Dynamic Mobile Malware Detection through System Call-based Image representation

Rosangela Casolare[1], Carlo De Dominicis[2], Giacomo Iadarola[3],
Fabio Martinelli[3], Francesco Mercaldo[1,3*], and Antonella Santone[1]
[1]University of Molise, Pesche (IS), Italy
{rosangela.casolare, francesco.mercaldo, antonella.santone}@unimol.it
[2]University of Padova, Padova, Italy
carlo.dedominicis.1@studenti.unipd.it
[3]IIT-CNR, Pisa, Italy
{giacomo.iadarola, fabio.martinelli}@iit.cnr.it

Received: December 30, 2020; Accepted: February 18, 2021; Published: March 31, 2021

Abstract

Mobile devices, with particular regard to the ones equipped with the Android operating system, are currently targeted by malicious writers that continuously develop harmful code able to gather private and sensitive information for our smartphones and tablets. The signature provided by the antimalware demonstrated to be not effective with new malware or malicious payload obfuscated with aggressive morphing techniques. Current literature in malware detection proposes methods exploiting both static (i.e., analysing the source code structure) than dynamic analysis (i.e., considering characteristics gathered when the application is running). In this paper we propose the representation of an application in terms of image obtained from the system call trace. Thus, we consider this representation to input a classifier to automatically discriminate whether an application under analysis is malware or legitimate. We perform an experimental analysis with several machine and deep learning classification algorithm evaluating a dataset composed by 6817 real-world malware and legitimate samples. We obtained an accuracy up to 0.89, showing the effectiveness of the proposed approach.

Keywords: mobile security, malware analysis, system call, dynamic analysis, Android, machine learning, deep learning, classification

1 Introduction

Malware detection is referring to the process of detecting the presence of malware, which is malicious code developed by attackers with the purpose of creating damage (for instance, to silently exfiltrate sensitive and private information of unaware users), within a system.

Malware detection is a growing problem and it is mainly applying to mobile platforms due to their spread. In addition to the spread of mobile devices, we have a huge amount of applications published on (official and unofficial) stores, which is too large to allow to manually scan each application for malware behavior [20].

There are two main technologies to defend unaware users against this phenomenon: signature-based approach and malware detection through behavior analysis. Signature-based malware detection is used to identify malware that security analysts already know. This technique, therefore, is not able to identify

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 12(1):44-63, Mar. 2021
DOI:*10.22667/JOWUA.2021.03.31.044*
*Corresponding author: University of Molise, Campobasso 86100, Italy, Tel: +39 0874 40 41

new versions of malicious code, as it does not know their signature. Newly released forms of malware can only be distinguished from benign files and activities by behavioral analysis [4].

Behavior-based malware detection analyzes an object based on a well-defined sequence of actions, which is characteristic of a type of malware. Then the object behavior is analyzed for suspicious activity: attempting to perform abnormal or unauthorized actions (i.e., disabling security checks, installing rootkits, etc.) indicates that the object is malicious.

The assessment of malicious behavior can be done by running a *dynamic analysis* during its execution, or, an alternative to detect the potential threat can be the *static analysis*, which looks for dangerous functionality by analyzing the code and structure of the object. The static analysis technique, relatively to Android environment, is based on reverse engineering of the Android application package (*.apk*). This procedure involves scanning *AndroidManifest.xml* and *classes.dex* files for malicious code without having to install and run the application. Dynamic analysis, on the other hand, examines the behavior of the application during its execution, for this reason the intrusion detection system in Android analyzes lists of processes, system call, network traffic and other features that allow the detection of intrusions [19].

Behavioral analysis for malware detection evaluates the application when it is running: it checks all requests for access to files, processes and connections. This identifies all suspicious activities, which allow us to understand if a file is malicious in advance, before it is released on the network and performs a malicious action.

Among the targets at the highest risk of being hit by malware we find unattended personal computers: the risk of infection is especially for those connected to a computer network, since it is possible to propagate the attack to all computers connected to that network. In recent years, however, smartphones and tablets have spread rapidly; these devices contain a huge amount of personal information (i.e., photographs, financial data, messages, emails, etc.), probably much more than that contained on computers.

Following the spread of these devices, malware writers have started migrating their attacks to new platforms. Among the most popular operating systems on mobile platforms we find Android: in fact in October 2020 it recorded a market percentage of 72.92%, followed by iOS with 26.53%, the remaining percentage is occupied by less known systems (i.e., KaiOS)[1].

In addition to its diffusion, which allows malicious code writers to be able to launch attacks on a large number of devices, the nature of Android also arouses their attention, as we are talking about an open operating system: yes, it allows for user customization, but on the other hand it makes the attackers' job much easier.

The number of malware attacks is always on the rise, just consider that in May 2020, 430,000 attacks were launched and that in the space of a month the number increased by 3.6%, recording an increase of 6.26% between July and August 2020[2]. These numbers refer only to attacks targeting Android devices, not counting those targeting other platforms and other types of devices.

The approach we propose is based on dynamic techniques for the detection of malicious behavior by Android applications. The approach considers the system call sequences. Usually, the malware evolution process is based on changes made to existing malware. Malware writers use obfuscation techniques or payloads already implemented in previous malware, to improve infection mechanisms or tend to combine them [6].

Dynamic analysis versus static analysis takes longer, as expecting applications to run for threat detection requires a longer amount of time to apply. Its slowness is the price to pay which however allows dynamic analysis to identify malware that could not be identified with static analysis, such as those that change during their execution.

[1] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
[2] https://news.drweb.com/show/review/?i=13991&lng=en

This approach represents an extension of the preliminary paper [7] published in the International Workshop on Security of Mobile Applications belonging to the 15th International Conference on Availability, Reliability and Security (ARES). We itemize the differences with respect to the previous work:

- the preliminary paper [7] is related only to malicious repackaged applications, while in this paper we consider also Android malware with different installation methods (i.e., *standalone*, *repackaging* and *update attack*);

- we consider different features with respect to the work proposed in [7]. As a matter of fact, the paper in [7] considers only four features, while the approach we propose takes into account 2141 features, extracted with five different feature extraction algorithms (GIST, Gabor, Autocolor Correlogram, Color Layout and Simple Color);

- the preliminary method published in [7] considers static analysis, while in this paper we propose an approach considering a dynamic analysis;

- the feature set considered in the preliminary work in [7] is obtained by exploiting the Androguard tool, working only on Android application. Differently the proposed approach, considering to make the detection requires just the system call traces, is general purpose and thus not dependant from the target operating systems of the applications to analyse;

- we represent an Android application as image obtained from the system call trace, while in the preliminary work in [7] we propose an image based on the number of identical, similar, new and deleted methods between two applications (we remark that these values are obtained with a static analysis);

- we work on a larger dataset: in the method proposed in [7] we evaluated 36 applications; differently, we evaluate the proposed approach by exploiting more than 6000 real-world Android applications. In detail we consider in the experimental analysis 3355 malware (belonging to 10 different malware families) and 3462 legitimate applications;

- in the evaluation of the workshop paper in [7] we consider only models built with machine learning techniques, while in this work we experiment with machine learning (ML) but also with deep learning (DL) techniques.

The paper proceeds as follows: in Section 2 are described the background notions about the techniques used in the proposed work; in Section 3 current state-of-the-art literature is analyzed and commented; in Section 4 we describe our method starting from the system call extraction to generate the PNG image and then to feature generation and classification; in Section 5 are described the dataset used for the experiment and the results obtained; finally conclusion and future research lines are drawn in Section 6.

2 Background

Our methodology relies on different ML, DL and feature extraction techniques. Briefly, we describe the main functionalities and knowledge on these models and techniques, and we refer to the literature for further information.

Table 1: Feature vector sizes with regard of the techniques taken into account.

Feature Extraction	No. Attributes
Autocolor	1024
GIST	960
Simple Color	64
Gabor	60
Color Layout	33

2.1 Feature Extraction

Feature extraction techniques are widely used in image classification tasks, also in malware detection domain [25, 24, 11]. In our methodology, they serve two purposes: focus the analysis on specific details (i.e., features) and reduce the size of the images to fixed-size vector dimensions. The sizes of the generated vector are reported in Table 1.

As shown from Table 1 five different feature extraction algorithms are considered for a total of 2141 features obtained. Below we briefly describe the feature extraction algorithms exploited in this work:

GIST The GIST descriptor [27] extracts the gradient information of an image, such as orientations, edges and scales, and summarizes them into a 960-length feature vector. It works analyzing specific areas of the image and summarized the information in a vector. Iteratively, the vector contains an approximate description of the entire image.

Gabor The Gabor filters study the frequency of patterns in the image, moving trough the image by region and point of analysis, and they are widely used for texture analysis [23]. The filters used in our approach generate a 60-length feature vector.

The images generated by the system call, similarly to any other image generated by malware or code analysis, do not contain identifiable shapes because they are not related to real objects, as it happens with photographs. On the contrary, the pixels of these images form mainly distribution of colours which may look random noise to the human eye. Thus, colour-based feature extraction techniques may play a fundamental role in improving the performance of the classification task. We tested some colour-based feature extraction techniques, based also on our previous work [11].

Autocolor Correlogram The Autocolor Correlogram filters summarize information on the spatial distribution of the colours in an image [26]. It merges statistics from the colour histogram with spatial similarities of image sub-areas. The vector extracted has size of 1024, the biggest one among the features extraction techniques taken into account in our methodology.

Color Layout The Color layout filter extracts the MPEG7 features, the multimedia content description standard [15], which generate a 33-length features vector. The image is analyzed in blocks, and colour distribution and averages are extracted to the output vector.

Simple Color The Simple Color Histogram filter is aimed to generate from an image under analysis three histograms [31]: the first one for the red color, the second one for the green color and the third for the blue one. Each histogram is composed by 32 bins and Each bin contains a count of the pixels in the image that fall into that bin.

2.2 Classification Models

In this work we adopted some of the most used ML and DL models to provide a short but complete overview of the classification for this kind of task. The classification models belong to the supervised learning models, where the training phase is guided by the pair input-output, where output represent a desired value (the class or label of the input sample).

Random Forest The Random Forest (RF) is a decision tree model [18], which construct the tree graph of decisions based on the training phase. The nodes of the tree contain conditional statements while the leaves contain the output. The classification task is performed by traversing the decision tree using the input and starting from the root node. The RF trees generation process produces different trees, independently constructed by subsets of training data, and then aggregated them based on their validation results.

Support-vector machine The Support-vector machine (SVM) [32] obtains maximum effectiveness in binary classification problems although, it is also used for multiclass classification problems. The SVM is based on the idea of finding a hyperplane that best divides a data set into two classes. For a classification activity with only two spatial dimensions, a hyperplane is represented as a line that separates and classifies a set of data. Support vectors are the data points closest to the hyperplane. These points depend on the set of data being analyzed and if they are removed or modified they alter the position of the dividing hyperplane. For this reason, they can be considered the critical elements of a dataset.

Multi-layer perceptron The MLP [10] is one of the most used feedforward artificial neural network models. It consists of, at least, three layers of nodes, namely input, hidden and output layers. In the inference step, the data starting from the input layers flows trough all the nodes and reach the output layers. Each node applies a nonlinear activation function, usually the Rectified Linear Unit (Relu). In the training phase, a backward step (called backpropagation) changes the weights of the connection of the neurons in order to minimize the error between the prediction and the correct label and then train the model.

Convolutional Neural Network The CNN [17] are deep neural network models well-known for their good results in image classification tasks. The key layers of these models are the convolutional layers, which extract information from the input pixels in a structure called 'feature map', that gathers the relevant information. The convolutional layers are followed by series of dense layers (similarly to the MLP), which perform the classification task.

3 Related Work

On the research area covered in this work, there are several studies, for example the researchers in [19] have studied the behavior of 10 popular Android malware families by focusing on the system call pattern of these families. They extracted the system call trace of 345 malicious applications from 10 Android malware families, such as: FakeInstaller, Opfake, BaseBridge, Iconosys, Plankton, DroidKungFu, Kmin, Gappusin and Adrd using the strace Android tool and compared them with the system calls model of 300 benign applications to verify the behavior of malicious applications. The researchers observed that malicious applications invoke certain system calls more frequently than benign applications.

In [6], the researchers designed a method for automatically selecting system calls by choosing them from a very large set of sequences: the most useful ones were selected for malware detection. The

method also, given the defined fingerprint in terms of frequencies of selected sequences of system calls, classifies an execution trace as malware or non-malware.

DaVinci [9] is an Android kernel module for dynamic system call analysis. It provides pre-configured high-level profiles to allow easy analysis of low-level system calls and runs on applications without requiring reverse engineering.

The authors of [5] have implemented an approach to perform dynamic analysis of Android applications and classify them as malicious or non-malicious. They have developed a system call capture system that collects and extracts system call traces of the applications installed on the device during their run-time interactions. Once data on system calls has been collected, it is analyzed in order to identify the type of behavior of Android applications. 50 malicious applications obtained from the Android Malware Genome Project and 50 benign applications obtained from the Google Play Store were analyzed. The goal is to classify the applications' behavior, considering the frequency of system calls made by each application as the core feature set. For the classification of the application as harmful or benign, the J48 Decision Tree algorithm and the Random Forest algorithm were used.

The researchers in [12] have thought instead, of detecting Android gaming malware with a dynamic system, the method is based on the analysis of system calls to classify malicious and legitimate games. The dynamic analysis was carried out during the runtime of the games, reporting the similarities and differences between benign game system calls and malware. It shows how the behavior of malicious activities via system calls during their activity allows an easier detection of malicious applications.

In [16] using multimodal deep learning to study both statically and dinamically the application's behavior, researchers found valuable results in Android malware detections, bringing to light the effectiveness of deep learning.

In DL-Droid [2] dynamic analysis is made extracting logs files directly from *.apk* installed on real phones. The extraction is made two times for each applications because of the chosen test input generation (i.e. stateless and stateful). Than the features about system calls are extracted and ranked using InfoGain (an algorithm used to gain informations about feature ranking). So the ranked list is classified using a Deep Neural Network

In [1] are compared differences between different machine learning techniques, such as input, analysis, dataset type, performance evaluation critera and algorithms. About this last one the research highlight some like Random Forrest, SVM and Naive Bayesian as the best ones to work with because of the higher accuracy rate. However, it emphasize the lack of a proper general malware detection model for both supervised and unsupervised machine learning classifiers.

DATDroid [30] a Dynamic Analysis Technique for Android malware detection, is composed by three steps: feature extraction, features selection and classification. The features are extracted from different sources (i.e. collection of system call, record of CPU and memory usage, collection of network packets), than occur a selection on these features to optimize the malware detection performance, hence a classification between benign application and malware is done using machine learning.

GSDroid [28] is a mechanism used to extract and represent low dimensional features. It can detect malicious behaviors in Android applications through graph signal processing based on system calls diagrams, this last one constructed considering the control dependency relations between system calls. The features considered are part of an intelligent expert system that is based on the machine learning technique, allowing to automate the malware detection.

In [29] researchers propose a new mechanism for detecting malware, based on TAN (Tree Augmented naive Bayes) which uses conditional dependencies between the relevant static and dynamic characteristics (i.e., system calls) useful for running applications. They trained three logistic regression classifiers, one for API calls, another for permissions, and the last for application system calls. Their output relationships then has been modeled as a TAN to identify when an application is malicious.

Authors in [13] have developed a new approach named *Artificial Malware-based Detection* (AMD)

in which an innovative genetic algorithm is used to generate, in artificial way, malware models. So the approach uses extracted malware models and also the artificial malware models. The evolutionary algorithm also works on the API call sequences' population to find new malware behaviors, applying some well-defined evolution rules. The artificial malware models are inserted into the set of unreliable behaviors, to create diversification between malware samples in order to increase the detection rate.

Researchers in [33] propose a new approach called Back-Propagation Neural Network on Markov Chains from System Call Sequences (BMSCS), which exploits the back-propagation neural network (BPNN) to perform the classification of the transition probability matrix of the Markov chain, which is generated by the sequences of system calls for the detection of Android malware. In this paper, the authors start from the thought that the probabilities of switching from one system call to another are different between malicious and benign applications.

In [8] the authors use a static approach, based on formal methods, which exploiting the model checking technique allows to perform a detection of the applications involved in colluding attacks. This attack in order to be launched requires the communication of two or more applications through the use of covert channels (i.e., unconventional communication channels such as Android shared preferences). Differently in this paper, we propose an approach to dynamically detect Android malware through the analysis of system calls. In detail we propose to represent applications in terms of images obtained from the system call traces.

4 The Method

In this section we describe the proposed approach for malware detection in Android environment by representing system call sequences in terms of images.

Figure 1 depicts our method, while the next subsections describe the methodology steps in details.

4.1 System Call Extraction

Below we explain how we gather the system call from an Android application. As a matter of fact, the rationale behind this step is to capture and store, in a textual format, the system call traces generated by running applications. For this purpose, the *.apk* file of each Android application is installed and initialised on an Android device emulator. Successively, a set of 25 different operating system events [34, 14] is generated (at regular time intervals equal to 10 seconds) and sent to the emulator and the correspondent sequence of system calls is obtained. In Table 2 are shown the operating system events we consider for this purpose.

We exploit a set of operating system events because several previous papers [34, 14] demonstrated that these events are considered by malicious writers to activate the payloads in Android environment. In Table 2, the first row shows the `BOOT` event i.e.,one of the most exploited operating system event to activate Android malware. This is not surprising because the `BOOT` event is sent to all the applications installed on the Android device in the instant in which the operating system terminates its booting process: this represents a perfect time for a payload to start its malicious action [21]. Malicious writers usually exploits the `BOOT` events event, to make able a malicious payload to start itself without any intervention or interaction of the unaware user with the Android operating system.

Other system events typically exploited by malicious writers are represented by the `ACTION_ANSWER` and `NEW_OUTGOING_CALL` events (respectively in the second and third row in Table 2): these events are sent in broadcast to the operating system (and, consequently, to all the running applications) in the moment in which a call phone is received or initialised by the unaware user.

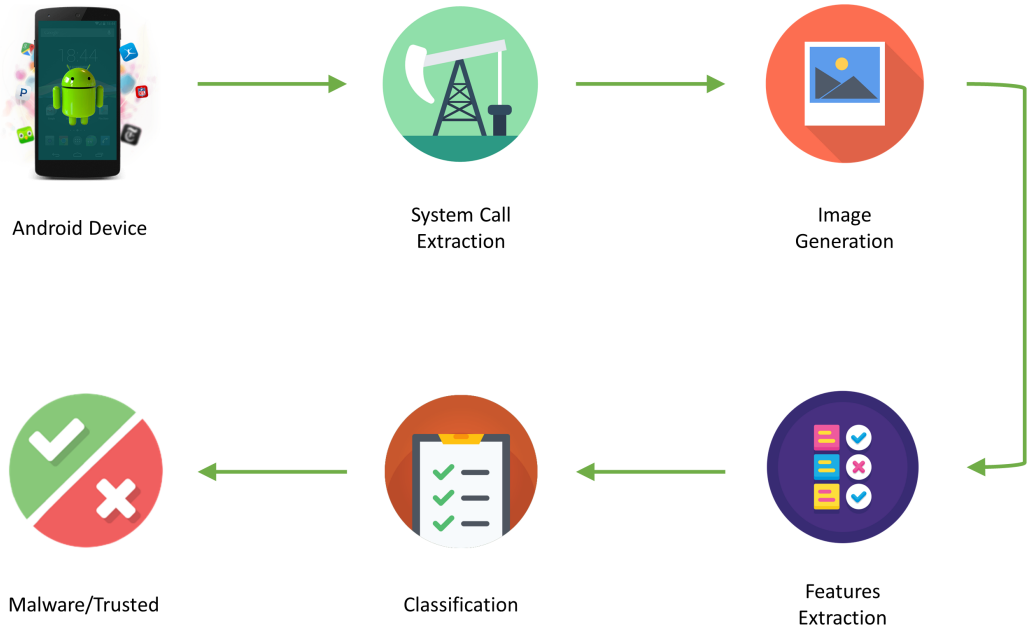

Figure 1: The proposed method for malware detection in Android using system call.

The system call retrieval from the Android application is performed by a shell script developed by authors aimed to perform in sequence a set of actions below described:

- initialisation of the target Android device emulator;
- installation the *.apk* file of the application under analysis on the Android emulator;
- wait until a stable state of the device is reached (i.e., when *epoll_wait* is received and the application under analysis is waiting for user input or a system event to occur);
- start the retrieve the system call traces;
- send one event from the operating system events shown in Table 2;
- send the choose operating system event to the application under analysis;
- capture system calls generated by the application until a stable state is reached;
- selection of a new operating system event (i.e., the operating system event following the one previously selected) and repeat the steps above to capture system call traces for this new event;
- repetition of the step above until all 25 operating system events in Table 2 have been considered (i.e., the Android application was stimulated with all the system events depicted in Table 2);
- stop the system call capture and save the obtained the system call trace;

Table 2: System events considered for the malicious payload activation.

#	System Event	Description
1	*BOOT_COMPLETED*	Able to catch the boot completed
2	*ACTION_ANSWER*	Incoming call
3	*NEW_OUTGOING_CALL*	Outgoing call
4	*ACTION_POWER_CONNECTED*	Battery status in charging
5	*ACTION_POWER_DISCONNECTED*	Battery status discharging
6	*BATTERY_OKAY*	Battery full charged
7	*BATTERY_LOW*	Battery status at 50%
8	*BATTERY_EMPTY*	Battery status at 0%
9	*SMS_RECEIVED*	Reception of SMS
10	*AIRPLANE_MODE*	The user has switched the phone into or out of Airplane Mode
11	*BATTERY_CHANGED*	Battery status changed
12	*CONFIGURATION_CHANGED*	The current device Configuration (orientation, locale, etc) has changed
13	*DATA_SMS_RECEIVED*	A new data based SMS message has been received by the device
14	*DATE_CHANGED*	Receives data changed events
15	*DEVICE_STORAGE_LOW*	Free storage on device is less than 10% of total space
16	*DEVICE_STORAGE_OK*	Free storage on device is adequate
17	*INPUT_METHOD_CHANGED*	An input method has been changed
18	*PROVIDER_CHANGED*	Providers publish new events or items that the user may be especially interested in
19	*PROXY_CHANGE*	Variation of proxy configuration
20	*SCAN_RESULTS*	An access point scan has completed, and results are available from the supplicant
21	*SENDTO*	Send a message to someone specified by the data
22	*SIM_FULL*	The SIM storage for SMS messages is full
23	*SMS_SERVICE*	CDMA SMS has been received containing Service Category Program Data
24	*STATE_CHANGED*	The state of Bluetooth adapter has been changed.
25	*WAP_PUSH_RECEIVED*	A new WAP PUSH message has been received by the device

- kill the process of the Android application under analysis;
- stop the Android emulator;
- revert its disk to a clean snapshot (i.e., before the installation of Android application under analysis).

Moreover we exploit the monkey tool belonging to the Android Debug Bridge (ADB[3]) version 1.0.32, to generate pseudo-random user events such as, for instance, clicks, touches, or gestures (with the aim to simulate the user interaction with the Android application under analysis).

[3] `https://developer.android.com/studio/command-line/adb`

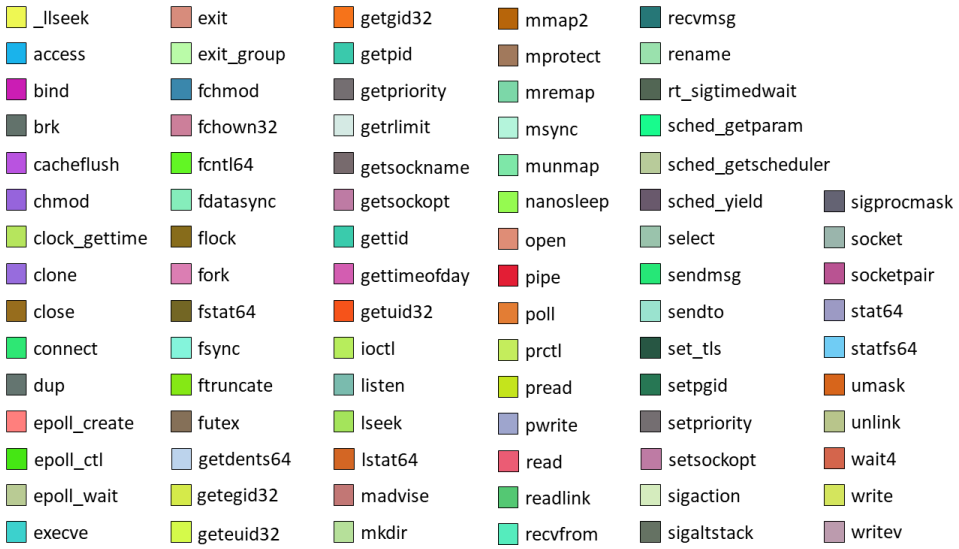

Figure 2: System call legend.

Listing 1: Snippet of code where system calls turns into single pixels.

To collect the syscall traces we consider strace[4], a tool freely available on Linux operating systems. In detail, we invoke the command *strace -s PID* to hook the running Android application process to intercept only syscalls generated by the application under analysis process.

4.2 Image Generation

Basically, starting from a log of system calls, we extract one by one the single calls and respecting the order given by the log we build the image. In Figure 1 we can see the code snippet with which we convert system calls: giving in input the single system call, through static calculus we assign a portion to remove (used at the moment we are going to compress the string). Than, in a loop, the system call turns into a bit string, converting one by one every single letter. In the lasts rows the bit string is compressed in order to became a string of 24 total bits that can identify uniquely the system call that it represent, hence the compressed string is divided in 3 sub-strings, that are converted in decimal obtaining the values for the RGB conversion used for the images representation as PNG file. Figure 2 shows the RGB value associated to each system call, it is useful to quickly identify, which are the system calls that compose an image.

As we can see, following are reported some image representation: in Figure 3 it is possible to see an image conversion about a trusted application, that is composed by colored squares, where each color is associated to the relative system calls. In particular both the malware samples, showed in Figure 4, are belonging to the same family i.e., Plankton. In the right image in Figure 4[5] and in the left image in Figure 4[6], we can see the representations about two different malware applications, that in this case they look similar to each other but different from the trusted application representation. In fact, the malware images have some common parts like the two brown bands, that on the contrary are absent in the trusted image. In this way, we already have a visual impact that allows us to notice the differences between

[4] `https://man7.org/linux/man-pages/man1/strace.1.html`
[5] identified by the *0a3be4156b705957d201a86250d0d7f4c5470f1737ed6d438a129a39b475397b* hash
[6] identified by the *0abccad6ac3523c968c44dec57d7a7bd50400a55eba02dc37c7c2f6e6a759292* hash

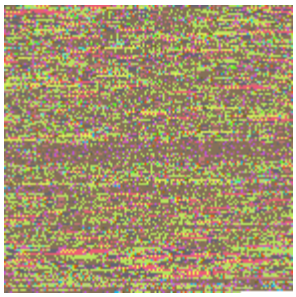

Figure 3: Image representing a trusted application.

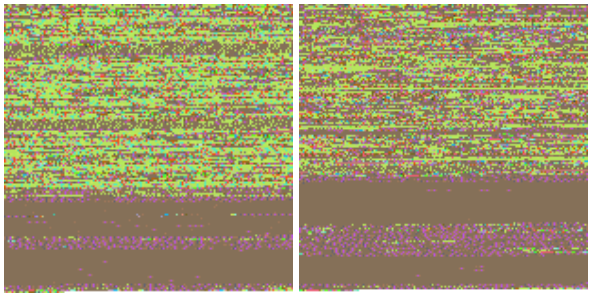

Figure 4: Images representing two different malware applications.

trusted and malware applications. Hence this method of representation through image generation, could be used to discriminate quickly malware applications from trusted ones.

4.3 Features Extraction

The size of the images depends on the system calls extracted in the dynamic analysis. Even if the malware were run in the controlled environment for a fixed amount of time, each malware invokes different combinations of system calls with different frequencies. Thus, the size of the images generated differs from one malware to another (the average size in our dataset was around 180x180x3). Moreover, the malware do not exhibit malicious operations most of the time; hence, the images that collect the sequences of system calls contain also legitimate operations, which result in noise for the malware detection task and may mislead the classification.

Therefore, features were extracted by the images. This preprocessing step converts the images to fixed-size vectors, compress the information and reduce the image size. Several features extraction techniques were adopted and tested, the experimental results are reported in Section 5.2.

Unlike the standard image classification task such as animal/clothes or number detection task, the images produced by the system calls do not contain recognizable shapes or object. They contain a pixel distribution of colours that encode the dynamic analysis information. Intuitively, the colour-based features extractor should produce more rich and useful features vectors than the ones focusing on edges and borders. Nevertheless, we applied and combined many of them to find the most suitable one for our classification task, to get the most from each descriptor techniques.

In detail, we collect vectors by combining the techniques and features reported in Table 4.

4.4 Classification

The image generated from the system calls and then vectorized, can be input in ML and DL models to perform the classification tasks. Our interest mainly regards the feasibility of using the system calls

as images to distinguish between malware and trusted applications, thus we experiment with different models and approaches. We tested both standard ML models (Random Forest and SVM) and also basic DL models (MLP and CNN) to provide a rough overview and study the different results.

The classification task proceeds with a standard approach. The vectors coming from different feature extraction techniques were grouped together in different datasets (see Table 4). The datasets are then split into training and test sets, and the models are trained on the training sets. The results of the test sets are collected and evaluated. The different dataset compositions and models applied provide an interesting overview, which allows studying the robustness of a model with regards to the datasets. Moreover, the experiments on different datasets provide information on the capabilities of the features extraction techniques to summarize interesting information for the problem under analysis, and then the efficiency of the features extraction techniques compared one to each other.

5 Experiment

In this section we describe, respectively, the real-world dataset involved in the experimental analysis and the results of the ML and the DL classifications.

5.1 The dataset

The dataset considered in the experimental analysis was gathered from two different repositories: relating to the malicious samples we obtained real-world Android malware from the Drebin dataset [3, 22], a very well-known collection of malware largely considered by malware analysis researchers, including the most widespread Android families. The malware dataset is freely available for research purposes [7].

The considered malware dataset consists of different Android malicious families characterized by different installation methods: (i) *standalone*, applications that intentionally include malicious functionalities; (ii) *repackaging*, known and common (legitimate) applications that are first disassembled, then the malicious payload is added, and finally are re-assembled and distributed as a new version (of the original application); and (iii) *update attack*, applications that initially do not show harmful behaviors and download an update containing the malicious payload, at runtime.

The malware dataset is also partitioned according to the *malware family*; each family contains malicious samples sharing several characteristics: the payload installation, the kind of attack and the events triggering the malicious payload [34].

Table 3 shows the 10 malware families involved in the experiment (i.e., the most populous ones in terms of malicious samples) with the details of the installation types, the kinds of attack, the events which activate the payload, the discovery date and the number of malicious samples belonging to each family.

To gather legitimate applications, we crawled the official app store of Google[8], by using an open-source crawler[9]. The obtained collection includes samples belonging to all the different categories available on the market.

We analyzed the dataset with the VirusTotal service[10], a web service able to run 61 commercial and free antimalware: this analysis confirmed that the trusted applications did not contain malicious payload while the malicious ones were actually recognized as malware.

The (malicious and legitimate) dataset is composed by 6817 samples: 3355 malicious (belonging to 10 different malicious families) and 3462 trusted.

[7] https://www.sec.cs.tu-bs.de/~danarp/drebin/
[8] https://play.google.com/store
[9] https://github.com/liato/android-market-api-py
[10] https://www.virustotal.com/

Table 3: The malware families.

Family	Description	Inst.	Events
Geinimi	It has the potential to receive commands from a remote server that allows the owner of that server to control the phone	r	MAIN
Plankton	Advance the update attack by stealthily upgrading certain components in the host applications, it does not require user approval.	u	MAIN
BaseBridge	It sends information to a remote server running one or more malicious services in background	r,u	BOOT, SMS, NET, BATT
Kmin	It is similar to BaseBridge, but does not kill antimalware processes	s	BOOT
GinMaster	It contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and install applications	r	BOOT
Opfake	It demands payment for the application content through premium text messages	r	MAIN
FakeInstaller	SMS trojan adding server-side polymorphism, obfuscation, antireversing techniques and frequent recompilation	r	BOOT, SMS
DroidDream	It is able to obtain root privileges, to obtain access to SD files and to change the device settings	r	MAIN
DroidKungFu	Its payload is able to run along with the original application process. It steals theIMEI, the OS Version, the device model and saves this information into a local file	r	BOOT, BATT
Adrd	It uploads infected cell phone's information to the control server every 6 hours and receive its commands, causing a great amount of network traffic	r	BOOT, NET, CALL

From the malicious and legitimate dataset we gathered the system call sequences with the procedure explained in the previous section. Subsequently, from each system call trace we generated the relative image representation and we extracted, for each image, the features with the GIST, the Gabor, the Autocolor Correlogram, the Color Layout and the Simple Color feature extractors: at the end of this process, for each image obtained from the system call traces, five different feature sets are obtained.

Once obtained the feature sets, we performed several combinations in order to obtain more complex feature sets with the aim to better capture the differences between malicious and legitimate applications. The list of feature set combinations is shown in Table 4.

5.2 Results

In this section we present the classification results we obtained by evaluated the different feature sets with different models built with machine and deep learning algorithms. We firstly present the machine learning experimental results and, subsequently, the deep learning ones.

Four metrics are considered to evaluate the performance of the classification with machine and deep

Table 4: Feature sets involved in the experimental analysis.

Feature Set	GIST	Gabor	Autocolor	Color	Simple	Vector size
FS1			X			1024
FS2			X		X	1088
FS3		X	X			1084
FS4		X	X		X	1148
FS5		X	X	X	X	1181
FS6	X		X			1984
FS7	X	X	X			2044
FS8	X	X	X	X	X	2141

learning algorithms: Precision, Recall, F-Measure and Accuracy.

The precision has been computed as the proportion of the Android application that are truly belonging to class (i.e., malware or legitimate) among all those applications which were assigned to the class. It is the ratio of the number of relevant applications retrieved to the total number of irrelevant and relevant applications retrieved:

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

where *tp* indicates the number of true positives and *fp* indicates the number of false positives.

The recall has been computed as the proportion of Android applications that were assigned to (malware or legitimate) class, among all the Android applications truly belonging to the class, i.e., how much part of the (malware or legitimate) class was captured. It is the ratio of the number of relevant applications retrieved to the total number of relevant applications:

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

where *tp* indicates the number of true positives and *fn* indicates the number of false negatives.

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

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

The accuracy of a measurement system is the degree of closeness of measurements of a quantity to that quantity's true value: it represents the fraction of the classifications that are correct. It is the sum of true positives and negatives divided all the evaluated Android applications:

$$Accuracy = \frac{tp+tn}{tp+fn+fp+tn}$$

where *tp* indicates the number of true positives, *tn* indicates the number of true negatives, *fn* indicates the number of false negatives, *fp* indicates the number of false positives.

With regard to machine learning experiment settings, for model building, we defined $T_{detection}$ as a set of labeled messages $\{(M_{detection}, l_{detection})\}$, where each $M_{detection}$ is the label associated to a $l_{detection}$ $\in \{ malware, legitimate\}$. For each $M_{detection}$ we built a feature vector F $\in R_y$, where y is the number of the features considered in training phase.

Table 5: Machine learning experimental results.

Feature	*RandomForest*				*SVM*			
Set	**Precision**	**Recall**	**F-Measure**	**Accuracy**	**Precision**	**Recall**	**F-Measure**	**Accuracy**
FS1	0.885	0.885	0.885	0.885	0.837	0.836	0.836	0.836
FS2	0.894	0.894	0.894	0.894	0.850	0.850	0.850	0.850
FS3	0.845	0.844	0.844	0.844	0.837	0.836	0.836	0.836
FS4	0.865	0.865	0.865	0.865	0.846	0.846	0.846	0.846
FS5	0.861	0.861	0.861	0.861	0.849	0.849	0.849	0.849
FS6	0.812	0.810	0.810	0.811	0.820	0.819	0.819	0.819
FS7	0.807	0.805	0.805	0.806	0.822	0.822	0.822	0.822
FS8	0.822	0.821	0.821	0.821	0.854	0.854	0.854	0.854
AVG	0.849	0.848	0.848	0.848	0.839	0.839	0.839	0.839

With regard to the learning phase, a k-fold cross-validation is exploited: the dataset is randomly partitioned into k subsets. A single subset is retained as the validation dataset for testing the 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 model method by exploiting 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;
4. apply the learned classifier to each element of T'.

Each classification was performed using 90% of the dataset as training dataset and 10% as testing dataset employing the full feature set.

Table 5 shows the experimental results we obtained from the machine learning models.
Similar experiments were performed using the MLP and CNN models, whose layers architectures are reported in Table 6. The parameter column refers to the percentage of neurons deactivated in the Dropout layers. The output shape of the Convolutional layers (layers 1-8 of the CNN) depends on the input shape, which differs from the feature set under analysis.

The MLP takes as input vectors in one dimension, thus the dataset of vectors directly applied on the model. On the other hand, the CNN requires images (matrices of pixels) as input, then the 1D vectors were reshaped into 2D matrices and 0-paddings were added, if necessary, to create squared matrices. Table 7 shows the results obtained from the deep learning experiments.

The performance results in test, reported in Table 5 and Table 7, lead to some interesting evaluations. First of all, the "colour-related" features (namely, the Autocolor Correlogram, the Color Layout and the Simple Color) perform better than the other features. Indeed, the feature sets on which the colour-related features are predominant (FS1-5) achieved better results than the FS6 and FS7, based on GIST and Gabor features. This difference is consistent on the ML models, especially for the Random Forest model, while the results gap is smaller in the DL models. We expected this outcome because the input images are simply a distribution of colours, with no detectable shape or object, thus the colour features can extract more useful information than features extractor based on edges, rotations and shapes features.

Table 6: Deep learning models architecture.

# Layer	*MLP*			*CNN*		
	Type	**Output Shape**	**Parameter**	**Type**	**Output Shape**	**Parameter**
1	Dense	500	-	Convolutional	$X_1, X_1, 32$	-
2	Dense	700	-	MaxPooling	$X_2, X_2, 32$	-
3	Dropout	700	0.5	Convolutional	$X_3, X_3, 64$	-
4	Dense	1000	-	MaxPooling	$X_4, X_4, 64$	-
5	Dense	500	-	Convolutional	$X_5, X_5, 128$	-
6	Dropout	500	0.5	MaxPooling	$X_6, X_6, 128$	-
7	Dense	250	-	Flatten	X_7	-
8	Dense	100	-	Dropout	X_7	0.5
9	Dropout	100	0.5	Dense	512	-
10	Dense	50	-	Dropout	512	0.5
11	Dense	2	-	Dense	256	-
12	-	-	-	Dropout	256	0.5
13	-	-	-	Dense	2	-

Table 7: Deep learning experimental results.

Feature	*MLP*				*CNN*			
Set	**Precision**	**Recall**	**F-Measure**	**Accuracy**	**Precision**	**Recall**	**F-Measure**	**Accuracy**
FS1	0.879	0.879	0.879	0.879	0.828	0.828	0.828	0.828
FS2	0.884	0.884	0.884	0.884	0.846	0.846	0.846	0.846
FS3	0.866	0.866	0.866	0.866	0.846	0.846	0.846	0.846
FS4	0.833	0.833	0.833	0.833	0.821	0.821	0.821	0.821
FS5	0.87	0.87	0.87	0.87	0.835	0.835	0.835	0.835
FS6	0.86	0.86	0.86	0.86	0.855	0.855	0.855	0.855
FS7	0.868	0.868	0.868	0.86	0.855	0.855	0.855	0.855
FS8	0.87	0.87	0.87	0.87	0.854	0.854	0.854	0.854
AVG	0.866	0.866	0.866	0.866	0.843	0.843	0.843	0.843

One more insightful consideration regards the averages of the results, reported in the last row of Table 5 and Table 7. The DL models seem to be more robust in the results with regard to the different features set, and achieve higher average results than the ML models. Probably, the higher complexity of the DL models architecture is able to better generalize the problem and then produce acceptable results with any kind of features. On the other hand, the ML models needs a feature set properly descriptive of the problem (i.e. colour-feature for our images) to produce the best results, but then are able to produce higher absolute value in performance. As a matter of fact, the Random Forest applied on the Autocolor Correogram and Simple Colo filter achieved the highest value in test, 0.894 in accuracy.

6 Conclusion and Future Work

Currently malware detection represents an open issue, considering the inability of signature-based free and commercial antimalware to detect new malicious payload and the ability of malware writers to

develop malicious code. To mitigate the spread of malicious code, with particular regard to Android environment, in this paper we propose a technique to automatically detect whether an application under analysis is malicious or legitimate. A dynamic analysis is considered i.e., we execute the application under analysis to extract the system call trace, that is used to generate an image of the application execution. Thus, once an image is obtained from each application, several machine and deep learning supervised classification algorithms are exploited to evaluate if this representation can be useful to discriminate between legitimate and malware Android applications. The experimental analysis, conducted on a real-world dataset of (legitimate and malware) 6817 Android applications show that all the considered classification obtained an accuracy up to 0.89, confirming the effectiveness of the proposed method in Android malware detection.

As future work, we plan to explore if the proposed representation of an Android application in terms of image can be helpful to identify also the belonging malware family. The experimental results are promising, the approach seems feasible for a quick preliminary evaluation on semi-realtime data, but needs to be improved to achieve higher results. It could benefit to insert this approach in a multi-level classification architecture, where the most interesting applications are filtered by our methodology; then, a more expensive but precise evaluation perform a static final analysis only on the subset of pre-selected applications. Furthermore, we will explore whether formal verification techniques can help us to reach a better accuracy in the malware detection task.

Acknowledgements

This work has been partially supported by MIUR - SecureOpenNets, EU SPARTA, CyberSANE and E-CORRIDOR projects.

References

[1] P. Agrawal and B. Trivedi. Machine learning classifiers for android malware detection. *Data Management, Analytics and Innovation*, pages 311–322, April 2020.

[2] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. Dl-droid: Deep learning based android malware detection using real devices. *Computers & Security*, 89:101663, February 2020.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of the 2014 Network and Distributed System Security Symposium (NDSS'14), San Diego, California, USA*, February 2014.

[4] M. B. Bahador, M. Abadi, and A. Tajoddin. Hlmd: a signature-based approach to hardware-level behavioral malware detection and classification. *The Journal of Supercomputing*, 75(8):5551–5582, March 2019.

[5] T. Bhatia and R. Kaushal. Malware detection in android based on dynamic analysis. In *Proc. of the 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security'17), London, UK*, pages 1–6. IEEE, June 2017.

[6] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting android malware using sequences of system calls. In *Proc. of the 3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile'15), Bergamo, Italy*, pages 13–20. ACM, August 2015.

[7] R. Casolare, C. De Dominicis, F. Martinelli, F. Mercaldo, and A. Santone. Visualdroid: automatic triage and detection of android repackaged applications. In *Proc. of the 15th International Conference on Availability, Reliability and Security (ARES'20), Virtual Event, Ireland*, pages 1–7. ACM, August 2020.

[8] R. Casolare, F. Martinelli, F. Mercaldo, and A. Santone. Detecting colluding inter-app communication in mobile environment. *Applied Sciences*, page 8351.

[9] A. Druffel and K. Heid. Davinci: Android app analysis beyond frida via dynamic system call instrumentation. In *Proc. of the 2020 International Conference on Applied Cryptography and Network Security (ACNS'20), Rome, Italy*, volume 12418 of *Lecture Notes in Computer Science*, pages 473–489. Springer, October 2020.

[10] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, July 1998.

[11] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone. Image-based malware family detection: An assessment between feature extraction and classification techniques. In *Proc. of the 5th International Conference on Internet of Things, Big Data and Security (IoTBDS'20), Prague, Czech Republic*, pages 499–506. Science and Technology Publication, May 2020.

[12] M. Jaiswal, Y. Malik, and F. Jaafar. Android gaming malware detection using system call analysis. In *Proc. of the 6th International Symposium on Digital Forensic and Security (ISDFS'18), Antalya, Turkey*, pages 1–5. IEEE, March 2018.

[13] M. Jerbi, Z. C. Dagdia, S. Bechikh, and L. B. Said. On the use of artificial malicious patterns for android malware detection. *Computers & Security*, 92:101743, May 2020.

[14] X. Jiang and Y. Zhou. *Android Malware*. Springer Publishing Company, Incorporated, 2013.

[15] E. Kasutani and A. Yamada. The mpeg-7 color layout descriptor: a compact image feature description for high-speed image/video segment retrieval. In *Proc. of the 2001 International Conference on Image Processing (ICIP'01), Thessaloniki, Greece*, pages 674–677. IEEE, August 2001.

[16] R. Kumar, X. Zhang, W. Wang, R. U. Khan, J. Kumar, and A. Sharif. A multimodal malware detection technique for android iot devices using various features. *IEEE access*, 7:64411–64430, March.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[18] A. Liaw, M. Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, December 2002.

[19] S. Malik and K. Khatter. System call analysis of android malware families. *Indian Journal of Science and Technology*, 9:1–13.

[20] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al. Deep android malware detection. In *Proc. of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY'17), Scottsdale, Arizona, USA*, pages 301–308. ACM, March 2017.

[21] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio. Download malware? no, thanks: how formal methods can block update attacks. In *Proc. of the 2016 IEEE/ACM 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE'16), Austin, Texas, USA*, pages 22–28. IEEE, May 2016.

[22] S. Michael, E. Florian, S. Thomas, C. F. Felix, and J. Hoffmann. Mobilesandbox: Looking deeper into android applications. In *Proc. of the 28th International ACM Symposium on Applied Computing (SAC'13), Coimbra, Portugal*, pages 1808–1815. ACM, March 2013.

[23] J. R. Movellan. Tutorial on gabor filters. *Open Source Document*, 40:1–23, 2002.

[24] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: visualization and automatic classification. In *Proc. of the 8th international symposium on visualization for cyber security (VizSec'11), New York, New York, USA*, pages 1–7. ACM, July 2011.

[25] S. Ni, Q. Qian, and R. Zhang. Malware identification using visualization images and deep learning. *Computers & Security*, 77:871–885, August 2018.

[26] K. Nirmala and A. Subramani. Content based image retrieval system using auto color correlogram. *Journal of Computer Applications*, 6(4):2013, March 2013.

[27] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, May 2001.

[28] R. Surendran, T. Thomas, and S. Emmanuel. Gsdroid: Graph signal based compact feature representation for android malware detection. *Expert Systems with Applications*, 159:113581, November 2020.

[29] R. Surendran, T. Thomas, and S. Emmanuel. A tan based hybrid model for android malware detection. *Journal of Information Security and Applications*, 54:102483, October 2020.

[30] R. Thangavelooa, W. W. Jinga, C. K. Lenga, and J. Abdullaha. Datdroid: Dynamic analysis technique in android malware detection. *International Journal on Advanced Science, Engineering and Information Technology*, 10(2):536–541, March 2020.

[31] Y. Tsuchiyama and K. Matsushima. Full-color large-scaled computer-generated holograms using rgb color

filters. *Optics express*, 25(3):2016–2030, 2017.

[32] A. Ukil. Support vector machine. In *Intelligent Systems and Signal Processing in Power Engineering*, pages 161–226. Springer, 2007.

[33] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang. Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences. *IET Information Security*, 11(1):8–15, March 2016.

[34] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the 33rd IEEE Symposium on Security and Privacy (S&P'12), San Francisco, California, USA*, pages 95–109. IEEE, May 2012.

Author Biography

Rosangela Casolare is a PhD student at the University of Molise. She has obtained a PhD scholarship co-financed by the Institute of Informatics and Telematics of the National Research Council (CNR) of Pisa. On February 21, 2019, she graduated with a Master's Degree in Software Systems Security at the same university. Her research activities concern the field of security of mobile systems, with particular emphasis on the development of methodologies for the detection of colluding applications through formal verification techniques in the Android environment.

Carlo De Dominicis is a master student in Data Science at University of Padua who is specializing in Machine Learning for Intelligent Systems. On April 2019 he graduated in Computer Science at University of Molise. Experienced some research activities, Carlo's intersests turned to continue his studies in order to apply them in the innovation field and modern issues.

Giacomo Iadarola is a Research fellow at the Institute of Informatics and Telematics (IIT) of the National Research Council of Italy (CNR), and PhD student at the Informatics Department of the University of Pisa. He got a Master's Degree in Computer Science, "Security and Privacy" specialization, at the University of Darmstadt and the University of Twente, in 2018. His research is mainly focused on Malware and Software Analysis, adopting Deep Learning and Formal Method models. Also, he collaborates, as a full-stack developer, with industries and other institutions on the European projects on which his research group is involved.

Fabio Martinelli is Head of Research at Institute of Informatics and Telematics (IIT) of the Italian National Research Council (CNR) where he leads the Trustworthy and Secure Future Internet group. He is co-author of more than two hundreds of papers on international journals and conference/workshop proceedings. His main research interests involve security and privacy in distributed and mobile systems and foundations of security and trust.

Francesco Mercaldo obtained his Ph.D. in 2015 with a dissertation on malware analysis using machine learning techniques. The research areas of Francesco include software testing, verification, and validation, with the emphasis on malware analysis on mobile systems. He worked as post-doctoral researcher at the Institute of Informatics and Telematics (IIT) of the National Research Council of Italy (CNR) in Pisa (Italy). Currently he is a Researcher at the University of Molise, Campobasso (Italy).

Antonella Santone received the Laurea degree in computer science from the University of Pisa, Italy, in April 1993, and the Ph.D. degree in computer systems engineering from the Dipartimento di Ingegneria della Informazione, University of Pisa, in September 1997. She is currently an Associate Professor of computer engineering with the University of Molise, since September 2017. Her research interests include formal description techniques, temporal logic, concurrent and distributed systems modeling, heuristic search, and formal methods in medical imaging.