I Find Your Behavior Disturbing: Static and Dynamic App Behavioral Analysis for Detection of Android Malware

Fabio Martinelli*, Francesco Mercaldo*, Andrea Saracino*, Corrado Aaron Visaggio†
*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy
{name.surname}@iit.cnr.it
†Department of Engineering, University of Sannio, Benevento, Italy
visaggio@unisannio.it

***Abstract*—Malicious Android applications are currently the biggest threat in the scope of mobile security. To cope with their exponential growth and with their deceptive and hideous behaviors, static analysis signature based approaches are not enough to timely detect and tackle brand new threats such as polymorphic and composition malware. This work presents BRIDEMAID, a novel framework for analysis of Android apps' behavior, which exploits both a static and dynamic approach to detect malicious apps directly on mobile devices. The static analysis is based on n-grams matching to statically recognize malicious app execution patterns. The dynamic analysis is instead based on multi-level monitoring of device, app and user behavior to detect and prevent at runtime malicious behaviors. The framework has been tested against 2794 malicious apps reporting a detection accuracy of 99,7% and a negligible false positive rate, tested on a set of 10k genuine apps.**

I. Introduction

Due to its increasing popularity, Android is currently the target of more than 99% of the security attacks toward mobile platforms. This is not surprising since Android holds 84.4% of the total market share[1] in smartphones, tablets, wearable devices, smart TV and smart home devices. According to the Trend Micro 1H 2015 Report[2] the volume of Android infected apps (*trojanized*) grew up from 4.26M of 2014 to 7.10M in the first half of 2015.

Standard trojanized apps [32] are now also sided by new threats such as polymorphic and composition malware [9], which exploit dynamic code load or modification to reduce the likelihood of being discovered.

Solutions to attempt to mitigate this issue came both from research and industrial field. From the industrial side, current solutions to protect users from new threats are mainly based on standard code or binary signature detection, which in mobile platform are still inadequate [24], [22]. The main issue is that using signature-based detection, a threat must be widespread for being successfully recognized, and attackers use different techniques to obfuscate code and binaries, making this signature collection task even harder. The dual approach for intrusion detection, is instead based on usage of classifiers for both static and dynamic behavioral analysis and data flow monitoring [27][26][6]. These approaches are effective in detecting specific threats and malicious behaviors, which are generally a subset of the existing ones, being thus not effective against the ones not belonging to this subset.

To supply this lack of generality, in this paper we propose BRIDEMAID (Behavior-based Rapid Identifier Detector and Eliminator of Malware for AndroID), a complete and accurate, on device analysis framework for Android apps which combines static and dynamic techniques to discriminate Android malware applications from legitimate ones.

The proposed framework leverages on a multi-level and multi-feature analysis which includes permission scoring and evaluation, opcode analysis, kernel level monitoring and API calls hijacking.

The resulting framework is modular, accurate and with a limited impact on performance. The framework has been tested on a dataset of more than 12.5k apps including both 10k verified genuine apps and malware extracted from three different datasets. The framework has also been tested against brand new attacks which are able to deceive native and commercial security checks, namely composition and polymorphic malware: it demonstrates that combining static and dynamic analysis we obtain an high detection ratio preserving the responsiveness user side if further analysis are not required. We compare our framework with the 57 signature-based antimalware hosted by VirusTotal [2], which provides the best benchmark for Android security evaluations, to demonstrate the advancement in state-of-the-art of mobile malware detection.

BRIDEMAID comes as the extension of two contributes: one implementing a method based on static analysis [7], while the second one relying on dynamic behavioral analysis [26] of malware behavior.

The novel contribution of the paper can be summarized as follows:

- We introduce BRIDEMAID, a novel framework for online detection of malicious Android apps, which exploits the benefits of both static and dynamic behavioral analysis for high detection accuracy of Android malicious apps.
- We present an analysis of the behavior of typical malicious apps, presenting the features that best help in identifying specific behaviors, to design effective strategies to detect and prevent any malicious misbehavior.
- BRIDEMAID has been tested against 2800 malicious applications also including proof of concept and real

[1] http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2] http://www.trendmicro.com/vinfo/us/security/threat-intelligence-center/mobile-safety/

polymorphic and composition malware, to verify the capability of the proposed framework in detecting zero day attacks.
- Finally we analyze the specificity of BRIDEMAID, testing it against a set of almost 10000 verified genuine apps.

The rest of the paper is organized as follows: Section II presents and characterizes the real world attacks widespread in mobile environment; Section III describes the BRIDEMAID framework and details its components; Section IV illustrates the experimental results on both malicious and genuine apps; Section V discusses related work; finally, Section VI draws the conclusions proposing some future directions.

II. Attack Patterns and Malware Behavior

In this section we describe the attack patterns implemented in current mobile malware (i.e. the modality employed by malware writer to embed the malicious payload in the application), and the malicious behaviors that mobile malware perform to damage the user.

A. Attack Patterns

Trojanized Apps: This attack pattern is the most common used by attackers to infect Android devices. Trojanized generally appears like genuine applications, showing functionalities of normal apps, i.e. videogames, utilities, etc. However, while the user is tricked by the genuine functionalities, trojanized apps run malicious code in background, which is likely to remain unnoticed by the user, till the effect becomes noticeable, e.g. the app manages to leak a consistent share of the user credit.

Repackaged Apps: The set of repackaged apps intersects without overlapping the set of Trojanized ones. A repackaged app is the new version of a genuine app maliciously modified by an attacker. The attacker disassembles the original app executable (apk file), changes or inserts new functionalities, reassembling the modified apk afterward. Users are easily tricked in installing repackaged apps, especially if the original app is a popular one, whose repackaged version is distributed through a non official marketplace. Added functionalities are not necessarily malicious, still malicious repackaged apps are the most common.

Polymorphic Malware: Malware showing the same behavior of a known ones still presenting an altered code, obtained through different program syntax, which still preserves the same semantic of the original code.

Composition Malware: Malware loading at run time malicious payload, does particularly difficult to identify since they do not show malicious code which could be statically analyzed [9]. This model basically relies on three capabilities: (i) the payload does not reside all in a single source, but it is obtained at runtime by composing different pieces, each one placed at a different location; (ii) the payload is observable only for a limited time window. The activation of the payload can be controlled by logical or temporal conditions. A condition could be, for instance, the detection of an antimalware scanning the app and, (iii) the application is benign at the beginning and for most of its lifetime. As a matter of fact, after the payload is executed, the host application turns to show the original benign behavior.

Zero Day Attacks: Not really an attack pattern, but a common word to indicate new threats which have been unknown till now. Zero day attacks are difficult to detect due to the new pattern unknown to black-list based intrusion detection systems.

B. Malware Misbehaviors

As discussed, the amount of malicious Android apps and malware families is continuously increasing and currently hundreds of thousands, whilst malware families count to hundreds. However, Android malicious apps shows for the majority a quite limited set of common behavior which can be grouped into a more limited and manageable number of classes:

1) **Rootkit**: malware that attempts to get super user (root) privileges on the device exploiting known vulnerabilities.
2) **SMS Trojan**: malware that send SMS messages stealthily and without the user consent, generally to subscribe the user to a premium services, send spam messages to user contacts, or exploit SMS-based authentication mechanism of some bank institutes to authorize unwanted transactions.
3) **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.
4) **Installer**: malware that install additional apps without the user consent.
5) **Botnet**: malware opening a backdoor on the device, waiting for commands which can arrive from an external server or an SMS message. Generally is combined with other behaviors.
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.

III. Detection Methodology

In this section we detail our methodology in order to explain the reason why we adopt a combined analysis.

As demonstrated by researchers in [17, 23], static analysis allows to advice the users about the maliciousness of an application in a poor amount of time, i.e. in terms of responsiveness is very powerful. Unfortunately, static

analysis may be ineffective in recognizing malware when attackers apply obfuscation techniques as, for instance, code reordering and junk code insertion. As matter of fact, malware writers implement increasingly sophisticated techniques for obfuscating malicious behavior, in order to evade detection strategies employed by actual antimalware products [32]. During its propagation, malware code changes its structure [18], through a set of transformations, in order to elude signature-based detection strategies. Indeed, polymorphism and metamorphism are rapidly spreading among malware targeting mobile applications [23]. The techniques of polymorphism and metamorphism have in common to change the form of each instance of malicious software with the aim to evade the signature-based detection. Polymorphic malware makes changes to code to avoid detection. It has two parts (i.e., a virus decryption routine (VDR) and an encrypted virus program body (EVB)), but one part (i.e., EVB) remains the same with each iteration, which makes the malware more easier to identify if compared with metamorphic one. The difficulty in metamorphic malware identification is indeed represented by the fact that malware in this case is completely rewritten with each iteration so that each succeeding version of the code is different from the preceding one. The code changes makes it difficult for signature-based antivirus software programs to recognize that different iterations are the same malicious program. In spite of the permanent changes to code, each iteration of metamorphic malware functions the same way. The longer the malware stays on device, the more iterations it produces and the more sophisticated the iterations are, making it increasingly hard for antivirus applications to detect and disinfect.

Hence, static analysis may be not adequate to detect the emerging threats [5]. On the other hand, dynamic analysis is hardly deceived by obfuscation techniques, in particular if such analysis is behavior-based. The rationale is that a malicious application, will perform an expected misbehavior whose effect is independent from the specific signature, i.e. is not affected by the obfuscation. However, dynamic analysis also brings noticeable drawbacks, such as computational overhead, which might cause battery depletion and performance degradation.

If the pre-filtering module marks the analysed application as malware, the user is notified, but whether the application is labelled as legitimate, our framework runs the application in order to extract a set of dynamic features to verify in the deep the application.

For these reasons analysis performed by the proposed framework is constituted of three consequent steps (static, meta-data and dynamic, as explained below) in which different features are retrieved and analyzed together with the two-fold effect of maximizing the malware detection rate and minimizing the amount of false positives. The main phases are illustrated in Figure 1. As shown, the behavior of each app is controlled from the very moment it enters the mobile device. In fact, as soon as the app has been downloaded, the static analysis engine will decompile

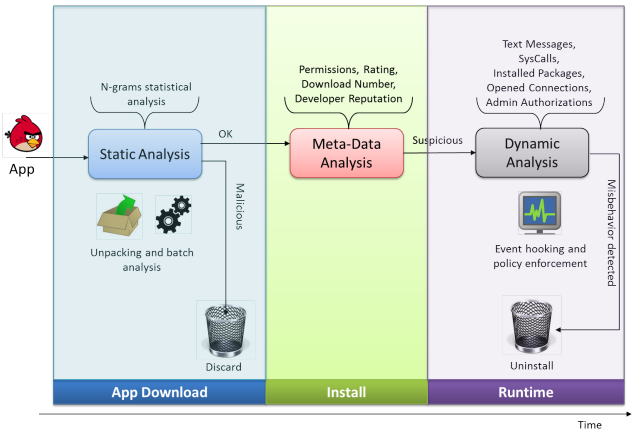

Figure 1. Analysis workflow of the proposed framework with the features extracted for each phase (i.e., Static, Meta-Data and Dynamic Analys).

the apk and analyze the source files looking for similarities in the executed actions exploiting n-grams. If the static analysis marks the app as malicious, the application will be removed, otherwise BRIDEMAID invokes the dynamic analysis to deep investigate about the trustworthiness of the application under analysis.

A. Static Analysis

We consider a binary classification problem in which an input application a has to be classified as malicious or genuine.

The pre-processing phase consists of two phases: a learning phase, in which the classifier is trained using a labelled dataset of applications, and the actual classification phase, in which an input application is classified as malicious or genuine. In both cases, each application is pre-processed in order to obtain numeric values (frequences of opcode sequences) suitable to be processed by the classifier.

1) Pre-processing: The pre-processing of an application consists of transforming an application a packed as an `.apk` file in a set of numeric values, as follows. We first use apktool[3] to extract from the `.apk` the `.dex` file, which is the compiled application file of a (Dalvik Executable); then, with the smali[4] tool, we disassemble the application `.dex` file and obtain several files (i.e., smali classes) which contains the machine level instructions, each consisting in an opcode and its parameters. From these files, we obtain a set of opcode sequences where each item is the sequence of opcodes corresponding to the machine level instructions of a method of a class in a.

We compute the frequency of opcodes $n$grams as follows. Let O be the set of possible opcodes, and let $\mathcal{O} = \bigcup_{i=1}^{i=n} O^i$ the set of $n$grams, i.e., sequences of opcodes whose length is up to n—n being a parameter of our method. We denote with $f(a, o)$ the frequency of the $n$gram $o \in \mathcal{O}$ in the application a: $f(a, o)$ is

[3]http://ibotpeaches.github.io/Apktool/
[4]https://code.google.com/p/smali/

hence the number of occurrences of o divided by the total length of the opcode sequences in a. Finally, we set the *feature vector* $\mathbf{f}(a) \in [0,1]^{|\mathcal{O}|}$ corresponding to a to $\mathbf{f}(a) = (f(a,o_1), f(a,o_2), \dots)$ with $o_i \in \mathcal{O}$.

In general, the size $|\mathcal{O}|$ of the feature vector $\mathbf{f}$ can be large, being $|\mathcal{O}| = \sum_{i=1}^{i=n} |O|^i$; however, not all possible $n$grams could be actually observed. We remark that we split the application code in chunks corresponding to class methods, since we want to avoid inserting meaningless $n$grams obtained by considering together instructions corresponding to different methods: in that case, indeed, we would wrongly consider as subsequent those instructions which belong to different methods.

2) Learning phase: The learning phase consists of obtaining a trained binary classifier C from two sets A_M, A_T of malware and trusted applications (the *learning sets*), respectively. The learning phase is divided into a feature selection phase and the actual classifier training phase.

The aim of the feature selection phase is two-fold: on the one hand, we want to reduce the dimension of the input—with $n = 2$, the size $|\mathcal{O}|$ of each feature vector $\mathbf{f}$ can be up to $\approx 10^{12}$. On the other hand, we want to retain only the more informative $n$grams, with respect to the output label, while removing noisy features. We consider $n$grams with $n = 2$ because a previous work [7] demonstrated that the sequences of two consecutive opcodes obtain better performance in Android malware identification, if compared with the opcode frequency (i.e., $n = 1$) [10] and values of n ranging from 3 to 5.

At first the average frequencies $\bar{f}_M(o)$ and $\bar{f}_T(o)$ are computed for each 2gram $o \in \mathcal{O}$ respectively on the malware and trusted applications:

$$\bar{f}_M(o) = \frac{1}{|A_M|} \sum_{a \in A_M} f(a,o)$$

$$\bar{f}_T(o) = \frac{1}{|A_T|} \sum_{a \in A_T} f(a,o)$$

We then compute the relative difference $d(o)$ between the two average values:

$$d(o) = \frac{\mathrm{abs}(\bar{f}_M(o) - \bar{f}_T(o))}{\max(\bar{f}_M(o), \bar{f}_T(o))}$$

The relative difference $d(o)$ is high if the 2gram o is frequent among malware applications and infrequent among trusted applications (and vice versa). Then, we build the set $\mathcal{O}' \subset \mathcal{O}$ of $n$grams composed of the h $n$grams with the highest values of $d(o)$, where h is a parameter of our method. We do not include in $\mathcal{O}'$ the $n$grams for which $d(o) = 1$, i.e., we purposely do not consider those 2grams which occur only in the trusted (malware) applications of the learning sets: this way, we strive to avoid building a classifier which works well on seen applications but fails to generalize. We then discard from $\mathcal{O}'$ each 2gram o_x for which another 2gram o_y exists in $\mathcal{O}'$ such that o_y is a supersequence of o_x: we perform this step in order to avoid considering redundant information, i.e., frequency of sequences of opcodes which largely overlap. Finally, we retain in $\mathcal{O}'$ only the remaining $k < h$ 2grams with the greatest value for $d(o)$—k being a parameter of the method. Accordingly, we set the *reduced feature vector* $\mathbf{f}'(a)$ corresponding to a using only the frequences of the 2grams in $\mathcal{O}'$, i.e., $\mathbf{f}'(a) = (f(a,o_1), f(a,o_2), \dots)$ with $o_i \in \mathcal{O}'$. The second step of the learning phase consists of training the actual classifier C using the reduced feature vectors obtained from the applications in the learning sets and the corresponding labels. In this work, we experimented with the Support Vector Machines (SVM) classifier.

3) Classification phase: The classification phase consists of determining if an application a is malicious or genuine, according to a learnt classifier C.

To this end, we repeat the pre-processing on a in order to obtain the reduced feature vector $\mathbf{f}'(a)$. Then, we input $\mathbf{f}'(a)$ to C and obtain a label in $\{\mathrm{malware}, \mathrm{trusted}\}$.

Note that, when pre-processing a, only the frequencies of $n$grams in $\mathcal{O}'$ have to be actually computed: in other words, some practical benefit can be obtained by building an effective classifier which works on a low number of features.

The classifier is built using as a training set 10% for each class of malware (and therefore families and samples) we considered in the study.

B. Meta-Data Analysis

This analysis is performed at deploy-time: from the apk it is extracted the manifest file (`AndroidManifest.xml`) to analyze the required permissions and compute a threat score. Also the source from which the apk has been installed is acquired and used to extract a set of metadata used to assess the trustworthiness level of the app. Namely the metadata extracted are the *market* of provenance (e.g. Google Play), the *download number*, the *user rating* and the *developer reputation* (if available). All these parameters are combined through the Analytic Hierarchy Process (AHP), which decides if the new app should be considered *Trusted* or *Suspicious*. The algorithm to compute the threat score and the instantiation of the AHP decision process are described in [11]. Apps which are considered trusted can execute on the device without additional check on the performed behaviors. Apps which instead are classified as suspicious will be subject to the control of the dynamic analysis monitors, as discussed in the following.

C. Dynamic Analysis

The dynamic analysis is performed at runtime, after the app has been successfully installed on the device and initiates the execution. The dynamic analysis, considers both global features, i.e. related to the device and operative system, and local features which are related to specifically monitored apps. The **Global Monitor** 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, 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.

IV. Experimental Results

In this section we present the experimental evaluation of BRIDEMAID, evaluating the detection capability and the false alarm rate, on a set of almost 12k apps both malicious and legitimate.

A. Datasets

The dataset is made of a set of 9804 genuine apps downloaded from Google Play[5] and a set of 2794 malicious apps belonging to 123 malicious families. The genuine apps have been downloaded from September 2014 to December 2015 and then verified with the VirusTotal service [2].

This service run 57 different antivirus software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on each application: the output confirmed that the trusted applications included in our dataset did not contain malicious payload. The obtained trusted dataset includes samples belonging to all the different categories available on the market. Moreover, we have verified that they were still present on the market in April 2016, hence, have not been found to be malicious, considering that the Google Bouncer service [19], removes apps found to be malicious after their usage, both from the market and remotely from devices which have installed it.

The malicious dataset is made of malicious apps of different families belonging to all the defined behavioral classes. The malicious apps have been extracted from the Drebin dataset [4, 28], the Genome Dataset [32] and the Contagio Mobile website[6]. Moreover to test the BRIDEMAID capability of not being deceived by obfuscation, we have added two composition malware [9], whose performed misbehavior belongs to the class of SMS-Trojan. A family of polymorphic malware is instead present in the Contagio Mobile dataset: `OpFake`. This malware, in fact, generates at run time different IP addresses to reach the server from which it will download the malicious payload.

[5]Dataset and extracting tool available at: https://github.com/MarcelloLins/GooglePlayAppsCrawler

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

B. Detection Results

All apps have been analyzed both from the static analysis module of BRIDEMAID and from the dynamic one. As discussed, the rational of joining these two approaches is the attempt to reach an higher detection accuracy, considering that both approaches are more effective against different behavioral classes. For comparison, the malicious apps have been also classified through the VirusTotal service. Table I details the detection results on the described set of 2974 malicious apps.

The first column from left represents the behavioral classes discussed in Section II, the second one lists the overall samples, whilst in the third and fourth column there are the number of samples recognized as malware respectively by static and malware analysis. The unified detection result is then expressed in the fourth column, where a malicious app is considered as detected if at least one between static and dynamic approach detects it. The last column reports instead the results for the analysis of VirusTotal. For this column, an app has been considered as malicious if at least half plus one of the AntiVirus report the app as malicious. All results are reported both for malware families and single samples. In Table I The malware are divided according to the previously discussed behavior-based classification, reporting also in a single set, those malware which mixes behaviors of two classes (Hybrid). Moreover we detail the results on a family of polymorphic malware (OpFake) and of composition malware [9], whose behavior belongs to the class of SMS Trojan.

As shown in Table I, the dynamic approach is globally more accurate than the static one, being able to detect an higher number of malware. In fact, the dynamic approach has two main advantages: (i) the multi-level analysis, which analyzing at the same time features coming from different levels, is able to detect a larger number of anomalies. Notwithstanding, the dynamic approach is not effective in detecting malware belonging to the botnet and spyware families. The reason has to be found in the behavior of these malicious apps, which, if observed at run time, is not easily distinguished from the behavior of genuine apps that interact with Internet, sending out and/or receiving data [26]. However, this behavior is easily detected through n-grams (static) analysis, which analyzing the code similarity with the one of known malware, is more effective than the dynamic approach. On the other hand, it is possible to see that the static approach is not able to detect composition malware. In fact, since the malicious code is not present originally inside the app and it is downloaded at runtime, the static analysis does not find known malicious ngrams. The dynamic approach, instead is effective in detecting the misbehavior, since the analysis is performed at run-time. In fact, the dynamic analysis performed by BRIDEMAID is particularly resilient to any code obfuscation mechanism, since this analysis focuses on the observed behaviors, which is not affected by obfuscation. Hence, joining the dynamic and static approach effectively brings an improvement in detection accuracy.

Table I
DETECTION RESULTS FOR ANALYZED MALICIOUS APPS.

Malware Type	Families	Samples	Static		Dynamic		BRIDEMAID	VirusTotal
			Fam	Sam	Fam	Sam		
Botnet	2	7	1	2	0	0	2	7
Installer	6	406	3	236	6	406	406	400
Ransomware	3	30	2	11	3	30	30	30
Rootkit	13	543	10	436	13	543	543	541
SMS Trojan	40	1295	33	771	40	1295	1295	1276
Spyware	38	231	38	231	21	161	231	220
Trojan	5	23	5	20	2	19	20	22
Hybrid	14	243	10	189	14	243	243	243
Composition (SMS Trojan)	1	2	0	0	1	2	2	0
Polymorphic (SMS Trojan)	1	14	1	14	1	14	14	14
Total	123	2794	103	1910	101	2713	2784	2753
Accuracy			68,4 %		97,2 %		99.7%	98%

In fact, BRIDEMAID reports an overall detection accuracy of 99.7%, which is 2.5% more accurate than the standalone dynamic approach and 31% more accurate than the standalone static analysis. Moreover, BRIDEMAID is more accurate (1.7%) than VirusTotal, which is not able to detect those malware whose signature is still not known in the antivirus databases. In fact, we can see that VirusTotal is ineffective against the composition malware, which are instead detected by BRIDEMAID. Being based on known signatures, VirusTotal is, in fact, not effective against Zero Day attacks differently from BRIDEMAID, which exploiting computational intelligence and hybrid white-list/black-list approaches, is more resilient to obfuscation techniques and generally able to detect new threats.

To evaluate the False Alarm Rate, BRIDEMAID has been used to classify the set of 9804 genuine apps previously described. Table II schematically reports the amount of apps wrongly classified as malicious and the False Positive (or Alarm) Rate (FPR). The FPR is null or

Table II
FALSE ALARMS AND FALSE POSITIVE RATE.

Total	Static	Dynamic	VirusTotal
9804	0	22	0
FPR	0	0.2%	0

negligibly low for both the components of BRIDEMAID and for VirusTotal. In particular, the n-grams analysis does not report any false positive, having thus the same advantage of a signature-based detector. The dynamic analysis part, instead reports 22 false positives, i.e. a FPR of 0.2%. Hence, an app is considered malicious in average every 500 hundreds analyzed apps. For more details, concerning the false positive measurement for the dynamic analysis, we point the interested reader to [26].

V. RELATED WORK

In the realm of static and dynamic analysis, further techniques have been recently proposed for detecting Android malware.

Droid Detective [15] discriminates an Android application by using a technique based on permission combination. The evaluation with a dataset of 1260 malware and 741 benign produces a detection rate respectively of 96% and 88% for malware and benign recognition. Liu and Liu [16] propose another permission-based approach: they extract requested and used permissions and make combinations of them to build a J48 classifier to test their dataset containing 28548 benign and 1563 malicious applications. Their evaluation obtains a precision equal to 89.8%. Our method, performing a dynamic analysis, is robust against obfuscation techniques that alter the application permission. Authors in [7] investigate whether frequencies of $n$grams of opcodes are effective in detecting Android malware, evaluating their method using 11120 applications, 5560 of which are malware belonging to several different families. They obtain an accuracy of 97% on the average, whereas perfect detection rate is achieved for more than one malware family. In contrast our method, whether the static analysis fails to discriminate a malicious behavior, performs a deep scanning while the application under analysis is running. Yerima et al. [31] present Bayesian classification models obtained from static analysis. They extract 20 features from 2000 application (1000 malware and 1000 trusted) to build the models, obtaining a precision rate equal to 94.4%. Our framework in addition to features extracted through static analysis, performs also dynamic analysis. In Fazeen and Dantu [12] propose a framework to identify potential Android malware applications by extracting the intention and the permission requests. They evaluate the solution using a dataset consisting of 1730 benign applications and 273 malware samples, obtaining an accuracy of 89% in detecting malicious samples, otherwise we experiment our framework on 15k Android real-world applications. Authors of reference [20] focus on permissions for a given application and examine whether the application description provides any indication for why the application needs a permission. They implemented a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description, achieving a average precision of 82.8%, and a recall of 81.5%. AutoCog [21] assesses description-to-permission fidelity of applications using NLP techniques to implement a learning-based algorithm to relate description with permissions. On an evaluation of eleven permissions, they achieve an average precision of

92.6% and an average recall of 92%. Our method is based on a set of features based on the structural characteristic of the application, including the permission request. Authors in [6] propose a method to detect Android malware based on three metrics, which evaluate the occurrences of a reduced subset of system calls, a weighted sum of a subset of permissions required by applications, and a set of combinations of permissions. In their experiment a sample of 200 malicious apps and 200 benign apps are considered; a 74% precision in the identification of malware is obtained.

Researchers in [8] explore the possibility to detect Android malware using system calls sequences. They use Machine Learning to automatically learn these associations; then they exploit them to detect Android malware. Experiments on 20 000 execution traces of 2000 applications (1000 of them being malware belonging to different malware families), performed on a real device, shows a detection accuracy of 97%.

The detection method presented in [14] uses data gathered by an application log and a recorder of a set of system calls related to management of file, I/O and processes. A physical device with a modified Android 2.1 is used for the experiments and 230 applications, in greater part downloaded from Google Play, were considered; among them, the method is able to detect 37 applications which steal some kind of personal data, 14 applications which execute exploit code, and 13 destructive applications.

Several resources are monitored by the tool proposed in [27], Andromaly, which considers: touch screen, keyboard, scheduler, CPU load, messaging, power, memory, calls, operating system, network, hardware, binder, and leds. The results are obtained using 40 benign applications and four malicious samples developed by the authors.

A similar approach is proposed in [13], where feature selection is performed on a set of run-time features related to network, SMS, CPU, power, process information, memory, and Virtual memory. Results are obtained by considering 30 benign and 5 malicious applications.

In reference [29] authors present a system designed to evaluate the maliciousness of Android applications combining static and dynamic analysis, where results of static analysis are used to guide dynamic analysis and extend coverage of executed code. They extract application permissions using static analysis and API call using dynamic ones. They evaluate their solution on 20 samples belonging to 8 malware families.

In our framework we consider a more extensive set of features in order to identify more accurately Android malware, with particular regards to the attack identification, and we evaluate an extended dataset of 2800 malicious applications and 10k genuine applications.

VI. Conclusion

Android malware is becoming more and more aggressive and attackers everyday develop malicious software able to easily evade the current antimalware technologies. So far, many different methods have been developed in order to detect mobile malware, based on either static or dynamic analysis. In this paper we propose a framework merging the two approaches, the first one based on the extraction of opcode $n$gram (i.e., based on static analysis), while the second one relying on dynamic analysis, i.e. extracting features when the application is running. The combination of two different approaches in a single framework allow us to obtain an accuracy in Android malware detection equal to 99,7%, overcoming the current signature-based antimalware technologies.

References

[1] Google play. https://play.google.com/store?hl=it, last visit 20 April 2015.

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

[3] Abdelfattah Amamra, Chamseddine Talhi, and Jean-Marc Robert. Smartphone malware detection: From a survey towards taxonomy. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 79–86. IEEE, 2012.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[5] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, Nadia Tawbi, et al. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001(184-189):79, 2001.

[6] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. A classifier of malicious android applications. In *Proceedings of the 2nd International Workshop on Security of Mobile Applications, in conjunction with the International Conference on Availability, Reliability and Security*, 2013.

[7] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 333–340. IEEE, 2015.

[8] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM, 2015.

[9] Gerardo Canfora, Francesco Mercaldo, Giovanni Moriano, and Corrado Aaron Visaggio. Composition-malware: building android malware at run time. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 318–326. IEEE, 2015.

[10] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. Mobile malware detection

using op-code frequency histograms. In *International Conference on Security, and Cryptography (SECRYPT)*, 2015.

[11] Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi, Andrea Saracino, and Daniele Sgandurra. Evaluating the trust of android applications through an adaptive and distributed multi-criteria approach. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1541–1546. IEEE, 2013.

[12] M. Fazeen and R. Dantu. Another free app: Does it have the right intentions? In *2014 Twelfth Annual Conference on Privacy, Security and Trust (PST)*, pages 282–289, 2014.

[13] Hyo-Sik Ham and Mi-Jung Choi. Analysis of android malware detection performance using machine learning classifiers. In *ICT Convergence (ICTC), 2013 International Conference on*, pages 490–495, Oct 2013. doi: 10.1109/ICTC.2013.6675404.

[14] Takamasa Isohara, Keisuke Takemori, and Ayumu Kubota. Kernel-based behavior analysis for android malware detection. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1011–1015. IEEE, 2011.

[15] Shuang Liang and Xiaojiang Du. Permission-combination-based scheme for android mobile malware detection. In *International Conference on Communications*, pages 2301–2306, 2014.

[16] Xing Liu and Jiqiang Liu. A two-layered permission-based android malware detection scheme. In *International Conference on Mobile Cloud Computing, Service, and Engineering*, pages 142–148, 2014.

[17] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[18] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: a study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy*, pages 692–708. IEEE, 2015.

[19] J. Oberheide and C. Mille. Dissecting the android bouncer. In *SummerCon*, 2012.

[20] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *22nd USENIX Security Symposium*, pages 527–542, 2013.

[21] Z. Qu, V. Rastogi, X. Zhang, and Y. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *21st ACM Conference on Computer and Communications Security*, pages 1354–1365, 2014.

[22] Rahul Ramachandran, Tae Oh, and William Stackpole. Android anti-virus analysis. In *Annual Symposium on Information Assurance & Secure Knowledge Management*, pages 35–40, June 2012.

[23] V. Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, Jan 2014. ISSN 1556-6013. doi: 10.1109/TIFS.2013.2290431.

[24] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon:evaluating android anti-malware against transformation attacks. In *ACM Symposium on Information, Computer and Communications Security*, pages 329–334, May 2013.

[25] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[26] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2016. ISSN 1545-5971. doi: 10.1109/TDSC.2016.2536605.

[27] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": A behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, February 2012. ISSN 0925-9902. doi: 10.1007/s10844-010-0148-x. URL http://dx.doi.org/10.1007/s10844-010-0148-x.

[28] Michael Spreitzenbarth, Florian Echtler, Thomas Schreck, Felix C. Freling, and Johannes Hoffmann. Mobilesandbox: Looking deeper into android applications. In *28th International ACM Symposium on Applied Computing (SAC)*, 2013.

[29] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.

[30] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *Communications Surveys & Tutorials, IEEE*, 16(2): 961–987, 2014.

[31] Suleiman Y. Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new android malware detection approach using bayesian classification. In *International Conference on Advanced Information Networking and Applications*, pages 121–128, 2013.

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