Acquiring and Analyzing App Metrics for Effective Mobile Malware Detection

Gerardo Canfora
Dept. of Engineering
University of Sannio
Italy
canfora@unisannio.it

Eric Medvet
Dept. of Eng. and Arch.
University of Trieste
Italy
emedvet@units.it

Francesco Mercaldo
Dept. of Engineering
University of Sannio
Italy
fmercaldo@unisannio.it

Corrado Aaron Visaggio
Dept. of Engineering
University of Sannio
Italy
visaggio@unisannio.it

ABSTRACT

Android malware is becoming very effective in evading detection techniques, and traditional malware detection techniques are demonstrating their weaknesses. Signature based detection shows at least two drawbacks: first, the detection is possible only after the malware has been identified, and the time needed to produce and distribute the signature provides attackers with window of opportunities for spreading the malware in the wild. For solving this problem, different approaches that try to characterize the malicious behavior through the invoked system and API calls emerged. Unfortunately, several evasion techniques have proven effective to evade detection based on system and API calls.

In this paper, we propose an approach for capturing the malicious behavior in terms of device resource consumption (using a thorough set of features), which is much more difficult to camouflage. We describe a procedure, and the corresponding practical setting, for extracting those features with the aim of maximizing their discriminative power. Finally, we describe the promising results we obtained experimenting on more than 2000 applications, on which our approach exhibited an accuracy greater than 99%.

Keywords

Malware, Android, Machine Learning

IWSPA'16,March 11 2016, New Orleans, LA, USA

http://dx.doi.org/10.1145/2875475.2875481

1. INTRODUCTION

Since the first SMS Trojan for Android was discovered by Kaspesky in August 2010, Android malware has been evolving, becoming more sophisticated especially in evasion techniques. As a matter of fact, recent malware families may be polymorphic, may encrypt malicious payload, adopt obfuscation techniques, make use of command and control communication channels, load malicious payload dynamically at runtime [32].

All those techniques make it difficult the malware detection for traditional signature based antimalware and call for new techniques for uncovering the malicious behaviour within Android applications. Moreover, malware may remain unnoticed for an average of three months [1, 21]: it has been observed that antivirus vendors took on average 48 days for a signature based antivirus engine to become capable of detecting new threats. Even Google Bouncer can offer a wide window of opportunities to malware writers as it has been showed to be vulnerable to detection avoidance [22].

There is hence a need for more effective detection approaches of Android malware, which could reduce the time elapsed between the diffusion and the discovering of new threats for Android platform. In order to overcome the evident limits of signature based detection, which are basically related to the collection and the consecutive distribution of the malware signature, many new approaches have been studied, mainly based on gathering features from the app and classifying them through a machine learning engine. These approaches aim at identifying the malicious behavior of the app, and the threat, in this case, is often characterized through a certain sequence of system or API calls, as in [31, 7]. One of the main limits of these approaches is that the malware writer can alter the original sequence of system or API calls leaving effective the app, but making ineffective the detection [19, 16].

In order to overcome this limitation, we propose to characterize the malicious behavior through the resources usage caused by the app under analysis. The underlying assumption is that the malicious app can exhibit, during its exe-

cution, some behaviors that consume the device resources (such as memory, CPU, network) in a peculiar way that is distinguishable from a legitimate behavior. As a matter of fact it should be very difficult to camouflage the resources consumption for a malware writer. Actually there are some works which explore this possibility, but they study the relation between malware and a specific resource type, like the use of transparent gestures [27], traffic packets [5], DNS traffic [23], and power consumption [12, 15, 20].

Our contributions include: (i) a set of numerical features related to resources usage which could allow a more accurate and complete analysis, (ii) a procedure to run applications in a controlled execution environment and collect features data, and (iii) a method for analyzing the collected data. We performed an experimental evaluation on more than 2.000 applications and obtained promising results (accuracy $> 99\%$).

The paper is organized as follows: Section 2 explores the related work; Section 3 explains the adopted methodology; Section 4 discusses the obtained results, and, finally, conclusions are drawn in Section 5.

2. RELATED WORK

Several authors extract resource-based metrics using dynamic analysis. Usually the focus is not malware detection, but monitoring resource consumption. In this section, we review literature about the usage of resource-based features using dynamic analysis in Android environment.

Wei et al. [29] propose an energy-based fair queuing as a pivotal instrument to improve the smartphone user experience (UX). Their approach represents a novel class of energy-aware scheduling algorithms that support proportional energy use, effective time constraint compliance, and a flexible trade-off between them. Their proposed algorithm is implemented in the Linux kernel V3.3 and verified on a test-bench based on a Linux scheduler simulator with user-specified energy loads. Simulation results show that their solution is more effective and flexible than the Linux scheduler in maximizing the user experience of energy-limited mobile systems.

Wineguide [11] application assesses network resources as signal strengths, monitored during usage of the applications. Higher signal strengths correspond to a better experience (e.g., speed). The aim of the method is to correlate UX to quality of service by taking into account both social and technical aspects.

Resources usage has been studied in literature also like a proxy for discriminating mobile malware from trusted applications. Andromaly [26] considers several features, like CPU usage, number of packets sent through the network, number of running processes, and battery level. The authors obtained a detection rate of 94%, but the apps used in the experimentation were developed ad-hoc by the authors.

Blasing and colleagues [4] use a combination of static and dynamic techniques. They first scan the code, then run the suspicious app in a totally monitored environment, catching all the events occurring in the sandbox such as files open operations and connections to remote server. They used a dataset composed by 150 applications in order to evaluate the proposed system. They retrieved the top 150 popular applications in October 2009 from the official Android Market. The applications were used to test the system showing positive results in the clustering. For testing malware, they used a self written fork bomb.

Shabtai et al. [25] use the knowledge-based temporal abstraction methodology. They detect suspicious temporal patterns to decide whether an intrusion is found, using memory, CPU, and power related features, obtaining a detection rate above 94%.

Researchers in [8] propose an Android malware detector evaluating permissions and system calls related to process management and to I/O operations. They obtain a precision equal to 74%, a recall equal to 80%, and a ROC area equal to 72%, using a dataset of 200 malicious applications.

TaintDroid [13] uses dynamic information flow tracking to detect sensitive data leakage. The dataset used included 30 popular third-party Android applications, and the method was able to recognize 20 apps with potential misuse of private information.

DroidScope [17] reconstructs semantic views to collect detailed execution traces of applications. This approach requires a customized Android kernel. The authors obtained a detection rate of 100%, but they analyzed only two Android malicious apps.

Authors in [30] propose an Android malware detection mechanism which is based on feature-network based, like incoming and outgoing information. They extract network spatial features and used independent component analysis from applications in order to determine the intrinsic Android malware domain name resolution communication behavior, obtaining a precision equal to 100% using 310 Android malware that have appeared since 2012.

Arora et al. [2] analyze the network traffic features building a rule-based classifier to detect Android malware. Their experimental results suggest that the approach is accurate and it detects more than 90% of the traffic samples. The work is focused only on Android malware which is remotely controlled or leaks information to some remote server, for a total of 27 different families.

Li et al. [18] propose a network traffic monitoring system used in the detection of Android malware. The system consists of four components: traffic monitoring, traffic anomaly recognition, response processing, and cloud storage. The system parses the protocol of data packets and extracts features, then uses SVM classification algorithm for data classification, establishing whether the network traffic is abnormal through correlation analysis. Their method is able to found 29 malware using a real world dataset composed by 60 applications.

Studies in [12, 15, 20] consider the power consumption as the distinguishable feature between benign and malicious applications. The method of [15] produced 99% true positive rate but using a sample of 3 malicious apps for validation; in [12, 20] no method performances are presented out of an experimental evaluation.

An approach for optimizing the energy efficiency is represented by Ecalc [14]: it analyzes a combination of bytecode execution traces and CPU profile of the application under development in order to estimate energy costs. Authors evaluated the approach by using a customized Android emulator for a total of three ad-hoc developed apps.

Another technique to monitor power CPU is proposed in [24]. The cited work aims at finding the optimum frequency for any application using Dynamic Voltage and Frequency Scaling (DVFS) and User Driven Frequency Scaling (UDFS). The proposed technique is implemented in a Samsung Galaxy S2 smartphone and it reduces 25% power consumption compared to default DVFS and 3% than existing UDFS technique.

Corral et al. [9] perform performance tests on seven versions of Android kernel monitoring battery consumption, I/O scheduling, and CPU frequency. They demonstrate that implementing kernel level customizations can lead to tangible improvements in the battery life and system performance of a smartphone.

As emerges from this discussion and at the best knowledge of the authors, the full set of features considered in this paper was never investigated with regards to Android malware detection.

3. METHODOLOGY

We propose a methodology based on three components:

1. a thorough set of numerical features related to resources usage which could allow a more accurate and complete analysis,
2. a procedure to run applications in a controlled execution environment and collect data, and
3. a method for analyzing the collected data.

The rationale for the first two components is to choose the features and the procedure to collect them so as to maximize the informativeness of those features with respect to the malware classification task (i.e., the third component). From another point of view, we strove to select, on one hand, a set of features which are "simple" enough to not require a complex collection procedure and, on the other hand, a collection procedure which can preserve the features informativeness despite their simplicity.

In the next sections, we present the features and the procedure, along with the experimental setting.

3.1 Features

We defined a set of features aimed at capturing the behavior of the application in terms of resources usage. To this end, we selected several features related to the following resources handled by Android operating system: (i) CPU; (ii) memory; (iii) storage; (iv) network. Moreover, we considered the resource consumption from two points of view: (a) consumption directly due to the application under analysis (AUA in the remaining of the paper), and (b) consumption due to the entire software environment (OS, AUA, and other applications). We call the two kinds of measures *AUA-grain* and *global-grain*, respectively.

In detail, the features concerning the CPU resource consumption are:

TotalUserCPU. The percentage of CPU time required for the user space by the operating system (global-grain).

TotalSystemCPU. The percentage of CPU time required for the kernel space by the operating system (global-grain).

PriorityCPU. The percentage of shared CPU assigned by the OS: it is useful to determine how much the CPU resource is used from the AUA (AUA-grain).

ProcessCPU. The total number of processes running for the AUA (AUA-grain).

ThreadCPU. The total number of threads running for the AUA (AUA-grain).

The features concerning the memory resource consumption are:

GLOBAL_FREE_MEM. The global amount of idle memory (global-grain).

GLOBAL_MAPPED_MEM. The global amount of used memory (global-grain).

GLOBAL_ANON_MEM. The global amount of anonymous mapping maps, i.e., the area of the virtual memory (all processes) not backed by any file (global-grain).

GLOBAL_SLAB_MEM. The global amount of slab—a slab is the amount by which a cache can grow or shrink. It represents one memory allocation to the cache from the device and whose size is customarily a multiple of the page size. A slab must contain a list of free buffers, as well as a list of buffers that have been allocated (global-grain).

VSS. The Virtual Set Size represents the total virtual memory size of the AUA—i.e., the total amount of memory in swap and RAM (AUA-grain).

PSS. The Proportional Set Size is the amount of AUA memory shared with other processes, accounted in a way that the amount is divided evenly between the processes that share it (AUA-grain).

RSS. The Resident Set Size is the portion of memory occupied by the AUA that is held in RAM (AUA-grain). The remaining part of the occupied memory exists in the swap space or file system, either because some parts of the occupied memory were paged out, or because some parts of the executable were never loaded (AUA-grain).

USS. The Unique Set Size is the set of pages that are unique for the AUA. This is the amount of memory that would be freed if the application is immediately terminated (AUA-grain).

The features concerning storage resource consumption are:

KBWRITTEN_IO. The amount of information (expressed in kB) written on the storage by all the processes (global-grain).

KBREAD_IO. The amount of information (expressed in kB) read from storage by all the processes (global-grain).

RCHAR_PROC_IO. The number of bytes which are read from storage by the AUA: the value is represented by the sum of bytes which the AUA process passed to `read()`[1] and `pread()`[2] system calls (AUA-grain).

WCHAR_PROC_IO. The number of bytes which are written into the storage by the AUA: the value is represented by the sum of bytes which the AUA process passed to `write()`[3] and `pwrite()`[4] system calls (AUA-grain).

SYSCR_PROC_IO. The value represents the number of read I/O operations, i.e., the occurrences of `read()` and `pread()` syscalls performed by the AUA (AUA-grain).

SYSCW_PROC_IO. The value represents the number of write I/O operations, i.e., syscalls like `write()` and `pwrite()`, performed by the AUA (AUA-grain).

Finally, the features concerning the network resource consumption are:

GLOBAL_RX_PACKETS. The number of packets received by all the processes (global-grain).

GLOBAL_TX_PACKETS. The number of packets transmitted by all the processes (global-grain).

GLOBAL_RX_BYTES. The number of bytes received by all the processes (global-grain).

GLOBAL_TX_BYTES. The number of bytes transmitted by all the processes (global-grain).

RX_PACKETS. The number of packets received by the AUA (AUA-grain).

TX_PACKETS. The number of packets transmitted by the AUA (AUA-grain).

RX_BYTES. The number of bytes received by the AUA (AUA-grain).

TX_BYTES. The number of bytes transmitted by the AUA (AUA-grain).

Table 1 shows an overview of the features, grouping them by the type of monitored resource and by the grain type (i.e., AUA-grain or global-grain).

[1]http://linux.die.net/man/2/read
[2]http://linux.die.net/man/2/pread
[3]http://linux.die.net/man/2/write
[4]http://linux.die.net/man/2/pwrite

3.2 Data collection procedure

We use different tools in order to monitor the resource consumption, each tool is related to the single monitored resource.

To retrieve information about the CPU consumption we use the `top` tool[5]. The `top` tool is able to provide an overview of the CPU activity in real time, i.e., while the applications are running. It displays a list of the most CPU-intensive tasks on the system, and it sorts the tasks by the CPU usage. With the `top` tool we extract both the global- and AUA-grain CPU features.

To extract the features concerning the global memory usage, we use the virtual memory statistics (`vmstat`) tool[6]. `vmstat` is a system monitoring tool that collects and displays information on operating system memory, processes, interrupts, and paging. Using `vmstat`, we are able to extract we extract both the global- and AUA-grain memory features.

Using `procrank`[7] tool we retrieve a quick summary of the process memory utilization. `procrank` is usually used to check if a process presents memory leakage. The binary is located in `/system/xbin` folder on Android devices. With `procrank` we extract following AUA-grain features: VSS, PSS, RSS, and USS.

The retrieval of information related to the I/O activity required to recompile the kernel since the tracking of these pieces of information is disabled by default. The information about the I/O activity is stored in the `proc/[PID]/io` folder but it is not accessible by a default Android version. To access such information, we modified the configuration file `goldfish_armv7_defconfig` which is located in the folder `/gfish/arch/arm/configs`. Obviously, the filename and the folder may change depending on the kernel and on the architecture of the device.

In the configuration file we enabled the following options (which are, by default, disabled—i.e., set to `n`):

```
CONFIG_TASKSTATS=y;
CONFIG_TASK_IO_ACCOUNTING=y;
CONFIG_TASK_XACCT=y;
CONFIG_TASK_DELAY_ACCT=y
```

In this work, we used the Android emulator and the Goldfish kernel[8] cloned from following git clone: https://android.googlesource.com/kernel/goldfish.git. The reason why we used the Goldfish kernel is because the Android emulator runs a virtual CPU that Google calls Goldfish. Goldfish executes ARM926T instructions and has hooks for input and output, such as reading key presses from or displaying video output only in the emulator. These interfaces are implemented in files specific to the Goldfish emulator and will not be compiled into a kernel that runs on real devices. The procedure for building an appropriate version of Goldfish involves several steps. After the configuration file editing, the command `make clean && make mrproper`

[5]http://linux.die.net/man/1/top
[6]https://www.freebsd.org/cgi/man.cgi?query=vmstat
[7]http://elinux.org/Android_Memory_Usage#procrank
[8]https://source.android.com/source/building-kernels.html

Resource	Feature	Global	AUA	Tool
CPU	TotalUserCPU	✓		`top`
	TotalSystemCPU	✓		`top`
	PriorityCPU		✓	`top`
	ProcessCPU		✓	`top`
	ThreadCPU		✓	`top`
Memory	GLOBAL_FREE_MEM	✓		`vmstat`
	GLOBAL_MAPPED_MEM	✓		`vmstat`
	GLOBAL_ANOM_MEM	✓		`vmstat`
	GLOBAL_SLAB_MEM	✓		`vmstat`
	VSS		✓	`vmstat`
	PSS		✓	`vmstat`
	RSS		✓	`vmstat`
	USS		✓	`vmstat`
Storage	KBWRITTEN_IO	✓		`iostat`
	KBREAD_IO	✓		`iostat`
	RCHAR_PROC_IO		✓	`proc/[PID]/io` folder
	WCHAR_PROC_IO		✓	`proc/[PID]/io` folder
	SYSCR_PROC_IO		✓	`proc/[PID]/io` folder
	SYSCW_PROC_IO		✓	`proc/[PID]/io` folder
Network	GLOBAL_RX_PACKETS	✓		DDMLIB libraries
	GLOBAL_TX_PACKETS	✓		DDMLIB libraries
	RX_PACKETS		✓	DDMLIB libraries
	TX_PACKETS		✓	DDMLIB libraries
	GLOBAL_RX_BYTES	✓		DDMLIB libraries
	GLOBAL_TX_BYTES	✓		DDMLIB libraries
	RX_BYTES		✓	DDMLIB libraries
	TX_BYTES		✓	DDMLIB libraries

Table 1: Overview of the our features, each feature is associated with a monitored resource (CPU, storage, memory, and network); in addition, each feature is associated with the "grain" level (AUA-grain or global-grain) and to the tool used to extract it.

cleans up any artifacts of previous compilations; the command `make goldfish_armv7_defconfig` is invoked to indicate the modified configuration file at compile time; with the command `export CROSS_COMPILE=/path/to/toolchains/arm-eabi-4.4.0/prebuilt/linux-x86/bin/arm-eabi-` we indicate the compiler to be used; and, finally, the command `make -j3 ARCH=arm` starts the compilation process by selecting an ARM type architecture. The result of the compilation is the `zImage` image kernel which is stored in the folder `/gfish/arch/arm/boot`. The `zImage` image file is specified, when the Android emulator starts, by using the option: `-kernel /path/to/zImage`.

Once the kernel is recompiled and run, we are able to extract information about the I/O storage activity. At this point, to retrieve global information about I/O activity we use the `iostat`[9] command, a useful utility for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. With `iostat` we extract the following global-grain I/O features, KBWRITTEN_IO and KBREAD_IO. To retrieve information about the storage usage related to the AUA we parse the `proc/[PID]/io` file, useful to extract the remaining staorage AUA-grain features.

To extract the features related to the use of the network, we developed a tool that extends the Device Monitor[10] tool which can be found in the Android SDK. Device Monitor is an open source tool that performs network analysis; however, it can not be invoked via command line, but only through its graphical interface, which does not fit our scenario as we need a command-line tool to invoke using the script we developed. We hence developed a tool able to extend the classes of Device Monitor using the DDMLIB libraries built-in in Device Monitor tools. With this tool, we extract both the global- and AUA-grain network features.

We developed a script able to invoke the different tools while the application is running, in order to automatize the features extraction process. The AUA is run for 60 s, then the script retrieves the features every 6 s for 10 times, i.e., we store 10 different measures for each feature. During the AUA execution, we used the `monkey` tool of the Android Debug Brigde (ADB[11]) in order to stimulate the AUA with GUI interactions. `monkey` generates pseudo-random streams of user events such as clicks, touches, or gestures; moreover, it can simulate a number of system-level events. Specifically, we configured `monkey` to send 4000 random UI events during the execution of the AUA. Summarizing, the script performs the following procedure:

[9]http://linuxcommand.org/man_pages/iostat1.html

[10]http://developer.android.com/tools/help/monitor.html

[11]http://developer.android.com/tools/help/adb.html (version 1.0.32)

1. copies the AUA into the storage of emulated device;
2. installs the AUA (using the `install` command of ADB);
3. gets the package name and the class (activity/service) of the AUA with the launcher intent (i.e., get the AUA entry point, needed for step 4);
4. starts the AUA (using the `am start` command of ADB);
5. gets the AUA process id (PID);
6. starts `monkey` (using the `monkey` command of ADB), instructed to send UI and system events;
7. waits 60 s;
8. collects the features;
9. waits 6 seconds;
10. repeats ten times from step 8;
11. kills the AUA (using the PID collected at step 5;
12. uninstalls the AUA (using the `uninstall` command of ADB);
13. saves the extracted features;
14. deletes the AUA from the device.

3.3 Data analysis

We analyze the collected data by means of machine learning techniques trained on a labeled dataset, as follows.

Let be $\mathcal{L}$ a *learning set* of pairs $(\vec{f}, l)$, where $\vec{f}$ is the numeric vector of the data collected for an application according to the procedure described before and l is a binary label indicating if the application is or is not malicious. First, during the training phase, we tune a Random Forest [6] classifier on $\mathcal{L}$—we set the number of trees $n_{\text{tree}} = 500$. We remark that the tuning is executed just once. Then, when analyzing the data collected for the AUA, we classify the corresponding numeric vector $\vec{f}$ with the trained classifier.

We considered several variants of this analysis technique, along two axes: (i) the preprocessing of the collected data, and (ii) the subset (if any) of the features being analyzed (i.e., global-grain, AUA-grain, or both). Obviously, for all variants, we applied the modification on the training data too.

Concerning the former, we explored two options: using the data as is, or using the Discrete Cosine Transform (DCT) of the data. More in detail, in the latter case, the vector $\vec{f}$ is composed, for each feature, of the DCT of its 10 collected readings. The rationale for using the DCT is to capture the frequency distribution of the signals, i.e., in our case, of resource consumption: from another point of view, using the DCT corresponds to assume that the discriminative power of the features is more in their variation over the time than on their actual values. Indeed, DCT has already been used on application execution traces for anomaly detection [10]. We denote the 6 resulting variants by Raw-All, Raw-Global, Raw-AUA, DCT-All, DCT-Global, and DCT-AUA. Variants with prefix Raw- use data as is, whereas those with prefix DCT- use the transformed readings; suffixes -All, -Global, and -AUA indicate which subset of the features have been used: all features, only global-grain, or only AUA-grain.

4. RESULTS AND DISCUSSION

We assessed our proposed methodology on a dataset of 2.133 real Android applications, 1.059 trusted and 1.074 malware samples.

The trusted applications were crawled from Google Play[12], by using a script which queries a python API[13] to search and download apps. The downloaded applications belong to all the 26 different available categories (i.e., Books & Reference, Lifestyle, Business, Live Wallpaper, Comics, Media & Video, Communication, Medical, Education, Music & Audio, Finance & News, Magazines, Games, Personalization, Health & Fitness, Photography, Libraries & Demo, Productivity, Shopping, Social, Sport, Tools, Travel, Local & Transportation, Weather, Widgets). The applications retrieved were among the most downloaded in their category and were free. The trusted applications were collected between January 2015 and April 2015 and were later analysed with the VirusTotal service[14]. This service runs 57 different antimalware software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on the app: the analysis confirmed that the crawled applications did not contain malicious payload.

The malware dataset was obtained from the Drebin dataset. This dataset consists of a total of 5.560 applications belonging to 179 different families, classified as malware [3, 28]. To the best of our knowledge, this is the most recent dataset of mobile malware applications currently used to evaluate malware detectors in literature. In order to improve the validity of the experiment by analyzing a (roughly) balanced sample of real applications, we randomly selected 1.074 malware applications.

We performed a ten-fold cross validation of our methodology, as follows. First, we partitioned our dataset in 10 subsets, each one with the same proportion of malware and trusted applications of the overall dataset. Then, for each partition, we built the learning set $\mathcal{L}$ by considering all but that partition and applied our methodology to each application of the partition, that is, we assessed our method on data that was not used for training. We measured the effectiveness in terms of False Positive Rate (FPR), i.e., the percentage of trusted applications which have been classified as malware, False Negative Rate (FNR), i.e., the percentage of malware applications which have been classified as trusted, and accuracy, i.e., the percentage of applications which have been correctly classified.

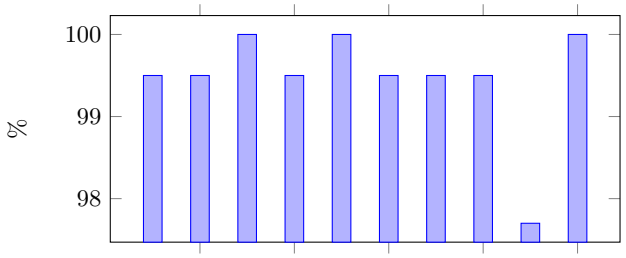
Table 2 shows the results, averaged across the 10 repetitions. Besides the average for accuracy, FPR, and FNR, the table shows the standard deviation for each metric.

The main finding of the experimental evaluation is that our proposed method is very effective: the Raw-Global vari-

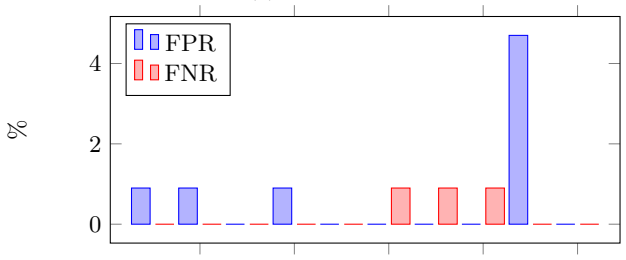
[12]https://play.google.com
[13]https://github.com/egirault/googleplay-api
[14]https://www.virustotal.com/

Variant	Accuracy		FPR		FNR	
	avg	sd	avg	sd	avg	sd
Raw-All	98.97	0.57	1.70	1.37	0.37	0.65
Raw-Global	99.52	0.69	0.74	1.45	0.18	0.38
Raw-AUA	98.92	0.63	1.69	1.16	0.46	0.66
DCT-All	95.82	1.29	5.69	2.42	2.70	1.66
DCT-Global	96.72	1.08	4.45	2.31	2.14	1.08
DCT-AUA	94.85	1.70	5.97	2.08	4.37	2.40

Table 2: Results (average and standar deviation) across the 10 repetitions.



(a) Accuracy.



(b) FPR and FNR.

Figure 1: Details of the results for Raw-Global variant on all the 10 repetitions.

ant obtains an accuracy of 99.52%. We note that this is one of the best results obtained in Android malware detection among those studies which performed an analysis over a large (more than 1.000, in our case) number of real world apps. It can also be seen that the value for obtained accuracy is quite robust with respect to the available learning data set (see the standard deviation in Table 2). This finding is further confirmed by Figure 1, which shows, for the best variant Raw-Global, accuracy (Figure 1a), FPR, and FNR (Figure 1b) for each one of the 10 repetitions.

Table 2 also shows that all the variants which analyze the feature as they are collected (Raw-All, Raw-Global, and Raw-AUA) obtain a good accuracy ($\geq$ 98.9%, in all cases). On the other hand, variants which perform the DCT of collected values perform slightly worse (from 94% to 97%). We interpret this result as the fact that our proposed data collection procedure is indeed able to capture feature values which have good discriminative power and no pre-processing is needed to filter out "noise" from the data.

Finally, it appears, from our experimental evaluation, that global-grain features are better than AUA-grain features: both Raw- and DCT- variants obtain better results with the formers alone than with the latters or with all the features. We think that the motivation of this result is in the way data is collected in our procedure. Running the AUA on a "clean" (emulated) device and starting the actual data collection after 60 s likely allows to observe the AUA right when its behavior mostly affects the Android operating system metrics: as a results, global features deliver a great discriminative power.

5. CONCLUDING REMARKS

Polymorphism, obfuscation, and dynamic loading of the payload are making malware detection very hard on Android platforms. Signature based commercial solutions are by now ineffective, as they offer attackers a huge window of opportunities consisting in the time necessary to capture and distribute the malware signature. For contrasting this problem, many approaches based on identifying malware behaviour through the invoked system calls and API calls have been proposed. The main limitation of this kind of approaches is that evasion techniques emerged for altering system calls sequences without modifying the actual behavior.

We proposed a solution which builds on a thorough set of features describing the device resource consumption for discriminating malware from goodware. This method demonstrated to be effective, as we obtained an accuracy greater than 99%, and it is much harder to camouflage for a malware writer. The experimentation provided us with the most appropriate set of features that can be used to identify malicious behaviors. As future work we aim at relating a specific pattern of resourse consumption to a peculiar behavior of the malicious app, which may concern an infection mechanisms, a communication action with a command and control server, or the execution of a specific payload.

6. REFERENCES

[1] A. Apvrille and T. Strazzere. Reducing the window of opportunity for android malware gotta catch'em all. *Journal in Computer Virology*, 8(1-2):61–71, 2012.

[2] A. Arora, S. Garg, and S. K. Peddoju. Malware detection using network traffic analysis in android based mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 66–71. IEEE, 2014.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proc. of 17th Network and Distributed System Security Symposium, NDSS*, volume 14.

[4] 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*, pages 55–62. IEEE, 2010.

[5] A. Boukhtouta, N.-E. Lakhdari, and M. Debbabi. Inferring malware family through application protocol sequences signature. In *New Technologies, Mobility and Security (NTMS), 2014 6th International Conference on*, pages 1–5. IEEE, 2014.

[6] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] 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*, pages 13–20. ACM, 2015.

[8] G. Canfora, F. Mercaldo, and C. A. Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 607–614. IEEE, 2013.

[9] L. Corral, A. B. Georgiev, A. Janes, and S. Kofler. Energy-aware performance evaluation of android custom kernels. In *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, pages 1–7. IEEE Press, 2015.

[10] A. Cuzzocrea, E. Medvet, E. Mumolo, and R. Cecolin. Detecting anomalies in embedded computing systems via a novel hmm-based machine learning approach. In *Hybrid Artificial Intelligent Systems*, pages 405–415. Springer, 2015.

[11] T. Deryckere, W. Joseph, L. Martens, L. De Marez, K. De Moor, and K. Berte. A software tool to relate technical performance to user experience in a mobile context. In *World of Wireless, Mobile and Multimedia Networks, 2008. WoWMoM 2008. 2008 International Symposium on a*, pages 1–6. IEEE, 2008.

[12] B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra. Location based power analysis to detect malicious code in smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 27–32. ACM, 2011.

[13] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[14] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 1–7. IEEE Press, 2012.

[15] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2008.

[16] J. Kwon, J. Jeong, J. Lee, and H. Lee. Droidgraph: discovering android malware by analyzing semantic behavior. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 498–499. IEEE, 2014.

[17] L. Kwong and Y. H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. 2012.

[18] J. Li, L. Zhai, X. Zhang, and D. Quan. Research of android malware detection based on network traffic monitoring. In *Industrial Electronics and Applications (ICIEA), 2014 IEEE 9th Conference on*, pages 1739–1744. IEEE, 2014.

[19] Q. Li and X. Li. Android malware detection based on static analysis of characteristic tree. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2015 International Conference on*, pages 84–91. IEEE, 2015.

[20] L. Liu, G. Yan, X. Zhang, and S. Chen. Virusmeter: Preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, pages 244–264. Springer, 2009.

[21] J. Oberheide, E. Cooke, and F. Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[22] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[23] B. Rahbarinia, R. Perdisci, and M. Antonakakis. Segugio: Efficient behavior-based tracking of malware-control domains in large isp networks. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 403–414. IEEE, 2015.

[24] M. A. Rumi, D. Hasan, et al. Cpu power consumption reduction in android smartphone. In *Green Energy and Technology (ICGET), 2015 3rd International Conference on*, pages 1–6. IEEE, 2015.

[25] A. Shabtai, U. Kanonov, and Y. Elovici. Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. *Journal of Systems and Software*, 83(8):1524–1537, 2010.

[26] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[27] B. Shrestha, M. Mohamed, A. Borg, N. Saxena, and S. Tamrakar. Curbing mobile malware based on user-transparent hand movements.

[28] M. Spreitzenbarth, F. Echtler, T. Schrek, F. C. Freiling, and J. Hoffman. Mobilesandbox: looking deeper into android applications. In *Proc. the 28th ACM Symposium on Applied Computing (SAC)*, 2013.

[29] J. Wei, E. Juarez, M. J. Garrido, and F. Pescador. Maximizing the user experience with energy-based fair sharing in battery limited mobile systems. *Consumer Electronics, IEEE Transactions on*, 59(3):690–698, 2013.

[30] T.-E. Wei, C.-H. Mao, A. B. Jeng, H.-M. Lee, H.-T. Wang, and D.-J. Wu. Android malware detection via a latent network behavior analysis. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1251–1258. IEEE, 2012.

[31] S. Y. Yerima, S. Sezer, and I. Muttik. High accuracy android malware detection using ensemble learning. *IET Information Security*, 2015.

[32] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.