Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 159 (2019) 1045–1052

Procedia Computer Science

www.elsevier.com/locate/procedia

23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

Energy Consumption Metrics for Mobile Device Dynamic Malware Detection

Fausto Fasano[a], Fabio Martinelli[b], Francesco Mercaldo[b,a,*], Antonella Santone[a,*]

[a] *Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy*
[b] *Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy*

Abstract

The ineffectiveness of signature-based malware detection systems prevents the detection of malware, even objects of trivial obfuscation techniques, makes mobile devices vulnerable. In this paper a dynamic technique to detect malware on Android platform is proposed. We exploit a set of energy related features i.e., feature which can be symptomatic of abnormal battery consumption. We built different models exploiting four different supervised machine learning classification algorithms, obtaining for all the evaluated models an accuracy greater than 0.91.

© 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of KES International.

Keywords: malware, Android, machine learning, classification, power management, security

1. Introduction and Related Work

Mobile devices widespread diffusion is attracting malicious attackers, which develop more and more aggressive code to steal private and sensitive information from our devices.

Mobile malware usually needs to perform system-intensive tasks to perform the harmful action[3]. This slows down the operating system, but it also has a secondary effect: all that processing power has to be fuelled by the battery. This results in a device keeping its charge much less than beforehand and requiring more charges across the week than before. Of course, there are multiple reasons for a fast-draining battery, such as an intensive app deliberately installed or simply battery getting old.

Dynamic methods for malware detection are based on features that can only be observed at runtime and that represent the behavior of applications (memory, CPU, network and statistics on system calls). Static approaches are less computationally intense than dynamic methods as they do not need applications to be run for identifying malware[2,5,12], but they are typically ineffective with obfuscated code as well as with run-time infections. On the other hand, dynamic methods are effective in identifying new threats, outperforming static methods, but they need

* Francesco Mercaldo, Antonella Santone
E-mail address: francesco.mercaldo@iit.cnr.it, antonella.santone@unimol.it

1877-0509 © 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of KES International.
10.1016/j.procs.2019.09.273

applications to be run to identify malicious behaviour, potentially infecting the device[6]. In addition dynamic methods are able to discriminate malware even when its code is obfuscated[14].

In current state of the art literature, researchers usually focused their attention on the detection of malware for mobile platform by analysing code[2,5] or device resources[6]. Few attention is dedicated to the energy consumption metrics, although it is a possible premonition of the presence of a malware.

2. The Method

The dynamic technique we propose relies on runtime observation of application under analysis behavior. We consider three categories: *CPU*, *Memory* and *Network*.

In detail in the *CPU* category is divided in two subcategories i.e., *CPU Usage* and *Virtual Memory*. In the first one falls the following metrics:

1. *user*: % of memory used by the application under analysis;
2. *kernel*: % of the memory requested by the Android kernel while the application under analysis is running.

In the second *CPU* subcategory (i.e., *Virtual Memory*) falls the number of minor and major page faults.

The *Memory* category there are following subcategories:

- *Native memory*: is the memory provided to the application process by the operating system. The memory is used for heap storage and other purposes;
- *Dalvik memory*: is the memory requested for the Dalvik virtual machine process;
- *Cursor memory*: is the memory requested for read-write access operation to the result set returned by database queries;
- *Android shared memory*: is the memory requested to share data between different processes.

In particular for each *Memory* subcategories following six features are gathered:

- *PSS*: is the total Proportional Set Size (PSS) is the RAM used by process. This is the sum of all PSS fields above it. It indicates the overall memory weight of your process, which can be directly compared with other processes and the total available RAM.
- *shared dirty*: is the amount of shared RAM that will be released back to the system when your process is destroyed. Dirty RAM is pages that have been modified and so must stay committed to RAM (because there is no swap in Android)
- *private dirty*: is the amount of RAM that will be released back to the system when your process is destroyed. Dirty RAM is pages that have been modified and so must stay committed to RAM (because there is no swap in Android)
- *heap size*: represents the RAM total usable by Dalvik Virtual Machine allocation for the application under analysis.
- *heap alloc*: represents the RAM actually allocated by Dalvik Virtual Machine allocation for the application under analysis.
- *heap free*: represents the allocable RAM (but not allocated yet) by Dalvik Virtual Machine for the application under analysis.

Table 1. The energy features involved in the study.

Category		Feature Names
CPU	CPU Usage	User, Kernel
	Virtual Memory	Page Minor Faults, Page Major Faults
Memory	Native memory	Native Pss, Native Shared Dirty, Native Private Dirty, Native Heap Size, Native Heap Alloc, Native Heap Free
	Dalvik memory	Dalvik Pss, Dalvik Shared Dirty, Dalvik Private Dirty, Dalvik Heap Size, Dalvik Heap Alloc, Dalvik Heap Free, Cursor Pss
	Cursor memory	Cursor Shared Dirty, Cursor Private Dirty
	Android shared memory	Ashmem Pss, Ashmem Shared Dirty, Ashmem Private Dirty
Network	Link layer	Number of ARP packets, Number of ICMP packets
	Internet layer	Number of IPv4 packets, Maximum packet size in bytes, Minimum packet size in bytes, Number of bytes, Number of packets, Number of IPv6 packets
	Transport layer	Number of TCP packets, Number of UDP packets

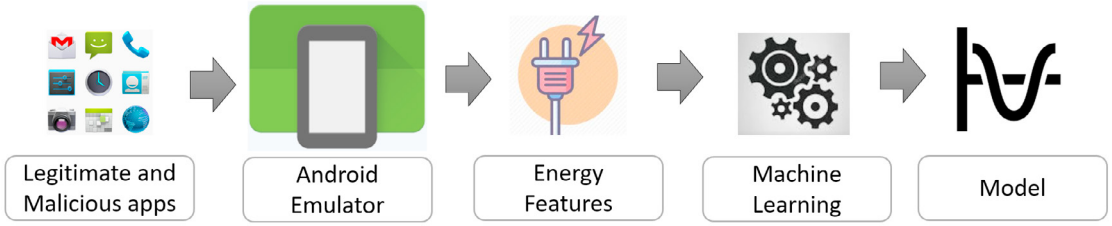

Fig. 1. Training.

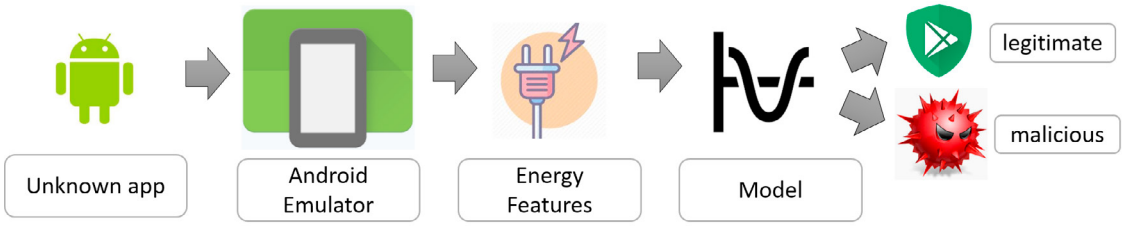

Fig. 2. Testing.

With regard to the third category, the *Network* one, three subcategories are considered: *link layer* (with following features: Number of ARP packets, Number of ICMP packets, Size in byte standard deviation), *internet layer* (with following features: Number of IPv4 packets, Maximum packet size in bytes, Minimum packet size in bytes, Number of bytes, Number of packets, Number of IPv6 packets) and *transport layer* (with following features: Number of TCP packets, Number of UDP packets).

In total, we extract 38 features from the execution traces, all referred to single applications resource usage. All the features considered are listed in Table 1.

Out of the considered features, 24 are related to different aspects of *Memory* usage. With regard to the *CPU* we extract 4 features and 10 are related to the *network* category.

Once described the proposed feature set, Figures 1 and 2 respectively show the *Training* and the *Testing* phases of the proposed method.

In detail in the *Training* phase we consider a set of labelled malicious and legitimate Android applications, these applications are ran on a virtual Android environment to extract the energy feature set.

The feature set related to the legitimate and malicious applications with the relative labels (i.e., *malicious* or *legitimate*) are the input for the supervised *machine learning* algorithms which input is a *model*.

The aim of the *Testing* phase (depicted in Figure 2) is to evaluate the effectiveness of the built model to discriminate between legitimate and malware Android applications.

Once gathered the energy feature from an unknown Android application, we test the feature set against the model built in the *Training* phase: the output is binary i.e., *legitimate* or *malicious*.

Table 2. The mobile malware families evaluated, the *dataset* column is related to the family repository: *D* for Drebin, while *H* for HelDroid.

Family	#samples	dataset
Adrd	84	D
Basebridge	48	D
DroidDream	81	D
DroidKungFu	100	D
FakeDoc	100	D
FakeInstaller	100	D
Geinimi	85	D
GinMaster	100	D
Kmin	100	D
Opfake	100	D
Plankton	100	D
Ransomware	100	H

3. The Evaluation

In this section we present the results we obtained.

We experiment the proposed method on a real-world dataset composed of legitimate and malicious applications belonging to different Android malware families.

We used a web crawler able to automatically download APK files from the Google official market[1]. The output of this step is an extended collection of Android applications. To obtain the malware dataset applications, we obtained samples from two malicious mobile application repositories: the Drebin[??] and the HelDroid[??] dataset. In detail, in the following study we consider a total of 2098 mobile applications, 1000 belonging to Google Play, while 1098 belonging to malware repositories (belonging to 12 different malware families).

Table 2 shows the malicious families considered.

Every family contains samples which have in common several characteristics, like payload installation, the kind of attack and events that trigger malicious payload For instance *FakeInstaller* samples have the main payload in common but have different code implementations, and some of them also have an extra payload. *FakeInstaller* malware is server-side polymorphic, which means the server could provide different apk files for the same URL request. There are variants of *FakeInstaller* that not only send SMS messages to premium rate numbers, but also include a backdoor to receive commands from a remote server. The *Opfake* samples make use of an algorithm that can change shape over time so to evade the antivirus. The *Opfake* malware demands payment for the app or content through premium text messages. This family represents the first polymorphic malware in Android environment. *DroidKungFu* installs a backdoor that allows hackers to access the phone when they want and use it as they please. They could even turn it into a bot. This malware encrypts two known root exploits, exploit and rage against the cage, to break out of the Android security container. When it runs, it decrypts these exploits and then contacts a remote server without the user knowing. *GinMaster* family contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and send to a remote website, as well as install applications without user interaction. The malware can successfully avoid detection by mobile anti-virus software by using polymorphic techniques to hide malicious code, obfuscating class names for each infected object, and randomizing package names and self-signed certificates for applications. *Geinimi* represents the first Android malware in the wild that displays botnet-like capabilities. Once the malware is installed, it has the potential to receive commands from a remote server that allows the owner of that server to control the phone. *Geinimi* makes use of a bytecode obfuscator, *Adrd* family is very close to *Geinimi* but with less server side commands, it also compromises personal data such as IMEI and IMSI of infected device. *DroidDream* is another example of botnet, it gained root access to device to access unique identification information. This malware could also downloads additional malicious programs without the user's knowledge as well as open the phone up to

[1] https://play.google.com/

control by hackers. The name derives from the fact that it was set up to run between the hours of 11 pm and 8 am when users were most likely to be sleeping and their phones less likely to be in use. *Plankton* uses an available native functionality (class loading) to forward details like IMEI, browser history to a remote server. Also *BaseBridge* sends information to a remote server running one ore more malicious services in background, like *Kmin* that transmits data like IMEI, IMSI and other files to premium-rate numbers. The *Ransomware* malicious payload exhibits the ability to deny the access to the mobile device[7], usually by locking the user interface or using a popup overlay and/or by encrypting the files stored on the external storage.

Once obtained the dataset, execution traces containing this information need to be collected by executing the applications in a controlled environment. These traces have been recorded by running the applications, one at a time, on the Android emulator. Monitoring scripts, with a monitoring interval of five seconds have been used. The procedure of executing the applications was automated by means of a Linux shell script, which has been run on a Linux Mint PC and made use of Android Debug Bridge (adb) [2], a command line tool that allows the PC to communicate with an emulator instance or with an Android device. Network log files were collected by capturing the network traffic of the emulator. Network statistics have been obtained by logging all network traffic of the emulator and by successively running the *tcpstat* tool, set to sample the features at 5s intervals. Log files for CPU, memory, and network are later unified by using timestamps recorded at execution time.

For applying stimuli to applications, the Monkey application exerciser [3] has been used in the script. It is a command-line tool that sends a pseudo-random stream of user events into the system, which acts as a stress test on the application software. One of the main problems of dynamic detection methods is in the execution of samples during the development phase. In fact, there is currently no method to verify automatically that malicious payloads are activated correctly. Due to the high number of samples that need to be used to obtain good quality classifiers, it is not even possible to perform this verification manually. This introduces some additional uncertainty in dynamic detection methods. In this work, we have tried to minimize the number of non-activated malicious payloads by providing a high number of stimuli (20,000, with a limit on execution time of 10 minutes) to each application (both malware and benign). It is our belief that the duration that we have chosen is a good tradeoff between time when most of the ransomware samples expose their malicious intentions and duration of the overall experimentation. The Android emulator of choice is the one included in the Android Software Development Kit [4] release 20140702, running Android 4.0, that was one of the most popular versions of Android. The reason why an Android emulator has been chosen instead of real devices is that this solution provides the ability to run a large number of applications, making the obtained dataset more significant. The Android operating system has been re-initialized each time before running each application, to avoid possible interferences (e.g., changed settings, running processes, and modifications of the operating system files) from previously run samples.

The classification analysis consists of building classifiers in order to evaluate the feature vector accuracy to distinguish between *legitimate* and *malware* Android applications.

For training the classifier, we defined T as a set of labeled application instances *(M, l)*, where each M is associated to a label $l \in \{legitimate,\ malicious\}$. For each M we built a feature vector $\mathrm{F} \in R_y$, where y is the number of the features used in training phase ($y = 38$). For the learning phase, we use a k-fold cross-validation: the dataset is randomly partitioned into k subsets. A single subset is retained as the validation dataset in order to evaluated the obtained model, while the remaining $k-1$ subsets of the original dataset are considered as training data. We repeated the process for $k = 10$ times; each one of the k subsets has been used once as the validation dataset. To obtain a single estimate, we computed the average of the k results from the folds.

We evaluated the effectiveness of the classification method with the following procedure:

1. build a training set $T \subset D$;
2. build a testing set $T' = D \div \mathrm{T}$;
3. run the training phase on T;

[2] http://developer.android.com/tools/help/adb.html
[3] http://developer.android.com/tools/help/monkey.html
[4] https://developer.android.com/sdk/index.htm

4. apply the learned classifier to each element of T'.

Each classification was performed using 30% of the dataset as training dataset and 70% as testing dataset employing the full feature set.

In order to enforce the conclusion validity we consider four different classification algorithms [11,10]:

- *HoeffdingTree* [9,8] (HT): is an incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams, assuming that the distribution generating examples does not change over time Hoeffding trees exploit the fact that a small sample can often be enough to choose an optimal splitting attribute.This idea is supported mathematically by the Hoeffding bound, which quantifies the number of observations (in our case, the instances) needed to estimate some statistics within a prescribed precision (in our case, the goodness of an attribute);

- *Support Vector Machine* [16]: the support-vector machine (SVN) constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks like outliers detection. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class (so-called functional margin), since in general the larger the margin, the lower the generalization error of the classifier[?] ;

- *Guassian* [13]: it involves a Gaussian process and it measures the similarity between points (the kernel function) to predict the value for an unseen point from training data. The prediction is not just an estimate for that point, but also has uncertainty informationit is a one-dimensional Gaussian distribution (which is the marginal distribution at that point[?] ;

- *K-nearest neighbours classifier* [1] (IBk): is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. In detail, in k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k=1, then the object is simply assigned to the class of that single nearest neighbor.

In the classification analysis, five metrics are considered: Precision, Recall, F-Measure and ROC Area.

The precision has been computed as the proportion of the examples that truly belong to class X among all those which were assigned to the class. It is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved:

$$Precision = \frac{tp}{tp+fp}$$

where *tp* indicates the number of true positives (for instance, a malware application rightly detected as *malicious*) and *fp* indicates the number of false positives (for instance, a trusted application wrongly detected as *legitimate*).

The recall has been computed as the proportion of examples that were assigned to class X, among all the examples that truly belong to the class, i.e., how much part of the class was captured. It is the ratio of the number of relevant records retrieved to the total number of relevant records:

$$Recall = \frac{tp}{tp+fn}$$

where *tp* indicates the number of true positives and *fn* indicates the number of false negatives (for instance, a malware application wrongly detected as *legitimate*).

The F-Measure is a measure of a test's accuracy. This score can be interpreted as a weighted average of the precision and recall:

$$F\text{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

The Roc Area is defined as the probability that a positive instance randomly chosen is classified above a negative randomly chosen.

Table 3 shows the classification results.

Algorithm	Precision	Recall	F-Measure	Accuracy
SVM	**0.913**	**0.933**	**0.929**	**0.937**
HT	**0.928**	**0.938**	**0.931**	**0.940**
Gaussian	**0.909**	**0.911**	**0.915**	**0.919**
IBk	**0.911**	**0.908**	**0.915**	**0.920**

Table 3. Performance results.

All the considered classification algorithms obtain a precision and a recall greater than 0.9. In detail the model obtaining the best performances (i.e., a precision equal to 0.928 and a recall equal to 0.938) is the one obtained with the HoeffdingTree algorithm. This is symptomatic that the proposed feature set is able to effectively discriminate between legitimate and malware Android application.

4. Conclusion and Future Work

In this paper a malware detector for Android platform is proposed. We consider 38 features belonging to three different categories: *CPU*, *Memory* and *Network*, subsequently the feature set is considered as input for four supervised machine learning algorithms. We reach a precision equal to 0.928 and a recall equal to 0.938 with the HoeffdingTree classification algorithm. Future works are related to the application of formal verification technique [15,4] to the energy related feature set considered with the aim to reach better performances.

Acknowledgments

This work has been partially supported by H2020 EU-funded projects SPARTA contract 830892 and C3ISP and EIT-Digital Project HII.

References

Altman, N.S., 1992. An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician 46, 175–185.

Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2018. Leila: formal tool for identifying mobile malicious behaviour. IEEE Transactions on Software Engineering .

Canfora, G., Mercaldo, F., Moriano, G., Visaggio, C.A., 2015. Composition-malware: building android malware at run time, in: 2015 10th International Conference on Availability, Reliability and Security, IEEE. pp. 318–326.

Ceccarelli, M., Cerulo, L., Santone, A., 2014. De novo reconstruction of gene regulatory networks from time series data, an approach based on formal methods. Methods 69, 298–305. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84927158722&doi=10.1016%2fj.ymeth.2014.06.005&partnerID=40&md5=d48bdbbe7485ece91340da58bb0bd32a, doi:10.1016/j.ymeth.2014.06.005. cited By 12.

Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., 2017a. Formal methods meet mobile code obfuscation identification of code reordering technique, in: 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), IEEE. pp. 263–268.

Cimitile, A., Mercaldo, F., Martinelli, F., Nardone, V., Santone, A., Vaglini, G., 2017b. Model checking for mobile android malware evolution, in: Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering, IEEE Press. pp. 24–30.

Ferrante, A., Malek, M., Martinelli, F., Mercaldo, F., Milosevic, J., 2017. Extinguishing ransomware-a hybrid approach to android ransomware detection, in: International Symposium on Foundations and Practice of Security, Springer. pp. 242–258.

Hoeglinger, S., Pears, R., 2007. Use of hoeffding trees in concept based data stream mining, in: 2007 Third International Conference on Information and Automation for Sustainability, IEEE. pp. 57–62.

Kourtellis, N., Morales, G.D.F., Bifet, A., Murdopo, A., 2016. Vht: Vertical hoeffding tree, in: 2016 IEEE International Conference on Big Data (Big Data), IEEE. pp. 915–922.

Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., 2017. Car hacking identification through fuzzy logic algorithms, in: Fuzzy Systems (FUZZ-IEEE), 2017 IEEE International Conference on, IEEE. pp. 1–7.

Martinelli, F., Mercaldo, F., Orlando, A., Nardone, V., Santone, A., Sangaiah, A.K., 2018. Human behavior characterization for driving style recognition in vehicle system. Computers & Electrical Engineering .

Mercaldo, F., Nardone, V., Santone, A., Visaggio, C., 2016. Hey malware, i can find you!, pp. 261–262. doi:10.1109/WETICE.2016.67.

Pérez, J.M., Muguerza, J., Arbelaitz, O., Gurrutxaga, I., Martín, J.I., 2007. Combining multiple class distribution modified subsamples in a single tree. Pattern Recognition Letters 28, 414–422.

Rastogi, V., Chen, Y., Jiang, X., 2013. Droidchameleon: evaluating android anti-malware against transformation attacks, in: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM. pp. 329–334.

Santone, A., Vaglini, G., 2012. Abstract reduction in directed model checking ccs processes. Acta Informatica 49, 313–341. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84864389117&doi=10.1007%2fs00236-012-0161-3&partnerID=40&md5=4f4cceffa15f2e8012617372a7621561, doi:10.1007/s00236-012-0161-3. cited By 15.

Webb, G., 1999. Decision tree grafting from the all-tests-but-one partition, Morgan Kaufmann, San Francisco, CA.