MADAM: a Multi-Level Anomaly Detector for Android Malware

Gianluca Dini[1], Fabio Martinelli[2], Andrea Saracino[1,2], and Daniele Sgandurra[2]

[1] Dipartimento di Ingegneria dell'Informazione
Università di Pisa, Pisa, Italy
firstname.lastname@iet.unipi.it
[2] Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche, Pisa, Italy
firstname.lastname@iit.cnr.it

Abstract. Currently, in the smartphone market, Android is the platform with the highest share. Due to this popularity and also to its open source nature, Android-based smartphones are now an ideal target for attackers. Since the number of malware designed for Android devices is increasing fast, Android users are looking for security solutions aimed at preventing malicious actions from damaging their smartphones.
In this paper, we describe *MADAM*, a Multi-level Anomaly Detector for Android Malware. MADAM concurrently monitors Android at the kernel-level and user-level to detect real malware infections using machine learning techniques to distinguish between standard behaviors and malicious ones. The first prototype of MADAM is able to detect several real malware found in the wild. The device usability is not affected by MADAM due to the low number of false positives generated after the learning phase.

Keywords: Intrusion detection, Android, Security, Classification

1 Introduction

In the last years, mobile devices, such as smartphones, tablets and PDAs, have drastically changed by increasing the number and complexity of their capabilities. Current mobile devices offer a larger amount of services and applications than those offered by personal computers. At the same time, an increasing number of security threats targeting mobile devices has emerged. In fact, malicious users and hackers are taking advantage of both the limited capabilities of mobile devices and the lack of standard security mechanisms to design mobile-specific malware that access sensitive data, steal the user's phone credit, or deny access to some device functionalities. In 2011, malware attacks increased by 155 percent across all platforms [1]: in particular, Android is the platform with the highest malware growth rate by the end of 2011.

To mitigate these security threats, various mobile-specific Intrusion Detection Systems (IDSes) have been recently proposed. Most of these IDSes are *behavior-based*, i.e. they do not rely on a database of malicious code patterns, as in the

case of *signature-based* IDSes. A behavior-based (or anomaly-based) IDS is a system that attempts to learn the normal behavior of a device. To this end, the system is firstly trained by receiving as input a set of parameters that describes the way the user normally behaves. Secondly, during the normal usage, the IDS is able to recognize as suspicious any behavior that strongly differs from those well-known, i.e. learnt during the first phase.

In this paper, we describe *MADAM*, a Multi-level Anomaly Detector for Android Malware, which monitors Android both at the kernel-level and user-level to detect real malware infections. MADAM exploits machine learning techniques to distinguish between standard behaviors and malicious ones. A first prototype of MADAM has been implemented for Android smartphones, but its theoretical approach can be extended to other mobile operating systems (OS) as well. The first set of results show that this approach works well with real malware and it is usable since it has a very low false positive rate.

The main contributions of the paper are the following:

- We describe the design and implementation of MADAM, a host-based *real-time* anomaly detector that exploits a *multi-level view* of the monitored smartphone, which considers both OS events, namely the issued system calls, and smartphone parameters, e.g. the user activity/idleness, to detect intrusion attempts.
- We show that a dataset with a *small number of parameters* (13 features), and a relatively small number of elements, is effective in describing the smartphone behavior to a machine learning system; furthermore, MADAM can *self-adapt* to new behaviors by including new elements in the training set learnt at run-time.
- The framework has been implemented and tested on *real devices* (Samsung Galaxy Nexus) to understand the users' experience. The tests have been performed with more than 50 popular applications and several user behaviors to measure the false positives; on the average, a user receives less than 5 false positives per day, and the overall performance overhead is acceptable, i.e. 3% of memory consumption, 7% of CPU overhead and 5% of battery.
- To the best of our knowledge, MADAM is the first anomaly-based IDS for Android that has been tested using *real malware*: furthermore, at the time of the tests, some of the tested malware were *zero-day-attacks* and current off-the-shelf security solutions were not able to detect them. The system shows a detection rate of 93%, and in particular of 100% with rootkits.
- MADAM is able to detect *unwanted outgoing SMSes* stealthily sent by Android malicious applications.

The rest of the paper is organized as follows. Section 2 lists some related work. Section 3 describes the MADAM architecture and its current implementation. Section 4 reports some preliminary tests and results. In Sect. 5 we discuss the features and the current limitations of the framework. Finally, Sect. 6 concludes by discussing some future works.

2 Related Work

Crowdroid [2] is a machine learning-based framework that recognizes Trojan-like malware on Android smartphones, by analyzing the number of times each system call has been issued by an application during the execution of an action that requires user interaction. A genuine application differs from its trojanized version, since it issues different types and a different number of system calls. Crowdroid builds a vector of m features (the Android system calls). Differently from this approach, MADAM uses a global-monitoring approach that is able to detect malware contained in unknown applications, i.e. not previously classified. Furthermore, on *Crowdroid* only two trojanised applications have been tested, whereas on MADAM we tested ten real malware. A similar approach is presented in [3], which also considers the system call parameters to discern between normal system calls and malicious ones.

Another IDS that relies on machine learning techniques is *Andromaly* [4], which monitors both the smartphone and user's behaviors by observing several parameters, spanning from sensors activities to CPU usage. 88 features are used to describe these behaviors; the features are then pre-processed by feature selection algorithms. The authors developed four malicious applications to evaluate the ability to detect anomalies. Compared to Andromaly, MADAM uses a smaller number of features (13), and has been tested on real malware found in the wild, and shows better performance in terms of detection and, especially, of false positives rate. After the learning phase, the false positive rate of MADAM is 0.0001, whereas that of [4], which uses a sampling method similar to that of MADAM and with a comparable sampling rate (2 seconds), is 0.12. The detection rate of MADAM is 93%, while that of [4] is 80%.

Other approaches only monitor misbehaviors on a limited number of functionalities such as outgoing/incoming traffic [5], SMS, Bluetooth and IM [6], or power consumption [7] and, therefore, their detection accuracy is higher of other work but less general.

[8] monitors smartphones to extract features that can be used in a machine learning algorithm to detect anomalies. The framework includes a monitoring client, a *Remote Anomaly Detection System* (RADS) and a visualization component. RADS is a web service that receives, from the monitoring client, the monitored features and exploits this information, stored in a database, to implement a machine learning algorithm. In MADAM, the detection is performed locally and, more importantly, in real-time. [9] proposes a behavior-based malware detection system (*pBMDS*) that correlates user's inputs with system calls to detect anomalous activities related to SMS/MMS sending. MADAM is more general since it considers all the activities on a smartphone. A further framework targeted at SMS/MMS monitoring is *Proactive Group Behavior Containment* [10], which is aimed at containing malicious software spreading in these messaging networks.

[11] and [12] propose *Kirin* security service for Android, which performs lightweight certification of applications to mitigate malware at install time. Kirin certification uses security rules that match undesirable properties in security

configuration bundled with applications. [13] performs static analysis on the executables to extract functions calls usage using `readelf` command. Hence, these calls are compared with malware executables for classification. Finally, [15] surveys some security solutions for mobile devices.

3 MADAM Approach

MADAM is a Multi-level Anomaly Detector for Android Malware that concurrently monitors Android at the kernel-level and user-level to detect real malware infections using machine learning techniques to distinguish between standard behaviors and malicious ones. In fact, the problem of anomaly detection can be seen as a problem of binary classification, in which each normal behavior is classified as "Standard", whereas abnormal ones are classified as "Suspicious". Some behavior-based IDSes rely on computational intelligence and machine learning techniques, such as clustering [2], probability-based classifiers [4] [5], decision trees [5] and others. Henceforth, we will use the generic term "classifier" for these techniques.

Classifiers automatically learn how to classify a set of items. In the proposed scenario they could be seen as a black-box whose input is a set of behaviors and the output for each behavior is "Standard" or "Suspicious". A classifier understands how to correctly classify elements after the execution of a training phase. This phase is critical, since it determines the accuracy of the classifier. Hence, it is fundamental to provide the classifier with a good training set.

To build a good dataset for smartphones, i.e. one that represents a typical smartphone behavior, MADAM considers elements that represents behaviors both when the user is active and when she is idle. Moreover, our training set also contains some malicious behaviors, which strongly differ from the standard ones. Usually, the collected features come from several sources of events [4]: choosing the right features to best represent the smartphone behaviors is a critical task, since their number and correlation determine the quality of the training set [16]. As discussed in Sect. 3.1, MADAM considers two levels, the kernel-level and the application-level. Table 1 provides a list of features that can be monitored at the kernel and user-level.

3.1 Multi-Level Detection

MADAM is a Multi-level Anomaly Detector for Android Malware that combines features extracted from several levels to (i) provide a wider range of monitored events and (ii) discover correlations among these events belonging to distinct levels. Currently, MADAM considers two levels, the kernel-level and the application-level. At the first level, MADAM monitors system calls. In fact, we believe that system calls are a good representative sample of the smartphone behavior, since their usage is a monitor for user activity, files and memory access, incoming/outgoing traffic, energy consumption and sensors status. More importantly, they can be used as monitors for intrusion attempts: this is based

Level	Features
Kernel	system calls running processes free RAM CPU usage
User/Applications	idle/active key-stroke called numbers sent/received SMS Bluetooth/Wi-Fi analysis

Table 1. Features at Distinct Levels

upon the assumption that an attacker has to execute one (or several) system calls to harm the system. At the second level, the extracted features consider whether the user is idle or not, and the number of sent SMSes. A high-level view of MADAM architecture is depicted in Fig. 1.

To extract features from these two levels, the framework includes two monitors. The first one is a kernel-level monitor that intercepts all the critical system calls, and that records the number of their occurrences during a period T. Hence, if m is the number of monitored system calls, this monitor returns a vector of dimension m at each period T.

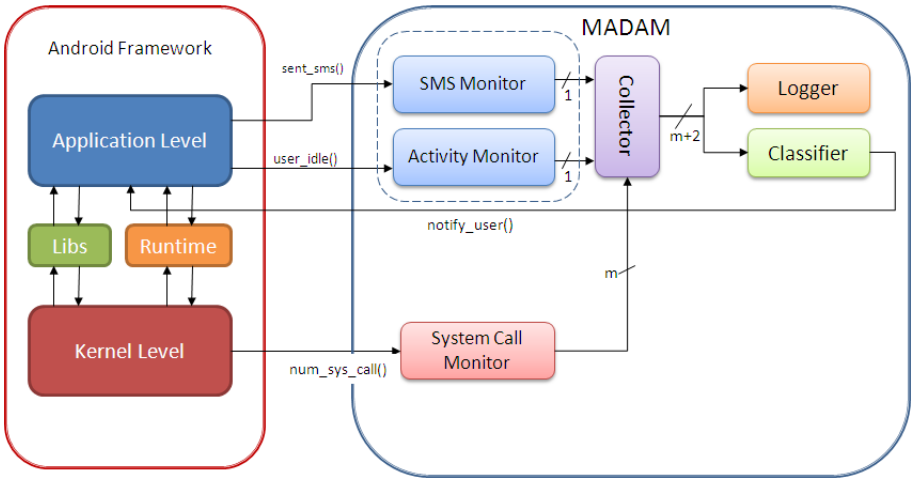

Fig. 1. Functional Blocks of MADAM

The second monitor is at the application-level, and it can be split in two sub-monitors that handle two different tasks: (i) to periodically measure the number of SMS sent in a time interval; (ii) to monitor the user idleness. The user idleness is a fundamental feature since the activity of the device is usually more intense

when the user is interacting with the device itself: hence, the number of issued system calls depends upon the status of the device/user. Since after a very short period of user inactivity the smartphone screen is turned off by the OS, the user can be considered active either if the screen is on or a voice call is active [17].

The elements of the datasets are vectors with $m+2$ features, where m is the number of monitored (critical) system calls and the last two features represent, respectively, the device status (idle or active) and the number of sent SMSes. A *collector* receives these features from all the monitors and then builds the vectors. These vectors are stored in local files using a *logger* module so that they can be used to build a training set, which is composed of $\frac{t}{T}$ vectors, where t is the total time spent collecting data and T is the logging interval (an input parameter of the framework). A training set is then used to obtain a trained *classifier*. This phase of data gathering, preprocessing and classifier training, is called the *Training Phase*. In the *Operative Phase*, which is the phase where the user actually uses the smartphone, each monitored vector is given as input to the trained *classifier* and, if it is classified as *suspicious*, a notification is immediately shown to the user.

3.2 Implementation

We have developed the framework on a *Samsung Galaxy Nexus* HSPA, with OS Android *Ice Cream Sandwich* version `4.0.1`, and Linux kernel version `3.0.1`. The lowest-level component of MADAM framework is the system call monitor, which has been implemented as a Linux kernel module that hijacks the execution of the monitored system calls: each system call is coupled with a counter that is incremented before its execution. In the current implementation, this module considers only a subset of all the available system calls on Android Linux, those that are rather critical, in term of security, in the description of the system behavior (see Sect. 4.1). The kernel module contains a task that periodically (with a period of T) logs the actual value of the counters on a shared buffer with the collector and then resets all the counters. The inclusion and execution of the hijacking module requires the Super User (SU) permissions: since on the Android production builds (the OS version installed on device by manufacturers) SU is disabled, during the tests the devices required *rooting*, which is a procedure to get root permissions.

The highest-level component of the framework includes an Android Application in Java, which has been implemented using the Android SDK. The first component of the Java Application is the MADAM *collector*, which periodically reads (i) the buffer shared with the kernel monitor, (ii) the user status (idle/active), (iii) the number of SMSes sent in the period T. Since Android only allows monitoring SMSes that are sent through the default SMS manager, i.e. an application can send SMSes without the user being notified, to detect sent SMSes MADAM exploits the Android system log file (*LogCat*), which contains the output of a low level function that is called each time an SMS is being sent. Furthermore, the Java application also includes two parallel tasks. The first one is the application-level *logger* (Figure 1), which reads the vectors built by the

collector and logs them in a log file that results in a matrix with $\frac{t}{T}$ rows. The second task is the *classifier* that states if the vectors built by the *collector* are good or suspicious. In the latter case, the classifier sends a notification to the user and logs those vectors that have been classified as suspicious, for further analysis. For classification we used Weka[3] version 3.6.6, an open source library in Java that includes several classification tools.

4 Experimental Results

In this section we describe in detail the tests which were performed both for malware detection and false positives measurement.

Application	Type	Native	Application	Type	Native
Adobe Reader	PDF reader	No	AlarmDroid	Alarm Manager	No
Angry Birds	Game	No	Angry Birds Space	Game	No
Astro	File Manager	No	Browser	Internet Browser	Yes
Calculator	Utility	Yes	Compass	Utility	No
Calendar	Utility	Yes	Voice Composer	User Interface	Yes
Color Note	Memo Manager	No	Defender II	Game	No
Camera	Video Capture	Yes	Contacts	Contact Manager	Yes
Download	Download Browser	Yes	Dropbox	Cloud Storage	No
Earth	3D Map Utility	No	Email	E-Mail Manager	Yes
Facebook	Social Network	No	Flash Player	SWF player	No
Fruit Ninja	Game	No	Gallery	Multimedia Viewer	Yes
GMail	Cloud E-Mail	Yes	Google Talk	Google Chat	Yes
Google+	Social Network	Yes	ilMeteo	Weather Forecasting	No
Hamster Bomb	Game	No	Opera	Web Browser	No
Instagram	Picture Sharing	No	Jewels Star	Game	No
SIM Manager	SIM Manager	Yes	Latitude	Advanced Navigator	Yes
Places	Smart Maps	Yes	Maps	Maps Utility	Yes
Messages	SMS/MMS Manager	Yes	Messenger	Chat Manager	Yes
MADAM	IDS	No	Movie Studio	Video Editor	Yes
Music	Audio Player	Yes	Navigator	Navigator	Yes
News and Meteo	News Utility	Yes	One Touch Drawing	Game	No
Play Store	Application Installer	Yes	QR Droid	QR Scanner	No
Google	Web Search	Yes	Skype	VoIP	No
System Panel	Task Manager	No	Smart System Manager	Task Manager	No
Superuser	Rooting Utility	No	Phone	Call Manager	Yes
Temple Run	Game	No	TGCom24	News Utility	No
Google Translator	Utility	No	Viber	VoIP	No
Wikipedia	Encyclopedia	No	YouTube	Video Streaming	Yes

Table 2. Tested Applications

4.1 Training Set and Classifiers

To do so, we have logged the behavior (through system calls) of the phone during the execution of normal actions performed by a user. In this logging phase we

[3] http://www.cs.waikato.ac.nz/ml/weka/

have tried to ensure that the device has not been infected: we have installed only popular applications from the official site (Google Play) having a high rating and positive comments. For a full list of tested applications refer to Tab. 2.

After a first set of preliminary tests, we have noticed that the system calls that best describe the device behavior are the following: `open`, `ioctl`, `brk`, `read`, `write`, `exit`, `close`, `sendto`, `sendmsg`, `recvfrom`, `recvmsg`. We expected such a result, since Android is a framework composed by several functional blocks that communicate using the mechanisms provided by the underlying Linux kernel and an increase in the smartphone activity causes directly a sharp increase in the occurrences of these system calls, all of which concern buffer or file operations, or communications between the framework components. This is why the change in the number of occurrences of these system calls is generally related with the user idleness. Hence, to build the training set, we consider as standard vectors those with a low number of occurrences of these system calls and the user idle, and those where the number of system call occurrences is high and the user is active.

In addition to the previous 12 features (11 system calls and user idleness), the vectors used for classification also includes a further feature representing the number of sent SMSes in the time interval T. In fact, monitoring SMS usage is semantically difficult through system calls only and SMS messages can be used to harm the user, stealing her credit. Moreover, SMSes are strongly related with the user activity. In fact, in a normal usage, an SMS is sent after the user has composed the message, which requires an active interaction. However, some applications send or receive SMSes to provide some kind of services. Since, SMS is a costly service, if compared to the amount of data that are sent with a message, applications should avoid SMS as communication channel as much as possible, and they should require that the user actively agrees with the sending of each message. Applications that send SMS messages when the user is idle should be considered suspicious. For all these reasons, we have logged several SMS sending phases, which represent real-life usage scenarios, and we have added the resulting vectors to the dataset.

Classifiers are not able to recognize a suspicious element if they are not trained also with some elements that belong to the suspicious behavior class. As previously said, a suspicious behavior is one that strongly deviates from those known to be good. Hence, we have manually defined some suspicious elements by creating both vectors with a high number of system call occurrences, when the user is idle, and vectors with an extremely high number of system call invocations, when the user is active. Figure 2 depicts some examples of standard and suspicious vectors. The picture depicts four sample vectors monitoring occurrences for 11 system calls with $T = 1$ sec. The last number of each vector means 1 for user active and 0 for user idle.

More malicious vectors were derived from the ones that we have defined using a data balancing method named *SMOTE* (Simple Minority Oversampling TEchnique), which creates new vectors from those provided by means of interpolation. To represent malicious behaviors concerning SMS messages, we have manually

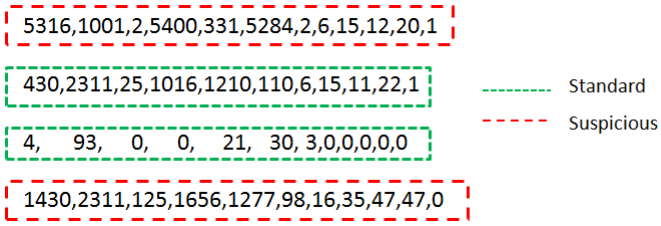

Fig. 2. Sample Vectors Monitoring Occurrences for System Calls and Idleness

defined and added to the training set some vectors with a number of sent messages that is very high compared to the user activity. We would like to point out that if classifiers are trained using such a dataset, which does not include malicious vectors generated by real malware, then each malware, if detected, can be considered as a *zero-day-attack*.

To increase the detection rate, our application runs in parallel two instances of the same detection framework, with a different sampling period T. The first instance is a short-term monitor with $T_{short} = 1$ sec, whereas the second instance constitutes a long-term monitor with $T_{long} = 60$ sec (both values are configurable at run-time). The cooperation of these two instances detects different types of misbehaviors. The short-term monitor is more effective in detecting "spiky" misbehaviors, i.e. with sudden, brief and sharp increase of the system call occurrences. On the other hand, the long-term monitor is aimed at detecting misbehaviors that distribute their action constantly in a long period of time, such as spyware, i.e. whose effect is not immediate.

Hence, two different datasets were built and used to train two classifiers of the same type. The classifier is a K-Nearest Neighbors (K-NN) [18] with $K = 1$ (1-NN). This classifier has very good performance and can easily adapt to a large number of problems, requiring a small amount of computation time to classify an element and a trivial update algorithm. We have also tested several other classifiers on our dataset but the 1-NN outperforms them all.

4.2 Experiments Description

Figure 3 describes at a high level the sequence of steps performed during the experiments.

During the *Training Phase*, the classifiers are trained with the initial, manually-defined, training set described in Sect. 4.1. The *Learning Phase* follows the training phase and it is used to learn behaviors that are specific of the user. This phase has been used to obtain an estimate of the False Positive Rate (FPR) trend (see Sect. 4.3), i.e. how the number of false positives decreases as they are used to progressively update the trained classifiers. During the *Operative Phase*, the trained classifiers are used to perform anomaly detection. During this experiment, this phase has been divided into two sub-phases: during *FPR measuring* the device has been tested with clean applications to compute the number of false positives

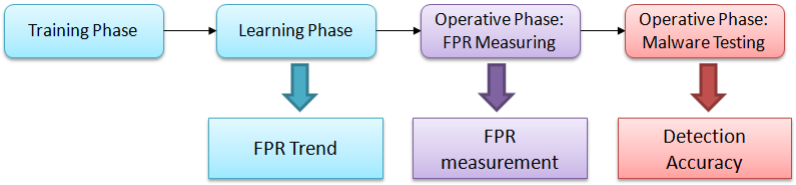

Fig. 3. Experimental Phases

raised per day; in *Malware Testing* trojanized applications have been installed on the device to determine the detection accuracy of the MADAM classifiers. Since the learning phase and FPR measurement greatly depend on the usage of the device, these tests were performed by three distinct users.

The next two sections describe the tests performed during the *Operative Phase*.

4.3 False Positive Measurement

Anomaly-based IDS have been criticized since they are more likely to generate false positives. False positives may strongly reduce the device usability, so we have performed a critical analysis of their occurrence on our system.

FPR Trend. A first experiment has been performed to estimate the FPR trend, i.e. how the number of false positives decreases as they are used to progressively update the trained classifiers. The training set that we have manually defined and given to the short-term classifier contained 900 standard vectors and 100 malicious ones. The long-term classifier has been trained with 250 standard vectors and 50 malicious ones. These datasets are relatively small and they represent some standard and basic behaviors (for the standard vectors), such as phone calling, SMS messages typing and sending, Internet browsing and gameplay of the popular game Angry Birds. Soon after the first dataset had been manually set up, and the classifier started, as we expected some false positives were raised (see Tab. 3 for details). False positives are likely to occur when the user performs a new behavior that strongly differs from those stored in the training set. Due to both the high number and the diversity of applications available for Android, unknown behaviors are likely to occur.

To reduce the occurrence of false positives, MADAM has to learn how the user behaves in an initialization phase, which we call the *learning phase*, where false positives are directly added to the classifier knowledge base without any user intervention. During the tests, the average duration of a learning phase to obtain a reasonable number of false positives is 30 minutes. However a new learning phase can be initiated actively by the user when she wants to update the classifier with the generated false positives, for example by a newly installed application (if she considers that application trustworthy). Figure 4 shows how the FPR decreases immediately after the training phase. During this experiment

we have updated the classifier in five steps: after ten minutes and then each hour for four hours. More details on this experiment are reported in Tab. 3.

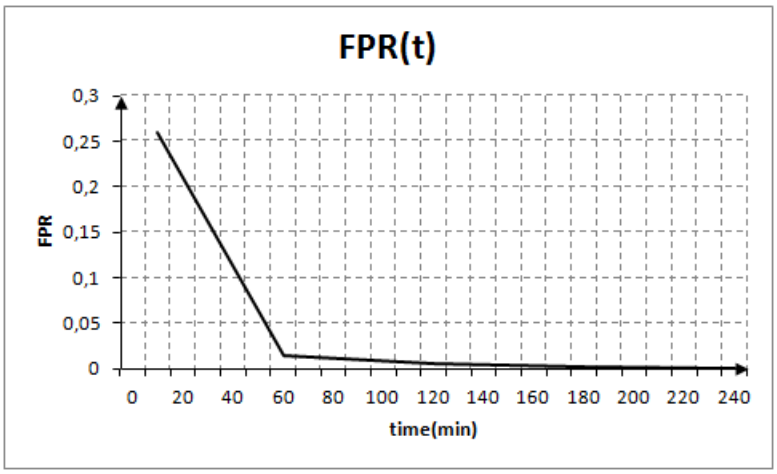

Fig. 4. FPR Decrease During the Learning Phase

Time	10 min	60 min	120 min	180 min	240 min
Vectors	610	3050	3660	3660	3660
False Positives	156	55	23	10	5
FPR	0.26	0.015	0.0061	0.0028	0.0011

Table 3. Learning Phase

FPR Measurement. We used the training set obtained from this learning phase to re-train the classifiers, and then we performed furthers experiments to estimate the number of false positives raised in 24 hours. During these tests, one of the smartphones has been equipped with more than 50 applications (see Tab. 2) and heavily used during the day. The other two smartphones have been set up, respectively, for moderate and basic usage. As expected, these smartphones with the lowest usage have raised a lower number of FP than the first one[4]. For this reason, here we only focus on the tests performed on the heavily-used smartphone.All the applications reported in Tab. 2 have been used during the 24 hours. The table lists, for each application, if it was natively installed on the device or it has been downloaded from *Google Play*.

[4] during these experiment the classifiers have not been updated with the false positives, which were added to the training set only at the end of the experiments.

The test returned 15 false positives ($FPR = 0.000171$), 9 of which were raised by the short-term classifier ($FPR = 0.000104$) and 6 by the long-term one ($FPR = 0.004167$). We updated the classifiers with the collected false positives, and then we reiterated the experiment for the following two days. Table 4 shows these results: on the average, on the heavily-used smartphone less than ten false positives are raised during 24 hours, with a descending trend.

Further tests have been performed using the non-trojanized version of some applications used for malware detection, to check if they would raise false positives as well[5]. We installed a clean version of the web browser `Opera` and of the `Hamster Bomb` game, while their trojanized versions were infected respectively by `OpFakeA` and `TGLoader`. As expected, no intrusions were detected.

Day	**Overall FPR**	T_{short} **FPR**	T_{long} **FPR**
1	0.000171	0.000104	0.004167
2	0,000139	0.000116	0.00137
3	0.000114	0.00008102	0.00208

Table 4. False Positive Rate

4.4 Malware Detection

We have tested MADAM with real Android malware hidden in trojanized applications: all the malware applications are taken from a repository[6] that is updated as soon as new threats are discovered. The tested malware belong to different categories, e.g. Trojan, Rootkit and Spyware.

`open`	`ioctl`	`brk`	`read`	`write`	`exit`	`close`	`sendto`	`sendmsg`	`recvfrom`	`recvmsg`	idleness	SMS Num
2246	25481	4341	47	16899	14416	12916	178	139	179	186	0	2

Table 5. One of the Malicious Vectors Monitored of `OpFakeA` Malware

Each malware has been monitored as standalone to avoid cross malware detections. Furthermore, to reduce the likelihood that the suspicious vector has been caused by a false positive, each malware has been tested three times, restoring the device to a clean state after each test. Table 5 reports one of the vectors that MADAM (the T_{long} instance) classified as malicious during the infection of the malware `OpFakeA` (see Tab. 6). The last two elements of the vector are the most important: they mean that 2 SMS messages sending requests have been

[5] these tests have been performed on some applications only because of the difficulty of finding a clean and trustworthy version of all the trojanized applications, which are only available on un-official markets.

[6] http://contagiominidump.blogspot.it/

issued in a time interval with no user activity, a behavior that should be considered malicious for the SMS policies formerly discussed. After these tests, the dataset has not been updated with the suspicious vectors, so that each detected malware can be effectively considered as a zero-day-attack.

Table 6 shows the results of the three tests performed for each malware. The table also specifies which instance of the system monitor, i.e. short-term (T_{short}) or long-term (T_{long}), has detected the malware. This should give an idea of which type of misbehavior is performed by the malware. Those malware that have been detected by both classifiers are usually the most aggressive. We will further discuss these results in Section 5.

Malware	Type	Detection Rate	T	Description
`Lena.B`	BootKit	100%	T_{short}	Modifies files in the system partition.
`Moghava`	Trojan	100%	T_{long}	Modifies pictures stored on the device. Gradually fills the SDCard memory.
`TGLoader`	RootKit	100%	Both	Obtains root privileges, installs other malicious applications, opens a backdoor.
`OpFakeA`	Trojan	100%	T_{long}	Sends SMS with SIM data, downloads applications and stores them on the SDcard.
`NickySpyB`	Spyware	66%	Both	Record calls, stores them on the SDcard then sends them with other user's data to an external server.
`Gone in 60 sec`	Spyware	66%	T_{short}	Sends user's data to an external server.
`KMin`	Trojan	100%	T_{long}	Sends SMS to premium rate numbers
`Lotoor`	Rootkit	100%	Both	Obtains root privileges and opens several backdoors.
`DroidDream`	Rootkit	100%	T_{long}	Obtains root privileges and opens a backdoor.
`Droid Kung Fu`	Rootkit	100%	Both	Sends device information to a remote server.

Table 6. Malware Detection Results

4.5 Performance

In the performed tests, the MADAM's impact on performance has not greatly influenced the user experience. The users have not noticed any reduction in responsiveness or in general visual performance. The periodic services of MADAM require an average of 7% of CPU overhead and of 3% MB of RAM space. Figure 5 depicts two different traces of CPU and memory usage, from left to right, without and with MADAM running, taken using the `System Panel` application. The native battery monitor of the Android settings reports that MADAM uses only 2-5% of the total smartphone battery.

5 Discussion

To the best of our knowledge, MADAM is the first *real-time* anomaly-based malware detector developed for *real devices*, specific for Android, that is able to detect *real malware* of different categories. The detection results and FPR are

Fig. 5. Traces of CPU and RAM Consumptions

better than those of previous anomaly-based detection systems for Android ([4], [2]).

The overall detection accuracy was of 100% for all malware, except for two that were not detected in one of the tests. These malware (`NickySpy` and `Gone in 60 seconds`) are spyware and their behavior is less aggressive compared to that of the other monitored malware. `NickySpy` records all the calls on the SDCard and then sends them via `HTTP` to an external server. The monitored device behaviors during a normal call and during an eavesdropped one show clear differences. However this misbehavior can be confused with the one in which a heavy application, such a 3D videogame, is running and, hence, the system can be deceived in such a situation.

`Gone in 60 seconds` is not a real malware but an application that a user intentionally installs on the device and when it is started reads all the user data, such as SMS messages and contacts, and sends them all to an external server. Then, the applications is automatically uninstalled, all in no more than 60 seconds (hence the name). During the execution, the application displays to the user a number that can be inserted on a website, hosted on the same server where the data have been sent, to retrieve the data. The behavior of this malware results more aggressive when there are much more data on the phone, in the other cases its detection can result tricky.

It is important to underline that, differently from previous approaches, our proposed framework is not based upon a per-application monitoring: instead, it performs a *global monitoring*, i.e. it is oblivious of which application(s) generated the event(s). This method can be more effective in identifying sudden behavior changes: as an example, a method that could be used to trick per-application controls is that of developing some applications, harmless if taken as standalone,

but that can cooperate to perform an attack. A proof-of-concept of these applications is presented in [4], where the malicious application, a video-game, looks harmless because it does not ask any dangerous Android permission. However, after the installation, this application shares the permissions with another trojanized application that does not performs malicious operations, but that has the permission to send both SMS and MMS. Then, the video-game starts to send sensitive information about user's contacts by means of SMS messages. Such an attack should be identified more easily by means of a global monitoring system, which considers all of the system calls issued in a time interval. Being a global anomaly detector, MADAM is able to detect an intrusion attempt but it is not able to detect the malicious source. However, its response can be used to trigger further components able to track and stop the source of the malicious behavior.

A question that may arise is how the user is able to distinguish between a false positive and a real intrusion. After the learning phase, occasional false positives become a rare event, so occasionally detection can be related with them. In fact, all the tested malware, show aggressive behaviors that cause periodical and multiple detections in a limited period of time. The only exception concerns SMS-based malware, which should be handled with an *ad hoc* strategy. A possible extension to this framework can automatically handle the occasional false positives, or can guide the user through a smart learning phase, to learn as much as possible from her behavior in a short period of time. The same framework can be used to trigger a new learning phase when a new trustworthy application is installed.

6 Conclusions and Future Works

In this paper, we have presented MADAM, a framework that allows early detection of intrusion attempts and malicious actions performed by real malware for Android devices. The framework exploits a multi-level approach, i.e. that combines features at the kernel-level and at the application level, and is based upon machine learning techniques. The first set of results is encouraging: the first prototype of MADAM for Android smartphone has managed to detect all the 10 monitored real malware, with a negligible impact on the user experience due to the few false positives issued per day. To the best of our knowledge, these results are a noticeable improvement to solutions presented in previous work, both for detection rate of real malware on current Android-based smartphones, and occurrences of false positives.

Since the tests provided promising results, we are working on an extension of this framework that combines the global monitoring approach with more specific monitors that consider additional features. With this extension we would like to create a database of expected behaviors that are related to high level actions, suck as starting a phone call. This should increase the system accuracy and allow the detection of a larger number of malware. Finally, whenever an alarm is triggered by this architecture, a further extension requires the tracing of the running applications to detect and stop the source of the attack.

References

1. Juniper Networks: 2011 Mobile Threats Report (February 2012)
2. I. Burguera, U.Z., Nadijm-Tehrani, S.: Crowdroid: Behavior-Based Malware Detection System for Android. In: SPSM '11, ACM (October 2011)
3. D. Mutz, F. Valeur, G. Vigna: Anomalous System Call Detection. ACM Transactions on Information and System Security **9**(1) (February 2006) 61–93
4. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss: Andromaly: a behavioral malware detection framework for android devices. Journal of Intelligent Information Systems **38**(1) (January 2011) 161–190
5. D. Damopoulos, S.A. Menesidou, G. Kambourakis, M. Papadaki, N. Clarke, S. Gritzalis: Evaluation of Anomaly-Based IDS for Mobile Devices Using Machine Learning Classifiers. Security and Communications Networks **5**(00) (2011) 1–9
6. A. Bose, K.G. Shin: Proactive Security For Mobile Messaging Networks. In: WiSe '06. (September 2006)
7. G.A. Jacoby, R. Marchany, N.J.Davis, IV: How Mobile Host Batteries Can Improve Network Security. IEEE Security and Privacy **4** (2006) 40–49
8. Schmidt, A.D., Peters, F., Lamour, F., Scheel, C., Çamtepe, S.A., Albayrak, S.: Monitoring smartphones for anomaly detection. Mob. Netw. Appl. **14**(1) (2009) 92–106
9. Xie, L., Zhang, X., Seifert, J.P., Zhu, S.: pBMDS: a behavior-based malware detection system for cellphone devices. In: Proceedings of the Third ACM Conference on Wireless Network Security, WISEC 2010, Hoboken, New Jersey, USA, March 22-24, 2010, ACM (2010) 37–48
10. Bose, A., Shin, K.G.: Proactive security for mobile messaging networks. In: WiSe '06: Proceedings of the 5th ACM workshop on Wireless security, New York, NY, USA, ACM (2006) 95–104
11. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: CCS '09: Proceedings of the 16th ACM conference on Computer and communications security, New York, NY, USA, ACM (2009) 235–245
12. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: Computer Security Applications Conference, 2009. ACSAC '09. Annual. (dec. 2009) 340–349
13. Schmidt, A.D., Bye, R., Schmidt, H.G., Clausen, J.H., Kiraz, O., Yüksel, K.A., Çamtepe, S.A., Albayrak, S.: Static Analysis of Executables for Collaborative Malware Detection on Android. In: Proceedings of IEEE International Conference on Communications, ICC 2009, Dresden, Germany, 14-18 June 2009, IEEE (2009) 1–5
14. Damopoulos, D., Menesidou, S.A., Kambourakis, G., Papadaki, M., Clarke, N., Gritzalis, S.: Evaluation of anomaly-based IDS for mobile devices using machine learning classifiers. Security and Communication Networks **5**(1) (2012) 3–14
15. La Polla, M., Martinelli, F., Sgandurra, D.: A survey on security for mobile devices. Communications Surveys Tutorials, IEEE **PP**(99) (2012) 1 –26
16. N. Kwak, C.H. Choi: Input Feature Selection for Classification Problems. IEEE TRANSACTIONS ON NEURAL NETWORKS **13**(1) (January 2002) 143–159
17. H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, D. Estrin: Diversity in Smartphone Usage. In: MobiSys'10, ACM (June 2010)
18. T. M. Cover, P.E. Hart: Nearest Neighbor Pattern Classification. IEEE Transactions on Information Theory **IT-13**(1) (January 1967) 21–27