Detecting Android Malware using Sequences of System Calls

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

Eric Medvet
Dept. of Engineering and Architecture
University of Trieste
Trieste, Italy
emedvet@units.it

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

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

ABSTRACT

The increasing diffusion of smart devices, along with the dynamism of the mobile applications ecosystem, are boosting the production of malware for the Android platform. So far, many different methods have been developed for detecting Android malware, based on either static or dynamic analysis. The main limitations of existing methods include: low accuracy, proneness to evasion techniques, and weak validation, often limited to emulators or modified kernels.

We propose an Android malware detection method, based on sequences of system calls, that overcomes these limitations. The assumption is that malicious behaviors (e.g., sending high premium rate SMS, cyphering data for ransom, botnet capabilities, and so on) are implemented by specific system calls sequences: yet, no apriori knowledge is available about which sequences are associated with which malicious behaviors, in particular in the mobile applications ecosystem where new malware and non-malware applications continuously arise. Hence, we use Machine Learning to automatically learn these associations (a sort of "fingerprint" of the malware); then we exploit them to actually detect malware. Experimentation on 20 000 execution traces of 2000 applications (1000 of them being malware belonging to different malware families), performed on a real device, shows promising results: we obtain a detection accuracy of 97%. Moreover, we show that the proposed method can cope with the dynamism of the mobile apps ecosystem, since it can detect unknown malware.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification -Validation; D.4.6 [**Operating Systems**]: Security and Protection—Invasive software

General Terms

Security

Keywords

malware, Android, dynamic analysis, security, machine learning

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...
http://dx.doi.org/10.1145/2804345.2804349

1. INTRODUCTION

A number of surveys by specialised companies and commercial press articles provide evidence that the mobile malware on the Android platform is growing in volume and impact [6, 8, 7]. There are two general approaches to implement malware detectors based on (a) static analysis or (b) dynamic analysis. Broadly speaking, the former does not require many resources, in terms of enabling infrastructure, and is faster to execute than the latter, but it is more prone to be evaded with techniques whose effectiveness has been largely demonstrated in the literature [24, 30, 35]. The latter is harder to bypass, as it captures the behavior, but it usually needs more resources and cannot be run directly on the devices (it is often performed on virtual or dedicated machines).

We propose a malware detection technique based on dynamic analysis which considers sequences of system calls that are likely to occur more in malware than in non-malware applications. The rationale behind our choice can be explained as follows. Often, the process of malware evolution mainly consists of modifications to existing malware. Malware writers use to improve mechanisms of infection, obfuscation techniques, or payloads already implemented in previous malware, or tend to combine them [36]. This indeed explains why malware is classified in terms of families, i.e., malicious apps which share behaviors, implementation strategies and characteristics [10, 37]. As a consequence, malicious apps belonging to the same family are very likely to exhibit strong similarities in terms of code and behavior.

Our assumption is that the behavior similarities among malicious apps could be used to detect new malware. We chose to characterize behavior in terms of sequences of system calls as this representation is, on the one hand, specific enough to capture the app behavior and, on the other hand, it is generic enough to be robust to camouflage techniques aimed at hiding the behavior. In other words, we assume that the frequencies of a set of system calls sequences may represent a sort of *fingerprint* of the malicious behavior. Thus, new malware should be recognized when that fingerprint is found.

The contributions of this paper can be summarized as follows:

1. We designed a method for (i) automatically selecting, among the very large number of possible system calls sequences, those which are the most useful for malware detection, and, (ii) given the fingerprint defined in terms of frequencies of the selected system calls sequences, classifying an execution trace as malware or non-malware.

2. We performed an extensive experimental evaluation using a real device on which we executed 2000 apps for a total of 20 000 runs: we found that our method delivers an accuracy up to 97%. We remark that, in most cases, previous works based on dynamic analysis validated their proposals using emulators and modified kernels, which produce outcomes which are less realistic than the outcomes deriving from real devices, i.e., the one used in this experimentation.

3. We assessed our method also in the more challenging scenario of zero-day attacks, where the detection is applied to *new* malware applications or *new* malware families, i.e., to applications whose behavior has never observed before (and hence has not be exploited to build the fingerprint).

The reminder of the paper is organized as follows: Section 2 thoroughly analyses related work, Section 3 introduces the approach, and Section 4 illustrates the validation of the approach. Finally, Section 5 draws the conclusions.

2. RELATED WORK

Malware detection techniques can be characterized in terms of the features they use to discriminate between malware and non-malware applications: those features can be obtained by means of static analysis or dynamic analysis. In general, static analysis captures suspicious patterns within the code (or artifacts related to code, such as meta information), whereas dynamic analysis captures suspicious patterns related to the behavior observed during the application running [16, 28]. Here we review recent works based mainly on dynamic analysis, and specifically on the analysis of system calls [14, 19, 26, 32, 18, 27, 21, 31, 20, 13].

Canfora et al. [14] propose a method for detecting mobile malware based on three metrics, which evaluate: the occurrences of a reduced subset of system calls, a weighted sum of a subset of permissions which the application requires, and a set of combinations of permissions. The experimentation of this paper considers a sample of 200 real world malicious apps and 200 real world trusted apps and produced a precision of 74%. CopperDroid [26] recognizes malware through a system calls analysis using a customized version of the Android emulator able to tracking system calls. Wang et al. [33] use an emulator to perform a similar task. The method was validated on a set of 1600 malicious apps, and was able to find the 60% of the malicious apps belonging to one sample (the Genoma Project), and the 73% of the malicious apps included in the second sample (the Contagio dataset).

Jeong et al. [19] hook system calls to create, read/write operations on files, and intents activity 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 sample included 2 malicious apps developed by the authors. In [32] the authors characterize the response of an application in terms of a subset of system calls generated from bad activities in background when they stimulate apps with user interfaces events. They use an Android emulator for running experiments and their evaluation is based on 2 malicious samples of DroidDream family (Super History Eraser and Task Killer Pro). Schmidt et al. [27] used the view from Linux-kernel such as network traffic, system calls, and file system logs to detect anomalies in the Android system. The method presented in reference [18] consists of an application log and a recorder of a set of system calls related to management of file, I/O and processes. They use a physical device, with an Android 2.1 based modified ROM image. The evaluation phase considers 230 applications in greater part downloaded from Google Play and the method detects 37 applications which steal some kinds of personal sensitive data, 14 applications which execute exploit code and 13 destructive applications.

Concerning system call usage in systems other than Android, authors in [21] developed a system calls collection module that was installed on ten machines running Microsoft Windows XP, used by people carrying out their normal activities. They observed 242 trusted different applications, collecting 1.5 billion system calls over a period of several weeks. Kolbitsch et al. [20] track dependencies among system calls to match the activity of an unknown program against the trained behavior models from six different malware families. They evaluate the technique with 300 Microsoft Windows worm with a detection effectiveness of 64% (varying from 10% regarding the Agent family from 90% with the Allaple family). In [13] the authors adopt system calls to detect malicious JavaScript. They evaluate two techniques for detecting malicious web pages based on system calls: the first one consists of counting the occurrences of specific system calls, while the second one consists of retrieving system calls sequences. The first technique produces an accuracy slightly higher that the second one (97% vs. 96%).

In summary, the main differences between our work and other dynamic analysis techniques for malware detection are:

- Our approach takes into account all the system calls rather than a reduced set and we consider sequences of system calls, rather than system calls taken in isolation.

- We validate our approach on a large set of 2000 applications. The only papers which consider a very large dataset [17, 38] do not propose neither assess a detection method, but evaluate properties of the infection rate within a specific marketplace.
- We obtain a high detection accuracy (97%). The papers proposing a method with better performances than ours have a data set whose size is much smaller than our data set and, in some cases, malware samples are written by the authors rather than taken from the real marketplaces.
- We run the experimentation on a real device, while quite all the papers make use of emulators, which reduces the truthfulness of the experiments.

3. DETECTION METHOD

We consider the problem of classifying an execution trace of a mobile application as trusted or malicious, i.e., classifying the corresponding application as non-malware or malware. An *execution trace* is a sequence t in which each element represents a system call being issued by the application under analysis (AUA): we consider only the function name and discard all the parameters values. We denote by C the set of all the possible system calls, i.e., the alphabet to which symbols forming a sequence t belong.

A system call is the mechanism used by a user-level process or application layer to request a kernel level service to the operating system, such as power management, memory, process life-cycle, network connection, device security, access to hardware resources [29]. When performed, a system call implies a shift from user mode to kernel mode: this allows the kernel of the operating system to perform sensitive operations. When the task carried out by the invoked system call is completed, the control is returned to the user mode. In [9] Android kernel invocations are sub-grouped into: (i) system calls which directly invoke the native kernel functionality; (ii) binder calls, which support the invocation of Binder driver in the kernel (Binder is a system for inter-process communication); and (iii) socket calls, which allow read/write commands and send data to/from a Linux socket. System calls are not called directly by a user process: they are invoked through interrupts, or by means of asynchronous signals indicating requests for a particular service sent by the running process to the operating system. A Linux kernel (which the Android system builds on) has more than 250 system calls: our method considers all the system calls generated by the AUA when running.

The classification method here proposed consists of two phases, the training phase, in which a classifier is built on a set of labeled examples, and the actual classification phase, in which a trace is classified by the classifier.

In the training phase, we proceed as follows. Let T be a set of *labeled examples*, i.e., a set of pairs (t, l) where t is a trace and $l \in \{\text{trusted}, \text{malicious}\}$ is a label. We first transform each trace t in a feature vector $\mathbf{f}(t) \in [0,1]^{|C|^n}$, where n is a parameter. Each element of $\mathbf{f}(t)$ represents the relative occurrences of a given n-long subsequence of system calls in t. For example, let $n = 3$ and let $f_{2711}(t)$ represents the relative occurrences of the subsequence $\{\texttt{open}, \texttt{stat64}, \texttt{open}\}$ in t (i.e., in this example, the index of the element of $\mathbf{f}$ which corresponds to the subsequence $\{\texttt{open}, \texttt{stat64}, \texttt{open}\}$ is 2711), then $f_{2711}(t)$ is obtained as the number of occurrences of that subsequence divided by the sequence length $|t|$.

The number $|C|^n$ of different features may be remarkably large: for this reason, we perform a feature selection procedure aimed at selecting only the k feature which best discriminate between trusted and malicious applications in T. The feature selection procedure comprises of two steps. Let T_t and T_m be respectively the set of trusted and malicious traces, i.e., $T_t := \{(t, l) \in T : l = \text{trusted}\}$ and $T_m := \{(t, l) \in T : l = \text{malicious}\}$.

In the first step, we compute, for each ith feature, the relative class difference δ_i as:

$$\delta_i = \frac{\left| \frac{1}{|T_t|} \sum_{(t,l) \in T_t} f_i(t) - \frac{1}{|T_m|} \sum_{(t,l) \in T_m} f_i(t) \right|}{\max_{(t,l) \in T} f_i(t)}$$

We then select the $k' \gg k$ features with the greatest δ_i among those for which $\max_{(t,l) \in T} f_i(t) > 0$—$k$ and k' are parameters of the method.

In the second step, we compute, for each ith feature among the k' selected at the previous step, the mutual information I_i with the label. We then select the k features with the greatest I_i. During the exploratory analysis, we also experimented with a feature selection procedure which took into account, during the second step, the inter-dependencies among features: we found that it did not deliver better results.

Finally, we train a Support Vector Machine (SVM) on the selected features for the traces contained in T: we use a Gaussian radial kernel with the cost parameter set to 1.

The actual classification of an execution trace t is performed by computing the corresponding feature vector $\mathbf{f}(t)$, retaining only the features obtained by the aforementioned feature selection procedure, and then applying the trained classifier.

4. EXPERIMENTAL EVALUATION

We performed an extensive experimental evaluation for the purpose of assessing our method in the following detection scenarios:

- *Unseen execution trace* Ability to correctly classify an execution trace of an application for which other execution traces where available during the training phase.
- *Unseen application* Ability to correctly classify an execution trace of an application for which no execution traces were available during the training phase, but belonging to a malware family for which some traces were available during the training phase.
- *Unseen family* Ability to correctly classify an execution trace of a malware application belonging to a family for which no execution traces were available during the training phase.

Clearly, the considered scenarios are differently challenging (the former is the least challenging, the latter is the most challenging), since the amount of information available to the training phase, w.r.t. the AUA, is different. We investigated these scenarios using a single dataset $\mathcal{T}$, whose collection procedure is described in the next section, and varying the experimental procedure, i.e., varying the way we built the training set T and the testing set T' from $\mathcal{T}$.

4.1 Baseline

In order to provide a baseline, we designed and evaluated a detection method based on application permissions. As discussed in sections 2, permissions have been shown to be a relevant feature for the purpose of discriminating between malware and non-malware applications [10, 23, 22, 11, 14, 34, 15, 12].

In particular, in the baseline method we build a feature vector $\mathbf{f}$ for each application where an element f_i is 1 or 0, depending on the respective ith permission being declared for that application. The list of all the permissions (and hence the length of the feature vectors) is determined once in the training phase. The remaining part of the baseline method is the same as in the proposed method: two steps feature selection, SVM training and actual classification using the trained SVM on the selected features. Note that the only parameter which matters in the baseline method is the number k of selected features.

4.2 Data collection

4.2.1 Applications

We built a dataset of traces collected from 2000 Android applications, 1000 trusted and 1000 malware.

The trusted applications were automatically collected from Google Play [1], by using a script which queries an unofficial python API [3] to search and download apps from Android official market. 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 were among the most downloaded in their category and they are free. We chose the most popular apps in order to increase the probability that these apps were actually trusted. The trusted applications were collected between January 2014 and April 2014 and they were later analysed with the VirusTotal service [5]. This service run 52 different antivirus software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on the app: the output confirmed that the trusted apps included in our dataset did not contain malicious payload.

The malware dataset was obtained from the Drebin dataset. This dataset consists of a total of 5560 applications belonging to 179 different families, classified as malware [10, 30]. To the best of our knowledge, this is the most recent dataset of mobile malware applications currently used to evaluate malware detectors in literature.

The malware dataset includes 28 different families, unevenly represented in the dataset. In order to improve the validity of the experiment, we randomly selected 1000 applications.

4.2.2 Execution traces

We aimed at collecting execution traces which were realistic. To this end, (i) we used a real device, (ii) we generated a number of UI interactions and system events during the execution, and (iii) we collected 10 execution traces for each application (totaling 20 000 traces), in order to mitigate the occurrence of rare conditions and to stress several running options of the AUA.

More in detail, the executions were performed on a Google Nexus 5 with Android 4.4.4 (KitKat). The Nexus 5 is provided with a Qualcomm Snapdragon 800 chipset, a 32-bit processor quad core 2.3 GHz Krait 400 CPU, an Adreno 330 450 MHz GPU, and 2 GB of RAM. The used model had 16 GB of internal memory.

Concerning the UI interactions and system events, we used the monkey tool of the Android Debug Brigde (ADB [2]) version 1.0.32. 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 2000 random UI events in one minute and to stimulate the Android operating system with the following events (one every 5 s starting when the AUA is in foreground): (1) reception of SMS; (2) incoming call; (3) call being answered (to simulate a conversation); (4) call being rejected; (5) network speed changed to GPRS; (6) network speed changed to HSDPA; (7) battery status in charging; (8) battery status discharging; (9) battery status full; (10) battery status 50%; (11) battery status 0%; (12) boot completed. This set of events was selected because it represents an acceptable coverage of all possible events which an app can receive. Moreover, this list takes into account the events which most frequently trigger the payload in Android malware, according to [36, 37].

In order to collect the traces for an AUA, we built a script which interacts with ADB and the connected device and performs the following procedure:

1. copies the AUA into the storage 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, needed for step 6);
6. starts system calls collection;
7. starts Monkey (using the `monkey` command of ADB), instructed to send UI and system events;
8. waits 60 s;
9. kills the AUA (using the PID collected before);
10. uninstalls the AUA (using the `uninstall` command of ADB);
11. deletes the AUA from the device.

Table 1: Statistics about the length $|t|$ of collected sequences forming our dataset $\mathcal{T}$.

Subset	Mean	1st qu.	Median	3rd qu.
Trusted	23 170	6425	15 920	31 820
Malware	12 020	2422	4536	11 160
All	17 600	3397	8198	23 390

Table 2: Percentage of ten most occurring system calls in our dataset, divided between traces collected for trusted (left) and malware (right) applications.

Call (trusted)	Perc.	Call (malware)	Perc.
`clock_gettime`	30.66	`clock_gettime`	28.78
`ioctl`	9.00	`ioctl`	8.85
`recvfrom`	8.67	`recvfrom`	8.85
`futex`	7.89	`epoll_wait`	7.50
`getuid32`	4.96	`getuid32`	6.64
`getpid`	4.84	`futex`	6.61
`epoll_wait`	4.78	`mprotect`	6.34
`mprotect`	4.67	`getpid`	5.64
`sendto`	4.60	`sendto`	2.74
`gettimeofday`	3.19	`cacheflush`	2.00

To collect system calls data (step 6 above) we used strace [4], a tool available on Linux systems. In particular, we used the command `strace -s PID` in order to hook the AUA process and intercept only its system calls.

The machine used to run the script was an Intel Core i5 desktop with 4 GB RAM, equipped with Linux Mint 15.

Tables 1 and 2 show salient information about the collected execution traces: the former shows the statistics about the length $|t|$ of collected sequences. It can be observed that the system calls sequences of trusted apps are, in general, much longer than those of malicious apps. This suggests that the behavior of trusted apps is much richer than the one of malicious apps, which is expected to be basically limited to the execution of the payload. From another point of view, malware apps could exhibit poorer variability behavior than trusted apps and hence recurring sequences, corresponding to the malware fingerprint, should be identifiable.

Table 2 shows the percentage of the ten most occurring system calls in our dataset, divided between traces collected for the trusted (left) and malware (right) applications. It can be seen that the simple occurrences of system calls is not enough to discriminate malicious from trusted apps. As a matter of fact, both malware and trusted applications exhibit the same group of most frequent system calls: `clock_gettime`, `ioctl`, `recvform` are the top three for both the samples.

4.3 Methodology and results

4.3.1 Unseen execution trace

For this scenario, we built the training set T by including 8 (out of 10) traces picked at random for each application in $\mathcal{T}$. The remaining 2 traces for each application were used for testing (i.e., $T' = \mathcal{T} \setminus T$). This way, several traces for each application and for each family were available for the training phase of our method. In other words, in this and the following scenario (Section 4.3.2), we used 80% of the available data for training and the remaining 20% for testing.

Table 3: Results on unseen execution traces.

Method	n	k	Accuracy	FNR	FPR
System calls	1	25	91.1	11.5	6.2
	1	50	92.0	10.0	6.0
	1	75	91.8	10.3	6.2
	2	250	96.1	3.8	4.1
	2	500	96.5	3.4	3.5
	2	750	96.3	4.0	3.4
	3	250	95.5	4.9	4.1
	3	500	96.2	4.4	3.1
	3	750	97.0	3.0	3.0
Permissions		25	56.9	97.7	0.2
		50	60.4	22.4	53.2
		100	69.8	44.6	18.9
		250	88.9	17.0	6.5
		500	90.4	16.3	4.3
		750	90.6	16.0	4.2

After the training phase, we applied our method to each trace in T' and measured the number of classification errors in terms of False Positive Rate (FPR)—i.e., traces of trusted applications classified as coming from malware applications—and False Negative Rate (FNR)—i.e., traces of malware applications classified as coming from trusted applications.

We experimented with different values for the length n of the calls subsequences and the number k of selected features—k' was always set to 2000. We varied n in 1–3 and k in 25–750, when possible—recall that k is the number of selected features among the $|C|^n$ available features, hence we tested only for the values of $k > |C|^n$, with a given n. In order to mitigate lucky and unlucky conditions in the random selection of the traces used for the training phase, we repeated the procedure 3 times for each combination of n, k by varying the composition of T and T'.

The parameters n and k represent a sort of cost of the detection method: the larger their values, the higher the amount of information available to the classifier and, hence, the effort needed to collect it. However, provided that some system mechanism was available to collect the system calls generated by each process, we think that no significant differences exist in an implementation of our method among different values for n and k which we experimented within this analysis.

Table 3 shows the results obtained with our method applied with several combinations of n and k values: for each combination, the table shows the average values of accuracy, FNR, and FPR across the 3 repetitions. The table also shows the results obtained with the baseline method (see Section 4.1). It emerges that the proposed method is largely better than the baseline, as we obtained a best-in-class accuracy of 97% with the former and 91% with the latter. It is important to notice that FNR is low for the highest values of n and k, and that such a value is balanced with FPR.

On the other hand, we note that 91% is a pretty high accuracy: this figure suggests that permissions indeed play an important role in Android malware detection. Yet, they are not enough to effectively discriminating malware, as the rate of false negative keeps high (16%). As a matter of fact, this value can be also explained by the common practice of "overpermissions" (malware writers tend to write a long list of permissions, including those which are not necessary to the app for hiding "suspect" permissions to users). Moreover, the permissions list is an indicator at a coarse grain for identifying malicious behavior, as it could happen that a malicious behavior requires the same permissions of a licit behavior.

Concerning the impact of n and k on the detection accuracy, it can be seen that, for both parameters, the higher the better. For n, this means that longer sequences of system calls better capture the application behavior, and are hence more suitable to constitute a fingerprint. On the other hand, k represents the number of sequences which the method deems relevant, i.e., the fingerprint size: results show that the longer the sequences, the larger the size needed to fully enclose the information represented by the sequences. However, we verified that larger values for k did not deliver improvements in the accuracy. It can be seen that increasing n the accuracy grows and it grows also faster with k.

With $n = 1$, our method just uses frequencies of system calls, rather than sequences: results of Table 3 show that this variant exhibits a lower detection rate, compared to the case in which sequences are considered. This finding suggests that the simple frequency of system calls can be a misleading feature: indeed, there are system calls which are more likely to be used in a malware rather than in a trusted app (e.g., `epoll_wait` and `cacheflush`, as reported in Table 2), but they do not allow to build a highly accurate classifier.

Concerning the execution times, the training phase of our method took, on the average for $n = 3$ and $k = 750$, 1183 s, of which 61 s for the feature selection procedure and 1122 s for training the SVM classifier: we found that these times grow roughly linearly with k' and k, respectively. The actual classification phase of a single trace took, on the average for $n = 3$ and $k = 750$, 18 ms: we found that this time is largely independent from n and grows roughly linearly with k. Finally, the trace processing (i.e., transforming a trace t in a feature vector $\mathbf{f}(t)$) took, on the average for $n = 3$, 135 ms: we found that this time grows roughly exponentially with n. We obtained these figures on a machine powered with a 6 core Intel Xeon E5-2440 (2.40 GHz) equipped with 32 GB of RAM: we remark that we used a single threaded prototype implementation of our method.

4.3.2 Unseen application

For this scenario, we built the training set T by including *all* the traces of 1600 (among 2000) applications picked at random and evenly divided in trusted and malicious.

Then, after the training phase, we applied our method to the traces of the remaining 400 applications and measured accuracy, FPR and FNR. This way, no traces for a given AUA were available for the training; however, traces for applications different from the AUA yet belonging to the same family could have been available for the training.

Table 4: Results on unseen applications, with $n = 3$.

k	Accuracy	FNR	FPR
250	94.1	7.0	4.8
500	94.7	6.4	4.3
750	94.9	6.1	4.2

Since the results of previous experiments show that the best effectiveness can be obtained with $n = 3$, we varied only k in 250–750, in order to investigate if the ideal fingerprint size changes when unseen applications are involved. We repeated the procedure 3 times for each value of k by varying the composition of T and T'.

Table 4 shows the results. The main finding is that our detection method is able to accurately detect ($\approx 95\%$) also unseen malware, provided that it belongs to a family for which some execution traces were available. This finding further validates that our method can build an effective malware fingerprint. Moreover, it supports our assumption that sequences of system calls capture the similarity of behavior among malware applications of known families.

Concerning the impact of k on the accuracy, it can be seen that 750 remains the value which delivers the best accuracy. Moreover, we found the upper limit of k value beyond which no significant accuracy improvement occurs, is lower than in the former scenario. We speculate that this depends on the fact that less families are represented in the training set, hence a slightly smaller fingerprint size is enough to fully exploit the expressiveness of sequences of $n = 3$ system calls.

4.3.3 Unseen family

For this scenario, we repeated the experiment for each malware family of our dataset.

In particular, for a given family, we built the training set T by including *all* the traces of *all* the malicious applications not belonging to that family and *all* the traces of *all* the trusted applications.

Then, after the training phase, we applied our method to all the traces of all the applications of the considered family and measured FNR. This way, no traces for a given AUA were available for the training; moreover, no traces for any application belonging to the same family of the AUA were available for the training. We executed this experimentation with $n = 3$ and $k = 750$.

Table 5 shows the results for the 10 families most represented in our dataset. We remark that, since we were interested in assessing our method ability in detecting malware belonging to unseen families, we did not tested it on trusted traces, which we instead used all for the training. It can be seen that FNR greatly varies among these families: it spans from a minimum of 3.5% to a maximum of 38.5%. Hence, there are some unseen families which could have been detected by our method (GinMaster and zHash); conversely, for other families the detection rate is remarkably lower. We think that this happens because some malware

Table 5: FNR on unseen families, with $n = 3$ and $k = 750$.

Family	FNR
DroidKungFu	31.8
GinMaster	3.5
BaseBridge	4.6
Geinimi	18.6
PJApps	23.7
GloDream	38.5
DroidDream	32.9
zHash	4.3
Bgserv	12.6
Kmin	27.4

family is more different, in terms of behavior, than others: hence, a fingerprint built without that family could not be effective enough to detect malware applications belonging to that family. We conjecture that a larger and more representative training set—as the one which could be available in a practical implementation of our approach—could address this limitation.

5. CONCLUDING REMARKS AND FUTURE WORK

We presented a method for detecting Android malware which is based on the analysis of system calls sequences. The underlying assumption is that a fingerprint of malware behavior can be built which consists of the relative frequencies of a limited number of system calls sequences. This assumption is supported by the fact that the typical evolution of Android malware consists in modifying existing malware, and hence behaviors are often common among different malicious apps. Moreover, capturing app behavior at such a fine grain allows our method to be resilient to known evasion techniques, such as code alteration at level of opcodes, control flow graph, API calls and third party libraries.

We used Machine Learning for building the fingerprint using a training set of execution traces. We assessed our method on 20 000 execution traces of 2000 apps and found that it is very effective, as it obtained a malware detection accuracy of 97%, which is high compared to previous works, most of which have been assessed on a much smaller dataset. Furthermore, our validation differs from the most discussed in literature, as it makes use of real devices rather than emulators or modified kernel, which makes the experiment more realistic. As future work, we plan to investigate the following concerns:

- Evaluate to which extent our method can withstand common evasions techniques [24, 25].
- Consider longer sequences (i.e., greater values of n), since this could allow to capture even better the malware behavior. Unfortunately, since the actual number of possible sequences grows exponentially with n, this also implies coping with a very large problem space.
- Improve the quality of the training data by labelling only those portions of the execution traces of malware applications which actually correspond to malicious behaviors.
- Extend our method to not only classify an entire execution trace as malicious or trusted, but also to specify exactly where, in the trace, there appears to occur the malicious behavior.

6. REFERENCES

[1] Google play. `https://play.google.com/store?hl=it`, last visit 24 November 2014.

[2] Android debug bridge. `http://developer.android.com/tools/help/adb.html`, last visit 25 November 2014.

[3] Google play unofficial python api. `https://github.com/egirault/googleplay-api`, last visit 25 November 2014.

[4] strace-linux man page. `http://linux.die.net/man/1/strace`, last visit 25 November 2014.

[5] Virustotal. `https://www.virustotal.com/`, last visit 25 November 2014.

[6] B. krebs. mobile malcoders pay to google play. `http://krebsonsecurity.com/2013/03/mobile-malcoders-pay-to-google-play/`, last visit 31 November 2014.

[7] Damballa labs. damballa threat report first half 2011. technical report, 2011. `https://www.damballa.com/downloads/r_pubs/Damballa_Threat_Report-First_Half_2011.pdf`, last visit 31 November 2014.

[8] Nqmobile. mobile malware up 163% in 2012, getting even smarter in 2013. `http://ir.nq.com/phoenix.zhtml?c=243152&p=irol-newsArticle&id=1806588`, last visit 31 November 2014.

[9] A. Armando, A. Merlo, and L. Verderame. An empirical evaluation of the android security framework. In *Security and Privacy Protection in Information Processing Systems IFIP Advances in Information and Communication Technology Volume 405, 2013, pp 176-189*, 2013.

[10] 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.

[11] A. Aswini and P. Vinod. Droid permission miner: Mining prominent permissions for android malware analysis. In *Proceedings of 5th International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*, 2014.

[12] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of 17th ACM Conference on Computer and Communications Security*, 2010.

[13] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detection of malicious web pages using system calls sequences. In *Proceedings of the 4th International Workshop on Security and Cognitive Informatics for Homeland Defense (SeCIHD 2014), in conjunction with the International Cross Domain Conference and Workshop (CD-ARES 2014) pp. 226-238*, 2014.

[14] 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.

[15] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova. Detection of malicious applications on android os. In *Proceedings of 4th international conference on Computational forensics*, 2011.

[16] M. Egele, T. Scholte, E. Kirda, and K. C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2), Feb 2012.

[17] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services, pages 281–294*, 2012.

[18] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for android malware detection. In *Proceedings of Seventh International Conference on Computational Intelligence and Security, pp. 1011-1015*, 2011.

[19] 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*, 2014.

[20] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. F. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium, pp. 351-366*, 2009.

[21] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security, pp. 399-412*, 2010.

[22] X. Liu and J. Liu. A two-layered permission-based android malware detection scheme. In *Proceedings of 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2014.

[23] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of 18th ACM Conference on Computer and Communications Security*, 2011.

[24] D. M. T. M. M. Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *Proceedings of International Conference on Availability, Reliability and Security (ARES), pp. 30-39*, 2014.

[25] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, Jan 2014.

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

[27] 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 *Proceedings of 15th International Linux Kongress*, 2008.

[28] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Tech. Report*, 14(1), Feb 2009.

[29] G. Sheran. Android apps security, apress.

[30] 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.

[31] A. Sung, P. Chavez, and S. Mukkamala. Accessminer: using system-centric models for malware protection. In *Proceedings of the 20th Annual Computer Security Applications Conference, pp. 326-334*, 2004.

[32] F. Tchakounté and P. Dayang. System calls analysis of malwares on android. In *International Journal of Science and Tecnology (IJST) Volume, 2 No. 9*, 2013.

[33] X. Wang, V. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the Computer Security Applications Conference, pp. 149-158*, 2009.

[34] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of 7th Asia Joint Conference on Information Security*, 2012.

[35] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium, pp. 27-27*, 2012.

[36] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, 2012.

[37] Y. Zhou and X. Jiang. Android malware, springerbriefs in computer science, 2013.

[38] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.