. We experimented with different values for the number of clusters, with $k \in \{3, 5, 7\}$; we also experimented with following three distances: Euclidean, Canberra, and Chebyshev. These distances are explained in more details in Section IV-C.

For each cluster c, a classifier C_c is built using the method proposed in [36] as follows. Let $\mathcal{L}_c$ be the set of pairs (s', l) where s' is the system calls sequence corresponding to the sub-trace t' whose $\vec{f'_{\text{CM}}}$ belong to the cluster, and l is the label in $\mathcal{L}$ of the trace of t'. First, we build the set G_c of all the n-grams occurring in the sequences $\mathcal{L}_c$, an *n-gram* being a substring of up to n symbols, where n is a parameter of the method which we set to 2. Second, for each sequence s' we build a numeric vector $\vec{f_{s'}} \in [0,1]^{|G_c|}$ where each element $f_{i,s'}$ is the relative frequency of the ith n-gram of G_c in s'. Third, we select a subset G'_c of G_c containing exactly m n-grams, m being a parameter of our method which we set to $m = 250$. The aim of this feature selection is twofold: (i) we want to select only the most promising n-grams, with respect to their discriminative power, and (ii) we want to limit the size of the feature space on which a subsequent classifier will operate. The set G'_c is composed of the m n-grams with the largest relative class difference δ_i:

$$\delta_i = \frac{\left| \frac{1}{|\mathcal{L}_c^{\text{benign}}|} \sum_{\mathcal{L}_c^{\text{benign}}} f_{i,s'} - \frac{1}{|\mathcal{L}_c^{\text{malicious}}|} \sum_{\mathcal{L}_c^{\text{malicious}}} f_{i,s'} \right|}{\max_{\mathcal{L}_c} f_{i,s'}} \tag{1}$$

where $\mathcal{L}_c^{\text{benign}}$ and $\mathcal{L}_c^{\text{malicious}}$ are the partition of $\mathcal{L}_c$ containing all and only the element with $l = \text{benign}$ and $l = \text{malicious}$, respectively. Finally, a binary classifier is trained on the instances in $\mathcal{L}_c$ using the features corresponding to n-grams of G'_c; we used a Random Forest (RF) classifier and we experimented with three values for the number of trees: $n_{\text{tree}} \in \{5, 50, 500\}$. Random Forest is a combination of different tree classifiers [39].

For each cluster, we also compute the expected accuracy of a classifier built as described above by means of a cross-folding procedure. That is, we first split $\mathcal{L}_c$ in n_{fold} partitions with equal size and equal proportion of benign and malicious pairs. Then, for each jth partition, we train a classifier on all the other partitions and assess it on the jth partition, in terms of False Positive Rate (FPR, i.e., ratio of benign sub-traces classified as malicious), False Negative Rate (FNR, i.e., ratio of malicious sub-traces classified as benign), and accuracy (i.e., $1 - \frac{1}{2}(\text{FPR}+\text{FNR})$). We set the *cluster accuracy* $A_c \in [0, 1]$ as the average of the accuracies computed over the n_{fold} partitions.

The output of the learning phase is a set of exactly k tuples, each consisting of:

- the cluster centroid, i.e., a vector $\vec{f'_{\text{CM},c}} \in \mathbb{R}^{19\times 4}$;
- the reduced set G'_c of n-grams which has been selected for the cluster;
- the trained classifier C_c;
- the cluster accuracy A_c.

B. Classification Phase

As stated before, the goal of the classification phase is to give an indication on sub-traces t' of traces t lasting time T about whether each t' is related to a benign or malicious behavior during the corresponding interval. We proceed as follows: First, we obtain the list $t'_1, t'_2, \ldots$ of sub-traces of

t and the corresponding $(\vec{f}'_{\text{CM}}, s')$ tuples. Then, for each sub-trace, we obtain the $n_{\text{centroids}}$ clusters closest to $\vec{f}'_{\text{CM}}$, i.e., those for which the considered distance metric between $\vec{f}'_{\text{CM},c}$ and $\vec{f}'_{\text{CM}}$ is the smallest—$n_{\text{centroids}}$ is a parameter of our method which we set to $n_{\text{centroids}} = 3$. Finally, for each of these close clusters, we obtain the numeric vector $\vec{f}_{s'}$ from s' by considering the n-grams in G'_c and then classify $\vec{f}_{s'}$ with C_c. The outcome of the classification is cast as a value in $[0, 1]$ which corresponds to the probability, according to C_c, that $\vec{f}_{s'}$ has a label $l =$ benign . We combine the $n_{\text{centroids}}$ classification outcomes of the sub-trace by means of a weighted average w in which the weights are determined by the corresponding cluster accuracies A_c. The underling motivation is to give more importance to an outcome generated by a classifier which has been trained on a cluster for which the difference between malicious and benign behaviors is sharper. The value of w is defined as an internal parameter in the range of $[0, 1]$ and is closer to 1 when the corresponding sub-trace is deemed to be benign, and closer to 0, otherwise. The cluster accuracies on the identification of malicious sub-traces obtained in our experiments are reported and further discussed in Section V.

IV. Experimental evaluation

In this section, we describe the experimental evaluation that we performed on our detection method. We first provide a description of the data collection process and our experimental setup. Finally, we discuss in detail the features we took into account and the distance metrics we used.

A. Data Collection

1,709 benign applications were automatically collected from Google Play [40] by using a script implemented by the authors; the script queries an unofficial Google Play python API [41] to search and download apps. The downloaded applications belong to different categories (call & contacts, education, entertainment, GPS & travel, internet, lifestyle, news & weather, productivity, utilities, business, communication, email & SMS, fun & games, health & fitness, live wallpapers, personalization). The applications retrieved are among the most downloaded ones in each category and they are free. We chose the most popular apps in order to increase the probability to download malware-free apps. In any case, the benign applications were analysed by the VirusTotal service [42], a service that checks submitted apps by using 57 different antimalwares (e.g., Symantec, Avast, Kasperky, McAfee, Panda): none of our benign apps was identified to include any malicious payload.

The malware dataset consists of 1,523 obtained from the Drebin dataset [43, 44].

Memory and CPU usage traces were recorded by running the applications, one at a time, on the Android emulator and by using a script that recorded features related to memory and CPU usage every two seconds. To collect system calls data we used strace [45], a tool for tracing system calls. In particular, we used the command `strace -p PID` in order to hook the process corresponding to the application under analysis and intercept only its system calls; to retrieve the CPU and memory values we used the `dumpsys CPUinfo` and `dumpsys meminfo` commands, respectively. Dumpsys [46] is an Android tool for dumping information about system status as text. Log files for CPU, memory, and system calls are later unified by using timestamps recorded at execution time. Each application has been run for 10 minutes, even though some of the traces are shorter, due to emulator hiccups. Although it could be the case that a longer execution period would provide us more significant results, we believe that the duration that we have chosen is a good trade-off between time when most of the malware samples expose their malicious intents and duration of the overall experimentation. This is also supported by the obtained results, as discussed in Section V.

The Android emulator of choice was the one included in the Android Software Development Kit [47] release 20140702, running Android 4.0. The reason why an Android emulator was chosen instead of real devices is that this solution minimizes the time necessary to set up each application before running, thus providing the ability to run a large number of applications with the purpose of making the obtained dataset more significant. The Android operating system was each time re-initialized (as if the operating system was reinstalled on the device) before running each application. This guarantees that the system is always in a mint condition when a new sample is started, thus avoiding possible interferences (e.g., changed settings, running processes, and modifications of the operating system files) from previously run samples.

The execution of the applications and the collection of features have been automated by means of a script that have been run on a Linux PC; the script made use of Android Debug Bridge (adb) [48] as well as of the Monkey application exerciser [49]. Android Debug Bridge is a command line tool that lets the PC communicate with an emulator instance or with an Android device. The Monkey is a command line tool that exercises the system with a stream of pseudo-random user events, thus acting as a stress test on the application software; the Monkey can be run on any emulator instance or on a device. In our script, the Monkey was used to activate different parts of the applications; adb was deployed to monitor application features, namely the memory usage of the considered sample, as well as to install the applications. In summary, for each application, the following actions have been performed:

1) Clean-up of the Android operating system
2) Application installation on the Android Emulator by means of adb
3) Memory and CPU monitors setup by starting periodic calls to the `dumpsys CPUinfo` and `dumpsys meminfo` commands
4) Initialization and run of the application for 10 minutes by using Monkey
5) Start of system call recording by means of strace

B. Features

We have chosen a set of memory and CPU related features on the assumption that malicious behavior is thereby reflected; this belief is supported by other works that can be found in the literature, such as [24, 25]. In detail, we have extracted three CPU-related features related to single applications under analysis:

- Total: total percentage of CPU used
- User: percentage of CPU used by user
- Kernel: percentage of CPU used by kernel

For what concerns memory-related features, we have extracted 15 features obtained by monitoring the following types of memory consumption: (i) the Dalvik Virtual Machine; (ii) the native memory usage; (iii) the total memory usage. In particular, for each of these three categories, we have extracted the following features:

- PSS: The total Proportional Set Size (PSS) is the RAM used by proces; it indicates the overall memory weight of your process, which can be directly compared with other processes and the total available RAM
- Shared: Shared Dirty is the amount of shared RAM that will be released back to the system when the process is destroyed; Dirty RAM is represented by pages that have been modified and must stay committed to RAM, since there is no swap in Android
- Private: Private Dirty is the amount of RAM that will be released back to the system when the process is destroyed
- Heap allocation: it represents the RAM actually allocated by Dalvik Virtual Machine allocation for the application under analysis
- Heap free: represents the allocatable RAM (but not allocated yet) by Dalvik Virtual Machine for the application under analysis

All considered memory features are about single applications under analysis.

The set of system calls considered in this work is based on the one successfully used in [36] and it is composed of following system calls: `msgget`, `getpid`, `ioctl`, `recv`, `semget`, `getuid32`, `mprotect`, `SYS_224`, `read`, `syscall_983042`, `write`, `gettimeofday`, `writev`, `sigprocmask`, `mmap2`, `munmap`, `close`, `lseek`, `brk`, `pread`, `fstat64`, `open`, `dup`, `fcntl64`, `stat64`, `getdents64`, `access`, `clone`, `semop`, `getpriority`, `fsync`, `nanosleep`, `_llseek`, `unlink`, `lstat64`, `pwrite`, `chmod`, `rename`, `sched_yield`, `pivot_root`, `mkdir`, `ipc_subcall`, `getsockopt`, `getcwd`, `pipe`, `sched_getscheduler`, `sched_getparam`, `socket`, `uname`, `getgid32`, `getegid32`, `geteuid32`, `ftruncate`, `syscall_317`, `select`, `rmdir`, `connect`, `bind`, `flock`, `setsockopt`, `getsockname`, `kill`, `fork`.

C. Distance Metrics

As mentioned in Section III, we took into account different distance metrics for KMeans++ clustering. Following, we discuss them in more detail.

Euclidean distance is the most commonly used distance metric that represents straight-line distance between two points in an Euclidean space; the distance between two points $\vec{x}$ and $\vec{y}$ is given by $d(\vec{x},\vec{y}) = \sqrt{\sum_i (x_i - y_i)^2}$.

Canberra distance has already been successfully used for intrusion detection, as discussed in [50]; it is given by $d(\vec{x},\vec{y}) = \sum_i \frac{|x_i - y_i|}{|x_i| + |y_i|}$.

Chebyshev distance between two vectors is the greatest of their differences along any coordinate dimension; it is given by $d(\vec{x},\vec{y}) = \max_i |x_i - y_i|$.

V. Results

In order to validate our methodology, we have performed experiments taking into account different distance metrics for clustering stage, different number of clusters and different number of trees for classification stage with Random Forest. More in detail, we have experimented with Euclidean, Canberra, and Chebyshev distance, number of clusters 3, 5, and 7 and number of trees ranging from small (5 trees) to high (500 trees). Concerning the time interval T which define the lasting of a sub-trace under analysis, we set it to $T = 40\,\mathrm{s}$, which corresponds to 20 observations for each sub-trace.

We have used 1,709 benign and 1,523 malicious applications for training and testing, respectively. The evaluation has been performed by using 3 fold cross validation; in each round, sub-traces belonging to 1,000 benign and 1,000 malicious applications have been used as a training set; the remaining ones are used as a test set. This implies that testing is always performed on applications that are previously unseen by the detector. Detection accuracy is obtained by averaging the results obtained in the three rounds. The obtained results in term of sub-trace classification accuracy are shown in Table I. From these results we can draw the following considerations:

- Increasing the number of trees in Random Forest classifier, does not necessary increase detection performance
- Increasing the number of clusters does increase detection performance
- There is no dominantly better distance, although the Eucledian one provides higher accuracy also when smaller numbers of clusters are considered
- The highest accuracy of detection is achieved with both Euclidean and Chebyshev distance with 7 clusters and 50 and 500 trees, respectively

The obtained accuracy varies for different investigated options, with the highest figure being 0.67: while this value in not great in absolute, we should recall that sub-traces have been labelled by implicitly assuming that all the sub-traces of a malicious trace are malicious and all sub-traces of a benign application are benign. In reality, though, not all sub-traces of a malicious application will contain malicious behavior. From this point of view, besides reflecting how accurate is our detection method, accuracy also measures how good our previously mentioned assumption holds. In other words, these results were obtained on the sub-traces of malicious and benign applications which are marked in the same way as

Table I
SUB-TRACE ACCURACY.

Distance	k	n_{tree}	Observation FPR	FNR	Acc.
Euclidean	3	5	0.12	0.61	0.63
Euclidean	3	50	0.33	0.44	0.61
Euclidean	3	500	0.37	0.41	0.61
Euclidean	5	5	0.14	0.53	0.66
Euclidean	5	50	0.17	0.51	0.66
Euclidean	5	500	0.24	0.45	0.66
Euclidean	7	5	0.14	0.58	0.64
Euclidean	7	50	0.28	0.39	0.67
Euclidean	7	500	0.29	0.38	0.67
Canberra	3	5	0.20	0.54	0.63
Canberra	3	50	0.60	0.23	0.58
Canberra	3	500	0.64	0.21	0.58
Canberra	5	5	0.24	0.46	0.65
Canberra	5	50	0.19	0.49	0.66
Canberra	5	500	0.21	0.47	0.66
Canberra	7	5	0.42	0.28	0.65
Canberra	7	50	0.54	0.16	0.65
Canberra	7	500	0.51	0.18	0.66
Chebyshev	3	5	0.07	0.78	0.58
Chebyshev	3	50	0.07	0.70	0.62
Chebyshev	3	500	0.03	0.81	0.58
Chebyshev	5	5	0.43	0.34	0.62
Chebyshev	5	50	0.49	0.29	0.61
Chebyshev	5	500	0.46	0.31	0.61
Chebyshev	7	5	0.54	0.28	0.59
Chebyshev	7	50	0.31	0.35	0.67
Chebyshev	7	500	0.32	0.34	0.67

applications themselves. Knowing that not all malicious sub-traces are actually malicious, we are aware of the fact that our initial set of labels is noisy and this is likely to negatively affect accuracy. Our future steps will be to further improve accuracy of observations detection by excluding those malicious sub-traces that are labelled as such, although malicious payload was not triggered during their execution.

VI. CONCLUDING REMARKS

In this paper we propose, discuss and enclose initial results on a detection method for identifying malicious sub-traces during applications execution. While most of the current state-of-the art approaches focus on classification of complete applications as malicious or benign, our goal was to propose a methodology suitable for identifying those parts of applications that exhibit a malicious behavior, thus enabling their further investigation and better understanding of malware in general.

Our methodology takes into account the behavior of memory, CPU, and corresponding system calls while applications are executed; it uses trained classifiers to recognize which parts of executions are malicious.

We validated our method on a dataset of thousands of real Android applications, part of which have been used for learning and another part for testing: we think that the results are positive and that the method is useful in performing initial marking of malicious sub-traces. Our method can be further improved by reducing noise in labelling sub-traces used in training.

REFERENCES

[1] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey Of Mobile Malware in the Wild," in *1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM).* ACM, 2011, pp. 3–14.

[2] G DATA, "Mobile Malware Report," 2015, online: https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q1_2015_US.pdf.

[3] Kaspersky Lab, "IT Threat Evolution in Q2 2015," 2015, online: https://cdn.securelist.com/files/2015/08/IT_threat_evolution_Q2_2015_ENG.pdf.

[4] "2016 trend micro security predictions: The fine line," Trend Micro, Tech. Rep., October 2015. [Online]. Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-the-fine-line.pdf

[5] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, Dec 2007, pp. 421–430.

[6] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 2, pp. 998–1022, 2015.

[7] D. J. Tan, T.-W. Chua, V. L. Thing *et al.*, "Securing android: a survey, taxonomy, and challenges," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 58, 2015.

[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *NDSS*, 2014.

[9] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Effectiveness of opcode ngrams for detection of multi family android malware," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on.* IEEE, 2015, pp. 333–340.

[10] G. Canfora, F. Mercaldo, and C. A. Visaggio, "Evaluating op-code frequency histograms in malware and third-party mobile applications," in *E-Business and Telecommunications.* Springer, 2015, pp. 201–222.

[11] F. Mercaldo, C. A. Visaggio, G. Canfora, and A. Cimitile, "Mobile malware detection in the real world," in *Proceedings of the 38th International Conference on Software Engineering Companion.* ACM, 2016, pp. 744–746.

[12] G. Canfora, F. Mercaldo, and C. A. Visaggio, "Mobile malware detection using op-code frequency histograms," in *Proceedings of International Conference on Security and Cryptography (SECRYPT)*, 2015.

[13] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *16th ACM*

conference on Computer and communications security (CCS). ACM, 2009, pp. 235–245.

[14] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *2nd USENIX conference on Web application development (WebApps)*. USENIX Association, 2011, pp. 7–7.

[15] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Ransomware steals your phone. formal methods rescue it," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2016, pp. 212–221.

[16] ——, "Download malware? no, thanks: how formal methods can block update attacks," in *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*. ACM, 2016, pp. 22–28.

[17] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya, "The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators," *CoRR*, vol. abs/1312.3245, 2013.

[18] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A Survey on Automated Dynamic Malware-analysis Techniques and Tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/2089125.2089126

[19] L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *12th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2009, pp. 244–264.

[20] M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf, "Mobile security catching up? revealing the nuts and bolts of the security of mobile devices," in *Symposium on Security and Privacy*, ser. SP '11. IEEE Computer Society, 2011, pp. 96–111.

[21] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": A behavioral malware detection framework for android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10844-010-0148-x

[22] H.-S. Ham and M.-J. Choi, "Analysis of android malware detection performance using machine learning classifiers," in *ICT Convergence (ICTC), 2013 International Conference on*, Oct 2013, pp. 490–495.

[23] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Acquiring and analyzing app metrics for effective mobile malware detection," in *Proceedings of the 2016 ACM International Workshop on International Workshop on Security and Privacy Analytics*. ACM, 2016.

[24] J. Milosevic, A. Ferrante, and M. Malek, "What does the memory say? towards the most indicative features for efficient malware detection," in *CCNC 2016, The 13th Annual IEEE Consumer Communications & Networking Conference*, IEEE Communication Society. Las Vegas, NV, USA: IEEE Communication Society, Jan 2016.

[25] J. Milosevic, M. Malek, and A. Ferrante, "A Friend or a Foe? Detecting Malware Using Memory and CPU Features," in *SECRYPT 2016, 13th International Conference on Security and Cryptography*, 2016.

[26] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," *EuroSec, April*, 2013.

[27] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 149–158.

[28] G. Canfora, F. Mercaldo, and C. A. Visaggio, "A classifier of malicious android applications," in *Proceedings of the 2nd International Workshop on Security of Mobile Applications, in conjunction with the International Conference on Availability, Reliability and Security*, 2013.

[29] F. Tchakounté and P. Dayang, "System calls analysis of malwares on android," *International Journal of Science and Tecnology (IJST) Volume*, vol. 2, 2013.

[30] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*. IEEE, 2011, pp. 1011–1015.

[31] Y.-s. Jeong, H.-t. Lee, S.-j. Cho, S. Han, and M. Park, "A kernel-based monitoring approach for analyzing malicious behavior on android," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1737–1738.

[32] A.-D. Schmidt, H.-G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak, "Enhancing security of linux-based android devices," in *in Proceedings of 15th International Linux Kongress. Lehmann*, 2008.

[33] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Malicious and unwanted software (MALWARE), 2010 5th international conference on*. IEEE, 2010, pp. 55–62.

[34] M. Dimjaševic, S. Atzeni, Z. Rakamaric, and I. Ugrina, "Evaluation of android malware detection based on system calls," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[35] A. Armando, A. Merlo, and L. Verderame, "Security issues in the android cross-layer architecture," *arXiv preprint arXiv:1209.0687*, 2012.

[36] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting android malware using sequences of system calls," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. ACM, 2015, pp. 13–20.

[37] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: A multi-level anomaly detector for android malware." in *MMM-ACNS*, vol. 12. Springer, 2012, pp. 240–253.

[38] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algo-*

rithms. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[39] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324

[40] Google, "Google Store," online: https://play.google.com/store.

[41] GitHub project, "Google Play Unofficial Python API," online: https://github.com/egirault/googleplay-api.

[42] Virus Total, "Suspicious Files Analyser," online: https://www.virustotal.com/.

[43] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[44] M. Spreitzenbarth, F. Echtler, T. Schreck, F. C. Freling, and J. Hoffmann, "Mobilesandbox: Looking deeper into android applications," in *28th International ACM Symposium on Applied Computing (SAC)*, 2013.

[45] Linux, "Strace," online: http://linux.die.net/man/1/strace.

[46] Google Inc., "Dumpsys Input Diagnostics," online: https://source.android.com/devices/input/diagnostics.html.

[47] Android Open Source project, "Android Software Development Kit," 2015, online: https://developer.android.com/sdk/index.html.

[48] ——, "Android Debug Bridge," 2015, online: http://developer.android.com/tools/help/adb.html.

[49] ——, "UI/Application Exerciser Monkey," 2015, online: http://developer.android.com/tools/help/monkey.html.

[50] S. M. Emran and N. Ye, "Robustness of chi-square and canberra distance metrics for computer intrusion detection," *Quality and Reliability Engineering International*, vol. 18, no. 1, pp. 19–28, 2002. [Online]. Available: http://dx.doi.org/10.1002/qre.441