MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention

Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli

Abstract—Android users are constantly threatened by an increasing number of malicious applications (apps), generically called malware. Malware constitutes a serious threat to user privacy, money, device and file integrity. In this paper we note that, by studying their actions, we can classify malware into a small number of behavioral classes, each of which performs a limited set of misbehaviors that characterize them. These misbehaviors can be defined by monitoring features belonging to different Android levels. In this paper we present MADAM, a novel host-based malware detection system for Android devices which simultaneously analyzes and correlates features at four levels: kernel, application, user and package, to detect and stop malicious behaviors. MADAM has been specifically designed to take into account those behaviors that are characteristics of almost every real malware which can be found in the wild. MADAM detects and effectively blocks more than 96 percent of malicious apps, which come from three large datasets with about 2,800 apps, by exploiting the cooperation of two parallel classifiers and a behavioral signature-based detector. Extensive experiments, which also includes the analysis of a testbed of 9,804 genuine apps, have been conducted to show the low false alarm rate, the negligible performance overhead and limited battery consumption.

Index Terms—Android security, intrusion detection system, malware, classification

1 Introduction

Smartphones and tablets have become extremely popular in the last years. At the end of 2014, the number of active mobile devices worldwide was almost 7 billions, and in developed nations the ratio between mobile devices and people is estimated as 120.8 percent [1]. Given their large distribution, and also their capabilities, in the last two years mobile devices have became the main target for attackers [2]. Android, the open source operative system (OS) introduced by Google, has currently the largest market share [1], which is greater than 80 percent. Due to the openness and popularity, Android is the main target of attacks against mobile devices (98.5 percent), with more than 1 million of malicious apps currently available in the wild [3].

Malicious apps (generically called *malware*) constitute the main vector for security attacks against mobile devices. Disguised as normal and useful apps, they hide treacherous code which performs actions in the background that threatens the user privacy, the device integrity, or even user's credit. Some common examples of attacks performed by Android malicious apps are stealing contacts, login credentials, text messages, or maliciously subscribing the user to costly premium services. Furthermore, all these misbehaviors can be performed on Android devices without the user noticing them (or when it is too late). It has been recently reported[1] that almost 60 percent of existing malware send stealthy premium-rate SMS messages. Most of these behaviors are exhibited by a category of apps called *Trojanized* that can be found in online marketplaces not controlled by Google. However, also *Google Play*, the official market for Android apps, has hosted apps which have been found to be malicious.[2]

Along with the vast increase of Android malware, several security solutions have been proposed by the research community, spanning from static or dynamic analysis of apps [4], to applying security policies enforcing data security [5], [6], to run-time enforcement [7], [8]. However, these solutions still present significant drawbacks. In particular, they are attack-specific, i.e., they usually focus on and tackle a single kind of security attack, e.g., privacy leaking [7], [8], or privilege escalation (jail-breaking) [5], [9]. Moreover, these frameworks generally require a custom OS [8]. Apart from these *ad hoc* security solutions, in an attempt to limit the set of (dangerous) operations that an app can perform, Android has introduced its native security mechanisms in the form of *permissions* and apps *isolation*. These two mechanisms, respectively, enforce access control to security critical resources and operations, and avoid that an app can interfere with the execution of another one. However, both permissions and isolation mechanisms have shown weaknesses [10].

In this paper we present a novel *multi-level* and *behavior-based*, malware detector for Android devices called *MADAM* (Multi-Level Anomaly Detector for Android Malware). In

- *A. Saracino and F. Martinelli are with the Instituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy. E-mail: {andrea.saracino, Fabio.Martinelli}@iit.cnr.it.*
- *D. Sgandurra is with the Department of Computing, Imperial College of London, London SW7 2AZ, United Kingdom. E-mail: d.sgandurra@imperial.ac.uk.*
- *G. Dini is with the Department of Ingegneria dell'Informazione, Università di Pisa, Pisa56100, Italy. E-mail: gianluca.dini@iet.unipi.it.*

Manuscript received 13 May 2015; revised 29 Jan. 2016; accepted 14 Feb. 2016. Date of publication 1 Mar. 2016; date of current version 12 Jan. 2018. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2016.2536605

1. http://usa.kaspersky.com/about-us/press-center/press-releases/kaspersky-lab-and-interpol-survey-reports-60-percent-android-at.
2. http://www.symantec.com/connect/blogs/yet-another-bunch-malicious-apps-found-google-play.

particular, to detect app misbehaviors, MADAM monitors the device actions, its interaction with the user and the running apps, by retrieving five groups of features at four different levels of abstraction, namely the kernel-level, application-level, user-level and package-level. For some groups of features MADAM applies an anomaly-based approach, for other groups it implements a signature-based approach that considers behavioral patterns that we have derived from known malware misbehaviors. In fact, MADAM has been designed to detect malicious behavioral-patterns extracted from several categories of malware. This multi-level behavioral analysis allows MADAM to detect misbehaviors typical of almost all malware which can be found in the wild. MADAM also has shown efficient detection capabilities as it introduces an 1.4 percent performance overhead and a 4 percent battery depletion. Finally, MADAM is usable because it both requires little-to-none user interaction and does not impact the user experience due to its efficiency.

MADAM achieves the above goals as follows: (i) it monitors five groups of Android features, among which system calls (type and amount) globally issued on the device, the security relevant API calls, and the user activity, to detect unusual user and device behavioral patterns; to this end, it exploits two cooperating proximity-based classifiers to detect and alert anomalies; (ii) it intercepts and blocks dangerous actions by detecting specific behavioral patterns which take into account a set of known security hazard for the user and the device; (iii) every time a new app is installed, MADAM assesses its security risk by analyzing the requested permissions and reputation metadata, such as user scores and download number, and it inserts the app in a suspicious list if evaluated as risky.

1.1 Paper Contributions

The contributions of this paper are the following:

- We discuss in detail MADAM, a behavior-based and multi-level malware detection systems for Android devices. MADAM detects misbehaviors of Android apps, stops them and prevents further malicious actions by removing the responsible app. MADAM combines several classes of features, from distinct Android levels, and applies both anomaly-based and signature based mechanisms.
- We have evaluated the effectiveness of MADAM against three datasets of malicious apps, namely the Genome [11], Contagio-Mobile [12], and VirusShare datasets. Experiments reported a detection accuracy of 96.9 percent on a testbed of 2,784 malicious apps, divided in 125 families, spanning from 2010 to 2015. Results have been compared with the VirusTotal [13] tool, showing comparable accuracy. MADAM has also been able to detect 9 malware families which evade VirusTotal checks, in particular the zero-day attack *Poder* [14].
- Extensive tests have been run to assess the false alarm rate in different real usage conditions. In particular, MADAM shows a False Positive Rate (FPR) of $2.8 \cdot 10^{-5}$ in normal usage conditions, on a week of normal device usage. Additional usability tests, aimed at evaluating the MADAM FPR, have been performed through the analysis of a set of 9,804 genuine apps. Only 22 apps (0.2 percent) have been considered a potential threat at install-time, none of which was considered harmful during run-time detection. We have also measured the performance overhead of MADAM, to estimate the impact on the user experience and on the battery duration. Experiments conducted through standard software tools reported a global performance overhead of 1.4 percent and of 4 percent on the battery duration.
- Furthermore, we propose a behavior-based taxonomy of existing Android pieces of malware into seven classes, used to derive the common patterns of misbehaviors across the same class.

The main novelty of MADAM is its cross-layer approach, and a novel integration of techniques (some of which already existing) that provides high efficacy with low overhead. MADAM has been conceived to prove that a multi-level approach makes it possible to dynamically detect most of current Android malware, right on the device with limited overhead. To verify that such approach is indeed viable, a large extensive set of tests have been performed to prove empirically its efficacy.

The rest of the paper is organized as follows. In Section 2 we propose a new malware classification, which is used by MADAM to help the detection of known and anomalous behaviors. We also describe the multi-level approach of MADAM and the Android features that are relevant to detect misbehaviors from each specific class. In Section 3 we thoroughly describe all the components of MADAM architecture, while in Section 4 we detail the algorithm to detect malware by correlating features with misbehaviors. Section 5 discusses the detection results on three datasets including about 2,800 real malicious apps, by also presenting the experimental results on false alarm rate and performance overhead. Section 6 describes the relevant works in related field, which are compared with MADAM. Finally, Section 7 concludes the paper.

2 MADAM Approach to Malware Detection

In this section we propose a taxonomy of Android malware, which is used by MADAM to detect, identify and stop generic classes of misbehaviors.

2.1 MADAM Malware Behavioral Classes

The amount of malicious Android apps and malware families is continuously increasing. In fact, according to [3], more than ten millions of malicious apps for Android were available at the end of 2013. More recently, a report for the first half of 2014 [15] presents an increase consisting of 20 new malware families. Notwithstanding the huge number of malicious apps and threats, Android malware can be grouped into a more limited and manageable number of classes, representative of a specific (malicious) behavior. For these reasons, we propose the following behavior-based malware taxonomy (*behavioral classes* of malware)[3]:

3. Research proof-of-concepts are not considered in this classification.

TABLE 1
MADAM Levels of Analysis and Features

Level	Group	Feature	Description	Targeted Misbehavior
Kernel	Sys Calls	`open`, `read`, ...	System calls concerning file and inter-component communication	Sudden and unmotivated activity increase
Application	SMS	Number of SMS (SMS Num)	Amount and recipient of outgoing SMS	Unsolicited outgoing messages
Application	SMS	Suspicious SMS (SMS Susp)	Amount of SMS sent to recipients not in contacts	Spyware or registration to premium services
Application	Critical API	Administrator App	Verify if an app attempts to get admin privileges	Apps which attempt to take control of the device
Application	Critical API	New App Installation	Verify if an app attempts to install a new one	Unauthorized app installations
Application	Critical API	Process List	Verify if an app generates high number of processes	Buffer overflow (Rootkit) attacks
Application	Critical API	Critical SysCalls	Amount of critical system calls generated by an app	Apps that access files and resources in backround (Spyware, Botnet and Trojan)
Application	Critical API	SMS Default App	Check default SMS manager	Unsolicited outgoing SMS
Application	Critical API	Foreground App	Check which app is interacting with user	Unsolicited SMS and preventing user from interacting with the device (Ransomware)
User	User Activity	User Presence	If the user is interacting with the device	Unsolicited activities of Spyware, Botnet, Installer and Rootkit
User	User Activity	On Call	Verify if a phone call is ongoing	Unsolicited activities of Spyware, Botnet, Installer and Rootkit
User	User Activity	Screen On	Verify if the device screen is on	Unsolicited activities of Spyware, Botnet, Installer and Rootkit
Package	App Metadata	Permissions requested (`manifest.xml`)	Riskiness of app	Suspicious requests of dangerous permissions
Package	App Metadata	Market Info (User scores, ...)	Popularity of app	Trojan

1) *Botnet*: malware that open a backdoor on the device, waiting for commands which can arrive from an external server or an SMS message.
2) *Rootkit*: malware that perform buffer overflow to get super user (root) privileges on the device.
3) *SMS Trojan*: malware that: (i) send SMS messages stealthily and without the user consent, generally to subscribe the user to a premium services, or send spam messages to all of the user contacts; (ii) exploit authentication mechanism of some bank institutes, based on SMS messages, to authorize unwanted transactions (Banking Trojan).
4) *Spyware*: malware that take pieces of private data from the mobile device, such as IMEI and IMSI, contacts, messages or social network account credentials; an important sub-class is the malware that take pieces of data from location interfaces and send them to an external server without the user consent.
5) *Installer*: malware that install apps with new authorizations to increase the capability of harming the system.
6) *Ransomware*: malware that prevent the user from interacting with the device, by continuously showing a web page asking the user to pay a ransom to remove the malware, or that encrypt personal files asking a ransom to retrieve the decryption key.
7) *Trojan*: any malware whose behavior is not considered by the previous classes, such as those that modify or delete data from the device without the user consent or that infect personal computers when the device is connected via USB.

Note that some malicious apps may fall in more than one of these classes. For example, some apps include *Rootkits* and also *Botnet* functionality, or have the ability to install new apps and to send SMS messages. These eight classes have been introduced to allow MADAM to categorize different malware misbehaviors effectively. In fact, each malware class performs a set of misbehaviors that can be identified by correlating the analysis of specific features coming from different levels. This approach is detailed in the following.

2.2 Multi-Level Behavior-Based Feature Analysis

As we have previously underlined, MADAM needs to analyze different kinds of features, at different levels, to detect the misbehaviors of the malware behavioral classes. For this reason, MADAM monitors the Android system at four levels, and retrieves five groups of features (four at run-time and one statically).

An exhaustive list of the monitored features is reported in Table 1. The table reports, for each feature, the corresponding Android level, the group of features, its description, and the misbehavior(s) for which the feature is used. In particular, at the first level (*level I*, *kernel-level*), MADAM monitors the issued *system calls*, which describe the device behavior at the lowest level. In fact, any action performed by apps, or by the Android framework, is eventually translated into a sequence of system calls. Hence, MADAM intercepts system calls that are considered critical from the point of view of security, in particular those related to file operations. The rationale behind this approach is that MADAM, using these features, detects misbehaviors related to sudden increase of the system call occurrences, especially if not motivated by the user activity. These misbehavior are relevant to detect *Rootkits* attempting to perform a buffer overflow attack, and *Trojans* accessing user files. MADAM also monitors security relevant *critical API* (*level II*, *application-level*) which are invoked by the framework or apps, to perform operations which might be critical on the security side. Namely, the monitored actions are: (i) installing a new app, (ii) requesting administrator privileges (used by malware to acquire a greater control on device and avoid removal), (iii)

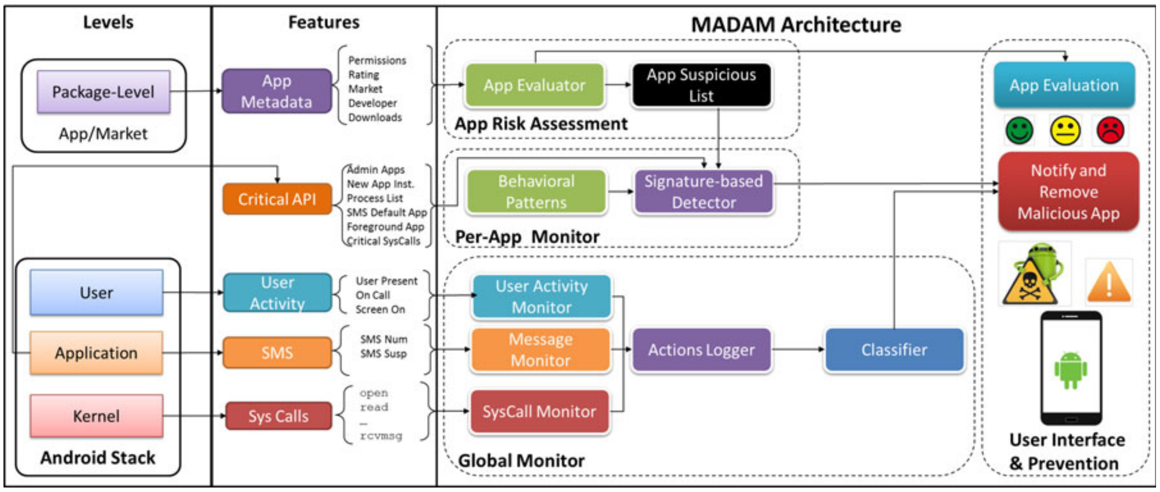

Fig. 1. Architecture of MADAM.

generating too many processes, (iv) constant monitoring on the app that is in foreground (i.e., the app that is actively interacting with the user) Monitoring these critical API calls is relevant for detecting misbehaviors related to *Ransomware*, *Spyware*, *Botnet* and *Installer* malware. Another relevant group of features at this level monitored by MADAM is the *SMS*. In fact, more than 90 percent of the pieces of malware found in the wild [3] is related to an improper usage of the SMS functionality, which also imposes a direct financial cost to the user. The user contact list is also monitored, to detect behavioral patterns (detailed in Section 3.3) of *SMS Trojans*.

By monitoring the *user activity* (*level III*, *user-level*) MADAM is able to understand when the user is (or is not) interacting with the device, relating these two behavioral profiles (active or idle) with the device activity. In fact, when the user is interacting with the phone, the number of events (both at kernel and application-level) monitored on the device is generally higher than when the user is not interacting. This group of features is directly correlated with the user activity and SMS features to detect misbehaviors related to *Spyware* and *Botnet*, i.e., messages sent when the user is not active, and *Installers*, i.e., malware installing apps being without user interaction. Finally, regarding the static analysis, MADAM analyzes static *app metadata* (*level IV*, *package-level*), namely the permissions declared by apps and reputation metadata, such as rating, marketplace and download number. These features are used by MADAM for performing a preliminary risk assessment of the app at deploy-time. This first evaluation of an app will then be used by the run-time MADAM framework to determine which apps need to be monitored for known patterns of misbehaviors. Note that MADAM, as detailed in Section 3.2, performs also a global monitoring of the Android system which considers the actions executed by all apps.

By correlating through the classifiers a subset of these features, MADAM is able to discern between normal and anomalous device behavior and to find the malicious actions performed by apps. On the other hand, a further subset of features is exploited by MADAM to detect known patterns of misbehaviors. The detection algorithm will be described in Section 4. Instead, in the next section, we detail the MADAM architecture, its components, their interaction and how MADAM collects, analyzes and correlates the features at four levels

3 MADAM Architecture

To derive the features at the four system levels, and to detect and prevent a misbehavior, MADAM can be logically decomposed into four main architectural blocks, which are depicted in Fig. 1 (in particular, see "Madam Architecture"). The first one is the *App Risk Assessment*, which includes the *App Evaluator* that implements an analysis of metadata of an app package (apk) (permission and market data), before the app is installed on the device. This evaluation computes the app's *risk score*, i.e., the likelihood that the app is a malware. Based on this risk evaluation, this component populates a set of suspicious apps (*App Suspicious List*), which will be monitored at run-time. The second block is the *Global Monitor*, which monitors the device and OS features at three levels, i.e., kernel (*SysCall Monitor*), user (*User Activity Monitor*) and application (*Message Monitor*). These features are monitored regardless of the specific app or system components generating them, and are used to shape the current behavior of the device itself. Then, these behaviors are classified as *genuine* (normal) or *malicious* (anomalous) by the *Classifier* component. The third block is the *Per-App Monitor*, which implements a set of *known behavioral patterns* to monitor the actions performed by the set of suspicious apps (*App Suspicious List*), generated by the *App Risk Assessment*, through the *Signature-Based Detector*. Finally, the *User Interface & Prevention* component includes the *Prevention* module, which stops malicious actions and, in case a malware is found, handles the procedure for removing malicious apps using the User Interface (UI). The UI handles notifications to device user, in particular: (i) the evaluation of the risk score of newly-downloaded apps by the *App Evaluation*, (ii) the reporting of malicious app (*Notify*) and (iii) to ask the user whether to remove them (*Remove Malicious App*). We now describe each architectural block in detail.

3.1 App Risk Assessment

When a new app is installed on the device (deploy-time), the *App Evaluator* component intercepts and hijacks the installation event. This component analyzes the metadata of

TABLE 2
Comparison of Behavior Vectors: User Idle (Top) vs User Active (Bottom)

open	ioctl	brk	read	write	exit	close	sendto	sendmsg	recvfrom	recvmsg	Idleness	SMS Num	SMS Susp
6	19	18	1	4	0	7	16	2	2	0	0	0	0
147	652	192	711	4	282	229	7	15	7	13	1	0	0

the new app to assess its risk, by retrieving features from the app package, related to critical operations, and from the market, related to app reputation. In detail, these features are: (i) the *permissions* declared in the manifest, (ii) the *market* of provenance, (iii) the total number of *downloads*, (iv) the developer *reputation* and (v) the *user rating*. The five parameters are analyzed through a hierarchical algorithm which returns a decision on the riskiness of the app classifying it as *safe* or *risky*.[4] Based on this decision, the user can choose whether to continue the installation (or not) of the new app. If the user chooses to install a risky app, its package name is recorded in the MADAM *App Suspicious List* and is continuously monitored looking for the known behavioral patterns. Note that MADAM extracts all these pieces of information in a process which is totally transparent to the user. The user can, however, decide whether she prefers to receive a notification of the decision of the *App Evaluator*, or to keep the process invisible. In the following, we assume that the user chooses the transparent approach (i.e., new apps are always installed, but inserted into the *App Suspicious List* if risky), as to allow MADAM to enforce security policies on the device. It is worth noting that the *App Evaluator* is not a detector of malicious apps. Instead, the *App Evaluator* aims at finding apps which are risky, which should be monitored at run-time by MADAM, improving the overall performance.

3.2 Global Monitor

The *Global Monitor* is at the core of the MADAM framework, since it is responsible of collecting the run-time device behaviors and classifying them as “genuine” or “malicious”. In MADAM, a *behavior* is represented through a vector of features. For each of them, MADAM records how many times a specific feature has been used in a period of time T_k. The features are extracted from different kinds of dynamic events (refer to Fig. 1, “Features”): *User Activity*, *Critical API* (in particular, *SMS*, i.e., text messages) and *System Call* (Sys Calls). The *Actions Logger* is the component that records all these features into a vector, which is then fed to the *Classifier*. This component is trained to recognize genuine behaviors related to normal device usage, and malicious behavioral patterns deviating from the genuine ones, derived from the seven classes of malware. The classifier correlates features from the three monitored levels, and detects misbehaviors which could pass unnoticed if monitored separately on the single levels. As we will detail in Section 5, the Global Monitor is effective in detecting malicious behaviors, especially for *SMS Trojan*, *Rootkit*, *Installers* and *Ransomware*. For other behavioral classes of malware, MADAM exploits a set of known malicious behavioral patterns, discussed in Section 3.3. We now describe each component of the *Global Monitor* in detail.

3.2.1 System Call Monitor

The *System Call Monitor* is used by the *Global Monitor* to intercept the system calls reported in Table 2 (first eleven columns). These system calls are related to file operations and network access. We have chosen to intercept these calls because we have observed that the greatest amount of operations issued on Android, and consequently by malware, are translated as operations on files at low-level. To intercept system calls on Android, MADAM exploits a kernel module to hook the critical system calls through system call table overriding. The kernel module is loaded through the `insmod` command and interacts with the rest of the MADAM framework through a shared buffer.

3.2.2 User Activity and Message Monitor

The *User Activity* and *Message Monitor* allow MADAM to intercept calls to security relevant API functions, namely related to SMS messages and user activity. As we have previously recalled, these features are critical from a security point of view to detect SMS sent to premium-numbers and/or without the user knowledge. MADAM hijacks security relevant methods, by monitoring their actual parameters and controlling the final outcome of the action. In particular, MADAM hijacks the `SendTextMessage()` and `SendDataMessage()` methods to control the events of outgoing SMS messages.[5] Furthermore, using standard Android APIs, MADAM also verifies (i) if the user is interacting with the device, (ii) if the device screen is on/off and (iii) if a phone call is ongoing. These elements are used to assess when the user is *active*. In particular, we can categorize the status of the user as being in one of two possible states (*active* or *idle*), which are strongly dependent on the activity of the phone itself. In the first user activity state (*active*) either (i) the user is actively interacting with the phone and the screen is on, or (ii) the screen is off but a phone call is ongoing. In fact, when the user is active, the phone has to show interactive contents on the screen and receives inputs from the user, or handles the elements involved in a phone call. Otherwise, in the second user activity state (*idle*), the phone is not active. As far as concerns the monitored features by MADAM, we note that in the *active* state a large amount of system calls is generated. On the contrary, in the *idle* state, a low number of system call is generated.

3.2.3 Action Logger and Classifier

Among all the features that MADAM collects, the *Action Logger* retrieves 14 features from three classes at three

4. We refer the reader to [16] for the full description of the algorithm, which we have implemented in MADAM with the *App Evaluator*.

5. MADAM exploits the *XPosed Framework* [17].

distinct levels (kernel, application, user). In detail, the first eleven features concern the system calls related to file modification and inter-component communication (i.e., `open`, `ioctl`, `brk`, `read`, `write`, `exit`, `close`, `sendto`, `sendmsg`, `recvfrom`, and `recvmsg`). To this end, MADAM implements a time-based representation of the *bag of system calls* model [18], in which the monitored system calls are represented through a vector $[f_1, \dots, f_{11}]$, and where each f_j is the number of occurrences of system call f_j globally issued on the device during each interval T_k. The 12th feature represents the user activity (idleness), being 1 if the user is active and 0 otherwise. The feature f_{13} records the amount of outgoing SMS messages sent every T_k sec. Finally, the feature f_{14} (*SMS Susp*) represents the amount of text messages sent to a recipient which is not in the device contact list. These features are then used to create the vectors for the two classifiers. As a clarifying example, we have reported in Table 2 two behavior vectors taking into account the two user profiles (active and idle).

The feature vector is then fed to two parallel classifier. The first instance is a *short-term* classifier with a period $T_k = T_{short}$ sec, whereas the second one constitutes a *long-term* monitor with a period $T_k = T_{long}$ sec (both values are configurable at run-time). The cooperation of these two instances detects different types of misbehaviors, i.e., different classes of malware. In particular, the short-term monitor is more effective in detecting "spiky" misbehaviors, i.e., with sudden, brief and sharp increase of the system call occurrences (typical of some *Rootkits*, for instance). On the other hand, the long-term monitor is aimed at detecting misbehaviors that distribute their actions constantly in a long period of time, such as *Spyware*, i.e., whose effect is not immediate.

Both classifiers have been trained to recognize a list (white-list) of real genuine behaviors collected on a real device (Samsung Galaxy Nexus with Android 4.3) not infected by any malware. These components act as an anomaly-based intrusion detection system (IDS) that, by definition, alerts as anomalies the behaviors which appreciably differs from the known ones. However, if a classifier is only trained to recognize genuine behaviors, it will never report any alert. For this reason, the MADAM classifiers have also been trained with a set of synthetic (malicious) behaviors which are appreciably different from the set of genuine ones, but are not related to a specific malware. The approach allows MADAM to detect also those classes of malware that are not associated with a specific signature.

The usage of computational intelligence (classifiers) and statistical techniques for intrusion detection is a well-known approach already exploited on other OSes and for network analysis [19], [20]. In fact, regardless of the specific environment, intrusion detection can be modeled as a problem of *binary classification*. The binary classification problem for intrusion detection can be formally defined as follows. Let us consider the set of classes $\Omega = \{\omega_0, \omega_1\}$ where ω_0 is the class of good behaviors and ω_1 is the class of malicious behaviors. Then, given a behavior $B \in \mathbb{R}^m$, $B = \{f_1, \dots, f_m\}$, where m is the total number of features describing the behavior, the goal of the classifier is to assign to B the correct class in Ω, through a function $D : \mathbb{R}^m \rightarrow \Omega$ named classifier.

Several classifiers can be used to solve the specific MADAM classification problem, in which each behavior vector B_i is composed of $m = 14$ features (reported in the heading row of Table 2). Among the classifiers, the K-NN (K-Nearest Neighbor) [21] is the one that best meets the MADAM needs for the following reasons: (i) all the m features describing a behavior in MADAM are numerical; (ii) since the rationale of MADAM's classification process is that an intrusion represents a consistent deviation from the device normal behavior, then by using numerical features it is possible to geometrically represent this difference. The K-NN classifier exploits this geometric representation to classify behaviors closer to genuine ones as belonging to class ω_0 and the behaviors closer to the malicious ones as belonging to class ω_1. Both the long-term and the short-term classifier are realized through a K-NN (K-Nearest Neighbor) classifier, which is a similarity-based classifier, i.e., it classifies two similar elements as belonging to the same class. The similarity measure used by the K-NN classifier is the euclidean distance measured in the features space, i.e., two elements are considered similar if geometrically close in the features space [21], [22]. This is computed as:

$$Similarity(x, y) = -\sqrt{\sum_{i=1}^{m} (x_i - y_i)^2},$$

where x_i and y_i are the features of the vectors x and y. If $K = 1$, then the case is simply assigned to the class of its nearest neighbor (the one with the largest similarity).

The similarity-based approach of the K-NN classifier is well suited with the rationale of the classification problem of MADAM. In fact, MADAM is trained with known behaviors, representing genuine device activities (white list), and artificially generated malicious behaviors, which are strongly different (distant) from the known ones. New behaviors that are similar to the known genuine behaviors are considered benign as well. On the other hand, new behaviors which are closer to the artificial behaviors, are classified as malicious. To further justify our classifier selection, we have considered other classifiers used for numerical (quantitative) features, namely Linear Discriminant Classifier (LDC), Quadratic Discriminant Classifier (QDC), Multi-Layer-Perceptron with back-propagation (MLP), Parzen Classifier (PARZC) and Radial Basis Function (RBF). All the classifiers have been trained with the same datasets and the same validation technique. The K-NN classifier gives the best classification results among all other classifiers, achieving the max accuracy when $k = 1$.

3.3 Per-App Monitor

The *Per-App Monitor* component is complementary to the *Global Monitor* since it is aimed at detecting additional, signature-based, known misbehaviors. The *Per-App monitor* is based on a set of known malicious *behavioral patterns* which considers the *Suspicious App List* created by the *App Risk Assessment* module, the alerts raised by the classifiers and a set of features at application-level not considered by the classifier. The *Per-App monitors* exploits behavioral patterns which represent suspicious behaviors that have been inferred by analyzing the behavioral classes of malware at API level and kernel level. To consider these behavioral patterns, *Per-App Monitor* constantly monitors three features, namely: (i) the list of apps with administrator privileges (*Admin Apps* in Fig. 1), which are those apps that can access a specific set of dangerous security relevant API and that cannot be removed unless the privileges are revoked, (ii) the *SMS default app*, which is the app that by default handles the operations related to text messages and that can be changed by the user, (iii) the *app in foreground*, which is the app currently interacting with the user.

3.3.1 Malicious Behavioral Patterns

We have defined seven malicious behavioral patterns, which are detected by the *Per-App Monitor*:

1) "*Text messages sent by a non-default message app*". This pattern aims to detect malware that stealthily send SMSs (i.e., SMS Trojans, Botnet, Spyware). MADAM considers as malicious any SMS sent by an app which is not the default messaging app.
2) "*Text messages sent to numbers not in the user contact list*". This behavioral pattern tackles malware that attempt to maliciously register to premium services via SMS (*SMS Trojan*), or send text to an hardcoded external number (*Spyware*, *Botnet*). When an app, different from the default messaging app, attempts to send a text message, MADAM hijacks the action and verifies if the recipient of the message is in the contact list. If the recipient is not in the contact list, the action is considered a misbehavior.
3) "*High number of outgoing message per period of time*". This behavioral pattern aims to detect *SMS Trojans* which send unsolicited messages to user contacts, e.g., for spam purpose or to drain the user credit. Outgoing messages are allowed till the ratio message per period does not reach a configurable threshold (default: 5 messages per minute). For higher values of this ratio, a misbehavior is detected.
4) "*High number of process per app*". This behavioral pattern targets malware that attempt to acquire root privileges, e.g., through buffer overflow (i.e., *Rootkit*). If the number of processes generated by an app is higher than a configurable threshold (default: 10 processes per app), the misbehavior is detected.
5) "*Excessive foreground time for non interacting and administrator app*". This behavioral pattern targets malware which attempt to take exclusive control of the device (i.e., *Ransomware*). If an app remains in foreground for a time longer than a threshold (default: 30 seconds), without user interaction (user not present), and the app owns administrator privileges, the misbehavior is detected.
6) "*Unauthorized installation of new apps*". This behavioral pattern tackles malware which attempt to install other apps on the device without the user authorization (i.e., *Installers*).
7) "*Unsolicited kernel level activity of background app*". This behavioral pattern is typical of *Botnet*, *Spyware* and some generic *Trojans*. If a suspicious app generates an `open`, `write` or `sendmsg` system call when is not in foreground, a misbehavior is detected.

Thresholds defined in the behavioral patterns are currently manually configured, though adding adaptive algorithm for self-tuning of these thresholds is scheduled as future work.

3.4 User Interface and Prevention

The *User Interface & Prevention* includes the *Prevention* module that acts as a security enforcement mechanism by blocking the detected misbehaviors related to behavioral patterns, e.g., a SMS being sent without the user authorization. In such a case, the *User Interface* (UI) module handles the process for removing the responsible app. The UI conveys to the user all the events which require an active interaction, such as for removing malicious apps, and is also used by the user to select which behaviors should be blocked or allowed. Finally, the UI is exploited by the *App Evaluator* to communicate to the user the risk score of a new app at deploy-time. In this case, the user can then decide whether to continue the installation (or not) of the app.

3.5 Deployment

MADAM comes as a package which contains the *MADAM apk* , implementing the *User Interface and Prevention Modules*, the *App Evaluator*, the *Per-App Monitor*, and *Global Monitor*. The MADAM package also contains the *Superuser*[6] *apk*, for handling attempts of accessing root privileges (see details in Section 5), and the *X-posed Installer apk*, for hooking and handling events relevant to the *Per-App Monitor*. When installed, MADAM deploys a kernel monitoring module, by issuing the `insmod` command. This command, and the X-posed Installer, require root access. For this reason, MADAM only runs on rooted devices with a kernel having module support. Due to these limitations, MADAM has not been conceived for distribution in the mass market. Rather, MADAM has been conceived to prove that a multi-level approach makes it possible to dynamically detect most of current Android malware, right on the device with limited overhead. These may constitute incentives for the OS manufacturer to integrate MADAM into the smartphone OS.

4 MADAM Detection Algorithm

In this section we detail how MADAM exploits the cooperation of the architectural components and the retrieved features to detect a misbehavior. In particular, the *Per-App Monitor* and *Global Monitor* exploit two distinct sets of features to detect different misbehaviors, i.e., known (mis) behavioral patterns and anomalies. Moreover, they also cooperate to identify more complex misbehaviors whose detection needs the analysis of features from both components (i.e., *Ransomware*). Once the misbehavior has been detected, security is enforced by the *User Interface & Prevention* module, which stops the misbehavior, notifies the user and removes the malicious app. Moreover the *App Evaluator* reduces the likelihood of false alarms, focusing the attention of MADAM on only those apps which effectively bring a risk.

When MADAM starts, the *App Evaluator* is launched in background waiting for new apps to be installed, assessing their risk at deploy-time to populate the *App Suspicious List*.[7]

6. http://play.google.com/store/apps/details?id=com.noshufou.android.su

7. No analysis is performed by the App Evaluator on those apps installed on the device before installing MADAM.

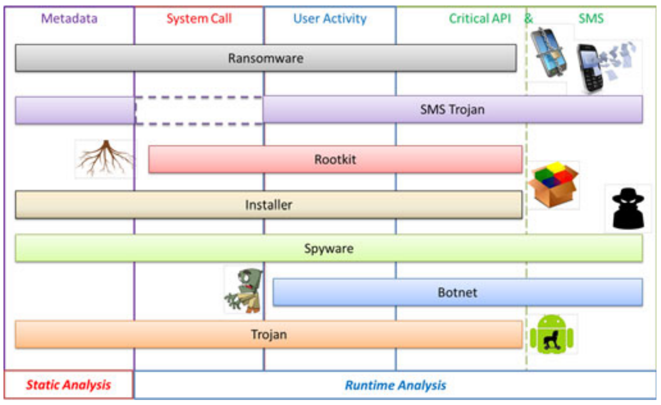

Fig. 2. Relevant features for the detection of the seven malware behavioral classes.

Afterward, the *Global Monitor* is launched in background, to retrieve the 14 features and classify the behaviors. This component either generates an alert or remains silent. In parallel, the *Per-App Monitor* is launched to monitor kernel and API features to detect and stop known behavioral patterns. To this end, two sets of behavioral patterns are checked continuously in background by the *Signature-based Detector*. The *Per-App Monitor* blocks the misbehavior by locating the responsible app in the Suspicious List. The *User Interface & Prevention* module kills the app deemed as responsible, proposing the user to remove it. The *User Interface & Prevention* considers as parameters the app suspicious list generated by the *App Evaluator*, the alarms generated by the *Global Monitor* and *Per-App* monitor. Before removal, the misbehaving app and the class of malware are communicated to the user, who takes the final decision on the app removal.

4.1 Correlating Features and Misbehaviors

We now detail how MADAM correlates the extracted features by the *App Classifier*, *Per-App Monitor* and *Global Monitor* to detect and tackle misbehaviors for each malware behavioral classes. Fig. 2 schematically summarizes the relations between the proposed malware behavioral classes and the groups of features used by MADAM to detect their misbehaviors. In detail, *Ransomware* are detected (i) by the *App Evaluator* by monitoring the requested dangerous permissions needed to take control of the device and (ii) by the *Global Monitor* by analyzing the anomalous increase in the occurrences of system calls that they cause to prevent the user-device interaction. Moreover, the *Ransomware* match the foreground behavioral pattern (the fifth pattern described in Section 3.3.1) verified by the *Per-App Monitor*. If the detection conditions are met, the *Ransomware* is removed by the *User Interface & Prevention Module* by deleting app files and executable (odex). *SMS Trojans* are pre-filtered by the *App Evaluator* due to the SMS-related permissions they require. If the SMS Trojan is installed, the *Global Monitor* detects the anomalous behavior of messages sent when the user is not active. In addition, the *Per-App Monitor* exploits the first three behavioral patterns (Section 3.3.1), typical of *SMS Trojans*. Also in this case, if the detection conditions are verified, the SMS is blocked the app is deemed as malicious and proposed for removal. Furthermore, the UI alerts the user of messages matching the behavioral pattern\s even if no alerts have been raised by the *Global Monitor*. Most recent Android versions (starting from Android 4.4) include the possibility to set a *default messaging app*, which has the exclusive access to SMS providers, responsible for writing and reading the Outbox and Inbox message folders. While such a feature is able to mitigate the action of several spyware, aimed at reading the messages, it is not effective against the majority of SMS Trojans, which can send text messages without leaving traces on the Outbox folder [23].

Rootkits are detected by the cooperation of the *Global Monitor*, which notifies sudden increase in the system calls amount, and the *Per-App Monitor*, which detects the app which matches the fourth known behavioral pattern (number of generated processes). If the *Global Monitor* raises an alert, and no app is matching the behavioral pattern, the alert is ignored. On the other hand, if no alert is raised by the *Global Monitor*, even if an app is generating a high number of process, MADAM can prevent the app from generating new processes. This decision is given to the user through the UI. *Installer* malware are detected through the *Per-App Monitor* which considers the unauthorized app installation as a malicious known behavioral pattern. An app is considered as not installed by the user when a new package is added (package_added intent) without receiving the user manual authorization. The action is blocked and notified to the user who can decide whether to continue the installation or remove the app deemed as malicious. The misbehaviors of *Installer* malware also affect system calls usage and raise an alert of the *Global Monitor*. *Spyware* apps have different misbehaviors depending on the specific implementation, which may affect all of the monitored features, spanning from suspicious permissions to activities (critical API or system calls) not related to user activity. *Spyware* are considered as risky by the *App Evaluator* since they include unmotivated requests of the networking and messaging permissions. Furthermore, the activity of sending information outside is detected by the *Per-App Monitor* through the seventh known behavioral pattern, and by detecting the activity change not related to user activity.

Using the same behavioral patterns used to detect *Spyware*, the *Per-App Monitor* also detects some *Botnet* and some *Trojan* malware. In particular the Trojan `Moghava`, which modifies the user pictures stored on SD Card, is detected by the *Global Monitor* due to the periodic system call activity, not related to user activity. On the other hand *Botnet* sending SMS with personal information, sent when they receive a command from the botmaster, are detected through the *Per-App Monitor* through the SMS-related known behavioral patterns. Moreover, the *Global Monitor* detects the event of outgoing message when the user is not active.

5 Results

This section presents and discusses the extensive set of experimental tests performed on the MADAM framework. A first set of experiments evaluates the accuracy of MADAM in real usage contexts. We report the detection results performed on two dataset of malicious apps, accounting more than 1,300 apps and 50 malware families. Note that when defining the malware behavioral classes, we did not consider these two datasets. Hence, the apps coming from these datasets were unknown to MADAM, and can be considered as zero-day malware. A further set of experiments has been conducted to assess the amount of false alarms generated in three different usage contexts. Finally, the impact of MADAM on performance and energy consumption (see Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/2536605) has also been evaluated through standard benchmarking apps.

5.1 Malware Detection Results

To verify the effectiveness of MADAM in detecting malware, we have extensively tested the framework against malware coming from four datasets. The whole testbed accounts about 2,800 applications coming from 125 different families. Namely the tested database are the following:

- *Genome*: Genome [11] is a collection of 1,242 malicious Android apps collected in 2010 and 2011. The apps are divided in 49 malware families that include almost all malware categories discussed in Section 2.2, namely *Botnet*, *Rootkit*, *SMS Trojans*, *Spyware*, *Installer* and *Trojan*. The vast majority of genome apps come from Chinese unofficial marketplaces.
- *Contagio Mobile*: Contagio Mobile is a blog-like website that collects malware for several mobile OSes. Differently from Genome, which has several samples for different pieces of malware, Contagio only presents few samples (generally one) for each malware family. Contagio collects malware since 2012, including also malicious apps found on Google Play, which may affect any kind of Android device, regardless of the nationality. We have tested MADAM against 18 malware families from the Contagio database.
- *VirusShare*: VirusShare is a website hosting a database of malware for several OSes, which also includes a good share of Android malware. The database is continuously updated and it is also possible to retrieve some of the latest threats discovered. We have used a set of 1,923 malicious apps divided in 90 families, with malware spanning from 2012 to 2015.

Experiments have all been performed on a jailbroken `Samsung Galaxy Nexus` running Android stock Jelly Bean (version 4.3). MADAM's detection results are reported in Tables 3 and 4. The two tables report, starting from the left column, the name of the analyzed malware families, the corresponding behavioral class discussed in the previous sections, the year of first retrieval according to different sources, e.g., [2], [3], [12], the number of analyzed samples for each family, and the number of samples detected respectively by MADAM[8] and by *VirusTotal*. VirusTotal [13] is a web-service for malware static analysis of files and URLs. When a file is submitted to VirusTotal, the web-service submits the file to be analyzed by 50-60 well known anti-virus software, and returns the results of the analysis for each anti-virus, i.e., either "clean" or the name of the threat/malware found. The *cached* mode of VirusTotal has been used for these experiments, i.e., only the hash of each app has

8. The MADAM column specifies if the application has been detected at run-time.

TABLE 3
Detection Results and Comparisons (Part. 1)

Malware	Class	Year	Samples	MADAM	VirusTotal
Lemon	Botnet	2012	6	0	6
MmarketPay	Botnet	2012	1	0	1
Basebridge	Installer	2011	330	330	330
CrWind	Installer	2015	1	1	1
FakeFlash	Installer	2012	3	3	3
Gamex	Installer	2012	7	7	7
Gapev	Installer	2012	7	7	7
Gappusin	Installer	2012	58	*58*	*0*
Ansca	Trojan	2011	1	0	1
Antares	Trojan	2012	2	0	2
Faketimer	Trojan	2012	16	16	16
Fujacks	Trojan	2013	1	0	0
Moghava	Trojan	2012	3	3	3
Copycat	Ransomware	2014	10	10	10
FoCober	Ransomware	2015	13	13	13
Koler.C	Ransowmare	2014	7	7	7
AsRoot	Rootkit	2011	8	8	8
Coogos	Rootkit	2014	8	8	8
Droidrooter	Rootkit	2011	3	3	3
DroidCoupon	Rootkit	2011	1	1	1
DroidKungFu	Rootkit	2011	402	402	402
ExploitLinuxLootor	Rootkit	2012	70	70	70
ExploitRageCage	Rootkit	2012	1	*1*	*0*
Oldboot.b	Rootkit	2012	1	1	1
Placms	Rootkit	2012	12	12	12
RATC	Rootkit	2012	1	*1*	*0*
Spyhasb	Rootkit	2012	13	13	13
Stiniter	Rootkit	2012	1	1	1
YZHC	Rootkit	2011	22	22	22
Adsms	SMS Trojan	2011	3	3	3
BaseBridge	SMS Trojan	2011	122	122	122
BgServ	SMS Trojan	2011	9	*9*	*0*
BeanBot	SMS Trojan	2011	8	8	8
DogoWars	SMS Trojan	2011	7	7	7
Dialer	SMS Trojan	2012	1	1	1
FakeInstaller	SMS Trojan	2013	925	925	925
FakeLogo	SMS Trojan	2012	20	20	20
FakeMart	SMS Trojan	2013	1	1	1
FakeNotify.B	SMS Trojan	2013	1	1	1
FakePlayer	SMS Trojan	2010	17	17	17
FakeRegSMS.B	SMS Trojan	2013	41	41	37
Foncy	SMS Trojan	2012	2	2	2
GGTrack	SMS Trojan	2013	5	5	5
GoldenEagle	SMS Trojan	2011	1	1	1
Hispo	SMS Trojan	2014	3	3	3
Jifake	SMS Trojan	2012	29	29	29
Jsmshider	SMS Trojan	2012	2	2	2
Mania	SMS Trojan	2012	6	6	6
Opfake	SMS Trojan	2012	14	14	14
Poder	SMS Trojan	2015	*1*	*1*	0
Qiscom	SMS Trojan	2012	1	1	1
Raden	SMS Trojan	2012	10	10	10
Saiva	SMS Trojan	2014	2	2	2
Samsapo	SMS Trojan	2013	1	1	1
Scavir	SMS Trojan	2012	1	1	1
Seaweth	SMS Trojan	2012	6	6	6
Selfmite.B	SMS Trojan	2013	1	1	1
SerBG	SMS Trojan	2013	14	14	14
SingaporeSMSWorm	SMS Trojan	2014	1	1	1
SmsSend	SMS Trojan	2013	1	1	1
Ssmsp	Sms Trojan	2014	1	1	1
Stealer	SMS Trojan	2014	14	14	14
TrojanSMS.BoxerAQ	SMS Trojan	2012	1	1	1
TrojanSMS.Denofow	SMS Trojan	2011	5	*5*	*0*
TrojanSMS.Hippo	SMS Trojan	2012	13	13	3
TrojanSMS.Stealer	SMS Trojan	2014	1	1	1
Updtbot	SMS Trojan	2012	1	1	1
Vidro	SMS Trojan	2014	5	5	5
XXShenqi	SMS Trojan	2014	1	1	1
Zsone	SMS Trojan	2011	12	12	12

TABLE 4
Detection Results and Comparisons (Part 2)

Malware	**Class**	**Year**	**Samples**	**MADAM**	**Virus Total**
ACNetDoor	Spyware	2015	1	0	1
Aks	Spyware	2012	5	5	5
Arspam	Spyware	2011	1	1	1
Booster	Spyware	2014	1	1	1
Cellshark	Spyware	2011	1	0	1
Dougalek	Spyware	2012	9	0	9
DynSrc	Spyware	2013	1	*1*	*0*
Fidall	Spyware	2012	1	0	1
Flexispy	Spyware	2011	2	2	2
GamblerSms	Spyware	2011	1	0	1
Gone60	Spyware	2011	9	9	9
GPSSMSSpy	Spyware	2011	6	0	6
Kidlogger	Spyware	2011	6	6	6
Kmin	Spyware	2011	52	52	52
Kiser	Spyware	2013	9	9	0
Maistealer	Spyware	2012	1	1	1
MobileSpy	Spyware	2012	14	0	14
Mobinauten	Spyware	2011	8	8	8
Mtracker	Spyware	2014	1	*1*	*0*
Netisend	Spyware	2011	1	0	1
NickySpy	Spyware	2011	2	0	2
Plankton	Spyware	2011	11	0	11
SndApps	Spyware	2011	10	10	10
Spitmo	Spyware	2011	11	11	11
Tapsnake	Spyware	2011	2	0	2
SMS Replicator	Spyware	2013	4	4	4
Sheridroid	Spyware	2012	2	0	2
SmForw	Spyware	2011	2	2	2
SmsSpy	Spyware	2013	1	1	1
SmsZombie	Spyware	2012	10	0	10
Spybubble	Spyware	2011	3	0	3
Spy.Imlog	Spyware	2012	1	1	1
Spyoo	Spyware	2012	3	3	3
Tesbo	Spyware	2012	1	0	1
Trackplus	Spyware	2014	6	0	6
Typstu	Spyware	2011	14	14	14
Vdloader	Spyware	2011	16	16	16
Walkiwat	Spyware	2011	1	1	1
Ycchar	Spyware	2012	2	0	2
Ksapp	Spyware + Installer	2012	6	6	6
DroidDream	Spyware + Rootkit	2011	16	16	16
Gmuse	Spyware + Rootkit	2014	3	3	3
zHash	Spyware + Rootkit	2011	11	11	11
Dabom	SMS Trojan + Installer	2014	2	2	2
Updtkiller	SMS Trojan + Installer	2012	1	1	1
Bosm.C	SMS Trojan + Installer	2015	1	1	1
Boxer	SMS Trojan + Installer	2011	27	27	27
Cawitt	SMS Trojan + Spyware	2012	1	1	1
Cosha	SMS Trojan + Spyware	2012	10	10	10
Fjcon	SMS Trojan + Spyware	2012	4	4	4
MobileTx	SMS Trojan + Spyware	2012	69	69	69
Nandrobox	SMS Trojan + Spyware	2012	13	13	13
Geinimi	SMS Trojan + Botnet	2011	69	69	69
Total	-	-	**2,784**	**2,700**	**2,709**
Accuracy	-	-	-	**96.9%**	**97.3%**

been submitted. For the sake of representation, Tables 3 and 4 reports the VirusTotal majority voting for each sample, i.e., an app is considered as malicious if the majority of reports considers the app as a malware.

Malware belonging to the *Rootkit* category is detected by the *Global Monitor*. In fact, we have observed that they cause strong fluctuations in the number of issued system calls, not coherent with the current user activity, both if the user is actively interacting with the device or not. Furthermore, all *Rootkits* using the so called "Rage Against The Cage" [24] technique to perform buffer overflow are detected by the *Per-App Monitor*, which observes a large number of forked processes, belonging to the app. Other rooting attacks, which attempt to re-write the file system of the `sys` folder, and to install the *BusyBox* tool, are also detected through system call analysis by the *Global* and *Per-App Monitors*. In fact, the attempt to perform the rewriting and mount of the `sys` folder generates a high amount of system call. Note that some specific rootkits firstly check if the device has been already rooted. In this case, to tackle this attack, MADAM includes a third-party utility named `Superuser`, which intercepts and blocks any request of accessing to root privileges. Finally, some rooting malware exploit specific kernel or OS vulnerabilities, such as the *GingerBreak* attack, to gain root privileges. These attacks are effective only against specific versions of Android, often working only on a specific device, and are generally fixed through patches and new system releases. Since these attacks are not general, MADAM needs to be trained specifically on each device to be able to tackle them. *Installer* malware are blocked as soon as they try to install a new application. The *App Evaluator* intercepts this event regardless of the source, i.e., even if the installation is requested from a malicious app.

MADAM has been proved totally effective against *SMS Trojans* correctly identifying 40 families of malware. We underline that, normally, Android does not allow the monitoring of the event of outgoing text messages sent by an app, unless the app developer explicitly declares the notification intent. However, through exploiting an X-Posed module, MADAM intercepts, controls (extracts text and recipient) and even blocks outgoing messages if deemed as malicious. We have analyzed 41 *Spyware* families. All of them have been correctly identified as suspicious by the *App Evaluator*. However, only 23 families (56 percent of accuracy on spyware families) have been detected by MADAM at runtime. We point out that actions of some *Spyware* are hard to detect at run-time, as once they receive the correct authorizations they perform operations which are legal from a behavioral point of view and can evade MADAM detection. Moreover, some non trojanized *Spyware* apps can be found also on marketplaces, distributed with the specific purpose of sending device information to an external server. To this end, the *App Evaluator* is able to discern between a trojanized and genuine app, complementing the capabilities of *Global Monitor*.

An important class of tested malware is the *Ransomware* class, which is a type of malware that became popular in the last years, especially in Europe [25], where MADAM has successfully identified the two families in the dataset. As an example, `Koler.c` is a *Ransomware* which, after installation, asks the user the administrator rights to control the action of "screen-lock". Afterward, the *Ransomware* uses a background service (watch-dog) to spam every 5 ms a web page as top activity (current user interface). Therefore, the user can only interact with the malicious interface, which also asks the user to pay an amount of money as a ransom for removing the malware. Moreover, since the app has the right of device administrator, it is not possible to remove it programmatically. However, MADAM is able to effectively contrast the `Koler.C` malware. At first, the *App Evaluator* correctly recognizes the app as suspicious, due to declared permissions necessary to control the device top activity. If the user installs the app, the app name is then inserted in the *Suspicious List*. Then, MADAM detects a misbehavior at kernel-level, caused by the high activity of the watch-dog service. Afterward, MADAM verifies, through the *Per-App Monitor*, if the malicious app detains administrator privileges and, in case, removes it, by deleting the corresponding apk and odex (executable) files. Then, MADAM forces a reboot of the device, to prevent the user from making the mistake of paying the ransom. At reboot, the app is not active anymore and the user is notified of the app removal.

Globally, MADAM has correctly detected 2,700 app samples out of 2,784, showing an accuracy of 96.9 percent, which is in line with the accuracy of VirusTotal, able to identify 2,709 samples (using the majority voting approach). However, it is worth noting that MADAM has been able to detect nine malware families which evade VirusTotal checks (shown in Italic in Tables 3 and 4). In particular, `Poder` is a recent malware (SMS Trojan) known to be able to evade most anti-virus [14], and VirusTotal does not detect it. In these cases, MADAM detects and stops the misbehavior of the outgoing SMS message, typical of SMS Trojan. These results shows how the MADAM approach is a valid and effective alternative to static signature-based approaches, which are more accurate against low profile malware whose signature is known, such as those spyware not detected by MADAM.

5.1.1 Evasion and Detection Limitations

First, we note that any behavior-based detection is subject to poisoning attacks. In the MADAM case, however, an adversary cannot poison the dataset to confuse the detector, since it only considers behavioral-classes defined in terms of malware's goals. On the other hand, in theory, it is possible for a smart adversary to execute mimicry attacks by inserting malicious code into a bening app to evade MADAM's detection. We believe that this is a general limit of any behavior-based intrusion detection system: the goal of MADAM is more focused in being effective at detecting real attacks, rather than generic attacks. In fact, existing evasion attacks, such as mimicry attacks, are more proof-of-concepts attacks rather than real attacks. Furthermore, we also note that evading detections is not a trivial task [26], since the attacker must be able to build a malware whose behavior both reaches the goal and does not matches the MADAM behavioral-class (which is built taking into consideration the goal). In particular, automating such evasion is not an easy task. However, this is an interesting research topic that deserves consideration. Finally, note that any malware, in particular those of the Botnet class, which waits for external commands is not discovered by MADAM at run-time until a command is received, i.e., until a misbehaviour is triggered. Similarly, some samples of malware are not clearly malware nor potentially-unwanted-programs (PUP), such as Spyware (an example being the “FarMap” sample), and also AV companies strive to classify them in one class or the other. However, in most of the cases, MADAM App Risk Assessment component signals these apps as “dangerous”.

5.2 Usability Analysis

In this section we analyze the impact of MADAM on user experience and performance of the device.

TABLE 5
False Alarms Experimental Results

Test	FPs	FPR	FPs/day
Light	3	$1 \cdot 10^{-5}$	0.5
Medium	8	$2.8 \cdot 10^{-5}$	1.1
Heavy	75	$2.6 \cdot 10^{-4}$	10.7

5.2.1 False Positives

False positives do not have a direct effect on the security of the device. However, they are not desirable since alarms require interaction with the user and become bothersome if not real. Keeping their number low is thus of capital importance to ensure usability. We have measured the real amount of false positives generated with three different patterns of real device usage. As detailed in the following, the generated number of false positives, with a normal device usage, amounts on the average to one per day.

Usability experiments have been conducted on three devices with three users and different configurations. In the first experiment, called *Light Usage*, we tested a Samsung Galaxy S2 with Android Jelly Bean version 4.2. The device only installed the native apps except for MADAM. The user has been instructed to keep the smartphone mainly in standby, except for performing/receiving phone calls and/or sending/receiving text messages from the default messaging app. In the second experiment, called *Medium Usage*, we used a Samsung Galaxy Nexus with Android Jelly Bean version 4.3. The device installed 54 legitimate apps, including the native ones and MADAM. The user has been instructed to use the device normally. The user accessed the Internet on a daily basis, using three instant messaging apps and two social network apps. He also played daily with a 2D graphic videogame (`Angry Birds Space`) and a 3D one (`Temple Run 2`), and took several daily pictures with the smartphone camera. No new apps have been installed by this user on the device. In the third experiment, called *Heavy Usage*, we tested a LG Nexus 4 equipped with Android Kit-Kat version 4.4. At the beginning of the experiment the device equipped 52 apps, including MADAM and super-user manager. The user has been instructed to keep the device always active (screen always on), interacting with it as much as possible. The user installed during the experiment 91 new legitimate apps and used gaming apps, camera to take pictures and record video, in addition to instant messaging, SMS and phone calls. The experiments lasted for one week, every day from approximately 10:00 to 21:00, to avoid the reduction of activity normally caused by the night. Results are shown in Table 5, which reports the total number of false positives issued during the three experiments by the two K-NN classifiers, and the average number per day. The FPR computed on the amount of monitored events by the two K-NN classifiers ($60 \times 60 \times 11 \times 7$ for the short-term and $60 \times 11 \times 7$ for the long-term).

It is worth noting that the number of FP per day noticeably raises with a heavy usage. In particular, a large number of false positives have been issued contemporary to the installation of new apps (the user installed 91 legitimate apps during the week). The event of installation of new apps is, in fact, computationally heavy and causes a sharp increase in the amount of issued system calls. However, the installation of a new app is handled by the package installer, which is a component of the OS. Hence, MADAM will only notify a (global) anomaly, without deeming any app as responsible for the misbehavior. Moreover, MADAM offers the possibility to handle false positives, allowing the users to re-train the classifiers, adding the false positives to the training set, as genuine events. Using this functionality, the user can train MADAM to learn her own behavior, likely reducing the amount of issued false positives. As a further experiments, we have retrained the classifiers adding five FPs from the heavy-usage experiment to the training set and then we have used the device in the same conditions of the same experiment. In this test, the average FPs per day have been reduced to 3. To further reduce the amount of issued FPs, the user can switch MADAM to the "Training Mode" using the UI. In this mode, the user will not be notified of any alarm, and the behaviors deemed as malicious are immediately added to the classifier training set. Note that for the K-NN classifier, this operation of adding a new element to the training set can be done without training it again from scratch. This feature further motivates our decision in using the K-NN. However, "Training Mode" should be used carefully, i.e., the user should be sure that her device is not infected when activating this mode, to avoid the risk of training the classifier with malicious behavior considered as genuine. Moreover, over-training the classifier may cause over-fitting with a consequent detection performance degradation.

An additional set of experiments has been performed by analyzing a set of 9,804 genuine apps extracted from Google Play, hence already analyzed and considered genuine by VirusTotal. This test is aimed at showing that genuine apps do not affect the FPR, thus the device usability. In this set, 22 out of 9,804 (0.2 percent) apps have been classified as suspicious by the *App Classifier* and inserted in the *Suspicious List*. After installation of these apps, no considerable deviation of the FPR have been observed during this analysis. We have made available online[9] the full list of analyzed good apps with the features relevant for the *App Classifier*, and the classification results.

5.2.2 Performance Overhead

The performance overhead of MADAM has been measured through a standard benchmark tool, i.e., the `Quadrant Standard Edition` app, which is distributed through Google Play.[10] Performance tests have been performed on the same device used for malware detection experiments: Samsung Galaxy Nexus, CPU dual-core 1.2 GHz Cortex-A9, RAM 1 GB, GPU PowerVR SGX540. The device runs Android 4.3 Jelly Bean, stock version. Apart from the native apps, the only apps installed were the super user manager and the MADAM application.

Table 6 reports the benchmark for the system when MADAM was running (third column from left, "MADAM") and when it was not (second column left, "Vanilla"). The last column reports the overhead computed as a percentage overhead between the two performances. Benchmarks are provided as indexes, where a highest value means a better performance. Benchmarks reported have been computed as the average of five experiments, both in "Vanilla" and "MADAM" configuration. The overhead of MADAM is caused by both the kernel module, which hijacks system calls, and a *Global Monitor* service that runs in background when the system is active. As shown, the performance impact of MADAM is acceptable; in fact, the overall performance impact (Total) is 1.4percent. It is worth noting that the stronger impact is on memory (9.4percent). This is mainly due to the chosen classifier. In fact, the K-NN classifier does not cause heavy load on the CPU. However, the K-NN requires to continuously keep in memory the whole training set, which may require a noticeable amount of space [22]. On the other hand, we note that MADAM has no impact on 2D/3D performances. This was expected, since MADAM functionalities does not influence the GPU. Furthermore, the 4 percent performance degradation on I/O is not perceived by user, whose experience is not altered [27]. Additional overhead to be considered is the one introduced by the *App-Evaluator* when a new app is installed. This overhead only affects the user experience once, i.e., when installing a new app, where the user is already prepared to wait some seconds for the app to be installed. On three devices, we measured that on average the *App-Evaluator* increases by 3 to 7 seconds the app installation phase.

TABLE 6
Benchmark Tests

Test	Vanilla	MADAM	Overhead
Total	2,911	2,868	1,4%
CPU	5,509	5,459	0,9%
Memory	2,660	2,409	9,4%
I/O	3,860	3,705	4%
2D	327	327	0%
3D	2,250	2,250	0 %

5.2.3 Energy Consumption

We have evaluated the battery consumption of MADAM monitoring the battery depletion over two periods of 24 hours. Results reported an overhead of 4 percent, which is in line with the overhead of current antivirus software[28]. Details on this evaluation are reported in Appendix, available in the online supplemental material. Finally, in Table 7 we compare the results of MADAM with existing security framework for Android. In particular, we compare the overhead, the detection rate, the kind of attacks detected, if rooting is required, and the kind of performed analysis.

6 Related Work

We can partition the related work in two classes: (i) run-time monitoring of the events, (ii) static analysis of the code, or of the medatada, to detect know patterns of misbehaviors.

Run-Time Detection: TaintDroid [29] is a security framework for Android devices which tracks information flow to avoid malicious stealing of sensitive information. Differently from MADAM TaintDroid targets a very specific class of attacks. Moreover, TaintDroid requires a custom ROM of the Android system, to implement the information flow

9. http://www.android-security.it/madam/goodware_app_list.xlsx.
10. http://play.google.com/store/apps/details?id=com.aurorasoftworks.quadrant.ui.standard

TABLE 7
Comparison of MADAM Results with Existing Frameworks

System	Dynamic/Static	Rooting	Detection Rate	Overhead	Attack
MADAM	Both	Yes	96.9%	1.4%	Several Classes of Attacks
TaintDroid [29]	Dynamic	Custom-ROM	N.A.	14%	Privacy Leak
Patronus [30]	Dynamic	Yes	87%	7.1%	SMS - Spyware
DroidAnalyzer [31]	Static	Offline	N.A.	N.A.	Rootkits
DroidSIFT [32]	Static	Offline (Server)	93%	N.A.	General
AlterDroid [33]	Static	Offline	97%	N.A.	Obfuscated Malware
ASF [34]	Dynamic	Custom ROM	N.A.	2-3 %	Access Control

mechanisms. In [30] the authors present *Patronus*, a HIPS for Android that can prevent mobile malware intrusions and detect malware at run-time. Patronus implements API hijacking to the binder at client and server side, to overcome the bypassing of the client-based hooking. The authors report a total overhead of their tool of 7.1 percent, while MADAM is 1.4 percent. A behavioral analysis of Android apps at the system call level is presented in [35]. The authors propose a framework called *CopperDroid* that discerns good behaviors from bad ones, by automatically stimulating malicious apps to misbehave through instrumentation. The analysis of behaviors is automatic, which means that the behavior of the stimulated app by user interaction is not considered as in MADAM. Android Security Framework (*ASF*) [34] is a generic and extensible security framework for Android that provides security API to facilitate the inclusion of security extensions in Android. This approach is orthogonal to MADAM: the goal of MADAM is to detect malware, i.e., anomalies, while ASF is more oriented to the enforcement of policies. The authors of [31] presents a system which aim at detecting rootkit hidden in trojanized apps. This framework, *DroidAnalyzer*, identifies the features which are typical of rootkits and then looks for them statically in the code of apps, performing the analysis on an external server. On the contrary, MADAM performs the analysis on the mobile device, and is focused on several classes of malware. *MOSES* [36] is a policy-based framework that enforces software isolation of apps (and data) on Android. MADAM is more focused on malware detection, even if it allows users to define some high-level policies for apps.

Static Analysis: Alterdroid [33] is a tool that compares the differences in behavior between an original app and automatically generated version that contain modifications (faults) to detect hidden malware, such as in pictures. Differently from MADAM, Alterdroid performs static analysis and does not target general malware, being not able to detect pieces of malware that do not hide malicious code in static resources. [37] proposes a method for malware detection based on embeddings of function call graphs in a vector space capturing structural relationship. This representation is used to detect Android malware using machine learning techniques by achieving a good accuracy. Similarly, [32] classifies Android malware via dependency graphs by extracting a weighted contextual API dependency graph as program semantics to construct feature sets. These approaches implement a static detection of Android malware while MADAM implements both static and run-time analysis. [38] statically analyzes app to derive a set of features for malware detection at application-level and evaluates several classifiers for Android apps. MADAM also analyzes system calls and user activities and classifies the activities at run-time. Similarly, *DREBIN* [39] performs static analysis of Android apps to gather features that are embedded in a joint vector space, such that typical patterns indicative for malware can be automatically identified and used for explaining the decision. An approach similar to the *App Classifier* of MADAM is presented in [40], which proposes to communicate an index assessing the risk level of an Android application. However, the proposed index is mainly intended for a comparison between similar apps, pushing the user to choose the less risky. MADAM, on the other hand, is specifically targeted at detecting malicious apps and conveys the risk score to the user to reduce the chance of installing malware.

Note that this paper largely extends the work presented in [41]. In particular, the completely novel additions are: (i) the original *Global Monitor* has been extended with a module for static app classification (*App Risk Assessment*), and a *Per-App Monitor* which analyzes and handles the behavior of specific apps; (ii) we have defined a malware classification in behavioral classes with feature correlation; (iii) we have performed a much more extensive experiments for both detection capability and device usability, also providing a performance analysis for user experience and for battery leakage. These updates have allowed us to improve the detection rate from 93 to 96 percent, and to reduce the CPU performance overhead from 7 to 1.4 percent.

7 Conclusion

This paper proposes MADAM, a multi-level host-based malware detector for Android devices. By analyzing and correlating several features at four different Android levels, MADAM is able to detect misbehaviors from malware behavioral classes that consider 125 existing malware families, which encompass most of the known malware. To the best of our knowledge, MADAM is the first system which aims at detecting and stopping at run-time any kind of malware, without focusing on a specific security threat, using a *behavior-based* and *multi-level* approach.

Acknowledgments

This work is supported by FP7 EU-funded project Coco Cloud under grant No. 610853, by H2020 EU-funded project NeCS No. 675320, Italian Ministry of Education, University and Research (PRIN Project TENACE), and University of Pisa (PRA 2016 program).

REFERENCES

[1] (2014). Global mobile statistics 2014 part a: Mobile subscribers; handset market share; mobile operators. [Online]. Available: http://mobiforge.com/research-analysis/global-mobile-statistics-2014-part-a-mobile-subscribers-handset-market-share-mobile-operators.

[2] (2014). Sophos mobile security threat reports. [Online]. Available: http://www.sophos.com/en-us/threat-center/mobile-security-threat-report .aspx.

[3] M. G. C. Funk. (Dec. 2013). Kaspersky security bullettin. [Online]. Available: http://media.kaspersky.com/pdf/KSB_2013_EN.pdf.

[4] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *Proc. ACM Eur. Workshop Syst Security*, Apr. 2013, pp. 1–15.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Proc. 19th Annu. Netw. Distrib. Syst. Security Symp.*, Feb. 2012, pp. 1–18.

[6] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard fine-grained policy enforcement for untrusted android applications," in *Proc. Data Privacy Manag. Auton. Spontaneous Security*, 2014, pp. 213–231.

[7] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proc. 4th Int. Conf. Trust Trustworthy Comput.*, 2011, pp. 93–107. [Online]. Available: http://dl.acm.org/citation.cfm?id=2022245.2022255.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971.

[9] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proc. 1st ACM Workshop Security Privacy Smartphones Mobile Devices*, 2011, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046624.

[10] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. Symp. Usable Privacy Security*, 2012, p. 3.

[11] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16.

[12] (2016). Contagio mobile, mobile malware mini dump. [Online]. Available: http://contagiominidump.blogspot.com.

[13] GoogleGroups. (2015). *Virustotal*. [Online]. Available: https://www.virustotal.com/.

[14] Dr.Web. (2015). Android malware review. [Online]. Available: http://news.drweb.com/show/review/?lng=en&i=9546.

[15] K. S. Labs. (2014). Kindsight security labs malware report h1. [Online]. Available: http://resources.alcatel-lucent.com/?cid=180437.

[16] F. Del Bene, G. Dini, F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, and D. Sgandurra. (2015). Risk analysis of android applications: A multi-criteria and usable approach. Consiglio Nazionale delle Ricerca - Istituto di Informatica e Telematica, Rome, Italy, Tech. Rep. TR-04-2015. [Online]. Available: http://www.iit.cnr.it/node/32795.

[17] T. C. (2013, May). Say goodbye to custom stock roms and hello to xposed framework. [Online]. Available: http://www.xda-developers.com/android/say-goodbye-to-custom-stock-roms-and-hello-to-xposed-framework/.

[18] D.-K. Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proc. IEEE 6th Annu. SMC Inf. Assurance Workshop*, Jun. 2005, pp. 118–125.

[19] D. Mutz, F. Valeur, G. Vigna, "Anomalous system call detection," *ACM Trans. Inf. Syst. Security*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[20] G. Vigna, W. Robertson, and D. Balzarotti, "Testing network-based intrusion detection signatures using mutant exploits," in *Proc. 11th ACM Conf. Comput. Commun. Security*, 2004, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030088

[21] T. M. Cover, P.E. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp. 21–27, Jan. 1967.

[22] O. Kramer, "Dimensionality reduction by unsupervised k-nearest neighbor regression," in *Proc. 10th Int. Conf. Mach. Learn. Appl. Workshops*, Dec 2011, vol. 1, pp. 275–278.

[23] A. Developer. (2015). *Android smsmanager api reference page*. [Online]. Available: http://developer.android.com/reference/android/telephony/SmsManager.htm l

[24] V. Misra. (2013). What are the exact mechanisms/flaws exploited by the "rage against the cage" and "z4root" android exploits?. [Online]. Available: http://www.quora.com/What-are-the-exact-mechanisms-flaws-exploited-by-the-rage-against-the-cage-and-z4root-Android-exploits.

[25] A. Lucent, "Motive security labs malware report h2 2014," 2014, https://www.alcatel-lucent.com/solutions/malware-reports.

[26] H. Kayacik and A. Zincir-Heywood, "Mimicry attacks demystified: What can attackers do to evade detection?" in *Proc. 6th Annu. Conf. Privacy, Security Trust*, Oct 2008, pp. 213–223.

[27] M. J. Darnell, "Acceptable system response times for TV and DVR," in *Proc. 5th Eur. Conf. Interactive TV: Shared Experience*, 2007, pp. 47–56. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763017.1763025

[28] (2015). How antivirus affect battery life. https://www.luculentsystems.com/techblog/minimize-battery-drain-by-antivirus-software/.

[29] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. (2014 Mar.). Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM* [Online]. *57(2)*, pp. 99–106. Available: http://doi.acm.org/10.1145/2494522

[30] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang, "Design and implementation of an android host-based intrusion prevention system," in *Proc. 30th Annu. Comput. Security Appl. Conf.*, 2014, pp. 226–235. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664245

[31] S.-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *J. Netw. Comput. Appl.*, vol. 38, no. 0, pp. 43–53, 2014.

[32] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2014, pp. 1105–1116. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660359

[33] G. Suarez-Tangil, J. Tapiador, F. Lombardi, and R. Di Pietro, "Thwarting obfuscated malware via differential fault analysis," *Computer*, vol. 47, no. 6, pp. 24–31, Jun. 2014.

[34] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky, "Android security framework: Extensible multi-layered access control on android," in *Proc. 30th Annu. Comput. Security Appl. Conf.*, 2014, pp. 46–55. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664265

[35] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," presented at the 6th Eur. Workshop Syst. Security, Prague, Czech Republic, Apr. 2013.

[36] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: Supporting and enforcing security profiles on smartphones," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 3, pp. 211–223, May 2014.

[37] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proc. ACM Workshop . Artif. Intell. Security*, 2013, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/2517312.2517315

[38] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proc. Security Privacy Commun. Netw.*, 2013, vol. 127, pp. 86–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04283-1_6.

[39] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2014, pp. 1–15.

[40] C. Gates, J. Chen, N. Li, and R. Proctor, "Effective risk communication for android apps," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 3, pp. 252–265, May 2014.

[41] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: A multi-level anomaly detector for android malware," in *Proc. Comput. Netw. Security*, 2012, vol. 7531, pp. 240–253.

Andrea Saracino received the PhD degree in computer engineering from University of Pisa. He is currently a postdoc researcher at the National Research Council of Italy (CNR). His research activity focus on intrusion detection, quantitative system modeling, privacy aware data analysis, and trust and reputation in distributed environments.

Daniele Sgandurra received the PhD degree in computer science from the University of Pisa. He is currently a research associate at the Department of Computing, Imperial College London. His main research fields include virtualization and cloud security, mobile security, attack graphs and risk management, and malware analysis.

Gianluca Dini is an associate professor at the Dipartimento di Ingegneria dell'Informazione, University of Pisa. His research interests include the field of distributed computing systems, with particular reference to security. He is currently working on security in networked embedded systems.

Fabio Martinelli is a senior researcher of Institute of Informatics and Telematics (IIT), Italian National Research Council (CNR) where he leads the security project. He is coauthor 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.