Effectiveness of Opcode $n$grams for Detection of Multi Family Android Malware

Gerardo Canfora*, Andrea De Lorenzo†, Eric Medvet†, Francesco Mercaldo*, Corrado Aaron Visaggio*
*Dept. of Engineering, University of Sannio, Benevento, Italy
{canfora, fmercaldo, visaggio}@unisannio.it
†Dept. of Engineering and Architecture, University of Trieste, Trieste, Italy
{andrea.delorenzo, emedvet}@units.it

***Abstract*—With the wide diffusion of smartphones and their usage in a plethora of processes and activities, these devices have been handling an increasing variety of sensitive resources. Attackers are hence producing a large number of malware applications for Android (the most spread mobile platform), often by slightly modifying existing applications, which results in malware being organized in families.**

Some works in the literature showed that opcodes are informative for detecting malware, not only in the Android platform. In this paper, we investigate if frequencies of $n$grams of opcodes are effective in detecting Android malware and if there is some significant malware family for which they are more or less effective. To this end, we designed a method based on state-of-the-art classifiers applied to frequencies of opcodes $n$grams. Then, we experimentally evaluated it on a recent dataset composed of 11120 applications, 5560 of which are malware belonging to several different families.

Results show that an accuracy of 97% can be obtained on the average, whereas perfect detection rate is achieved for more than one malware family.

I. Introduction

In recent years, mobile phones have become the main computing and communication devices. With the increasing number of capabilities these devices have been acquiring, they represent now the most used way to access to the web and cloud resources. The growth rate in new mobile malware is far greater than the growth rate of new malware targeting PCs [8].

New kinds of malware continuously emerge at a very fast pace, refining more and more the method of the attack and the system for obtaining a tangible gain (money, most of times) from the attack. A malware that is plaguing devices in recent days is the ransomware [5], which encrypts data stored on the device and holds it for ransom. The information is then released only after the victim pays the required amount, in bitcoin.

Protecting smartphones is getting more and more important, as these devices are used for accessing sensitive resources: for instance, with the diffusion of the bring-your-device policy [3], enterprise infrastructures will likely become a sensible target for mobile malware. Moreover, the smartphones will be the access point for controlling intelligent houses and cars, making electronic payments, and operating with the personal bank account.

In the mobile threat landscape, malware writers are focusing on the Android platform. This is not surprising: Android holds 84.4% of the total market share [4] in smartphones and tablets, and Gartner [7] shows that the worldwide sales of smartphones to end users totaled 301 million units in the third quarter of 2014. A recent survey [2] reports that Android in 2014 was the favourite target for mobile threats with 294 new families and variants discovered (trojan represents the main malware type).

The mechanisms employed by attackers to diffuse malware can be grouped basically in three categories [40, 41]: repackaging, attack upgrade, and drive-by downloads.

With repackaging, malware authors locate and download popular applications, disassemble them, enclose malicious payloads, re-assemble and then submit the new applications to official and/or alternative Android markets. Users could be vulnerable by being enticed to download and install these infected applications.

A second technique, the so-called update attack, was introduced to make harder the detection of malicious payload. Specifically, it may still repackage popular applications, but instead of enclosing the payload as a whole, it only includes an update component which will fetch or download the malicious payloads at runtime. As a result, static scanning of host applications to capture the malicious payloads may fail.

The third technique applies the traditional drive-by download attacks to mobile. Though they are not directly exploiting mobile browser vulnerabilities, they are essentially enticing users to download "interesting" or "feature-rich" applications.

Unfortunately, current solutions to protect users from new threats are still inadequate [1]. Current antimalware are mostly signature-based: this approach requires that the vendor be aware of the malware, in order to identify the signature and send out updates regularly. Signatures have traditionally been in the form of fixed strings and regular expressions.

Using signature-based detection, a threat must be widespread for being successfully recognized. In addition to this, there exist several techniques to allow the mobile malware to evade signature detection [34, 33], for instance trivial changes in the code are usually enough, e.g., the variables renaming into the malware code, as a study demonstrated [26].

In the meantime, simple forms of polymorphic attacks (i.e., malware that mutates at each infection) targeting Android platform have already been seen in the wild [13].

There is another problem affecting the ability to detect malware on Android platform. Antimalware software on

desktop operating system has the possibility of monitoring the file system operations. In this way, it is possible to check whether some applications assume a suspicious behavior: for example, if an application starts to download malicious code, it will be detected immediately by the antimalware responsible for scanning the disk drive. Android does not allow an application to monitor the file system: any application can only access its own disk space; resource sharing is allowed only if expressly provided by the developer of the application: this allows applications to download updates and run the new code without any control by the operating system and by antimalware.

Google, with the introduction of Bouncer [27], tried to mitigate the problem but attackers write malware which becomes increasingly aggressive and is able to evade easily the mechanism. Bouncer executes the application in a sandbox for a fixed-time window before publish it on the official market [6]: it is clear that if the malware action happens over this interval time Bouncer can not detect the malicious event.

As empirical experience reveals, attackers use to modify some existing malware, by adding new behaviors or merging together parts of different existing malware codes. This explains also why Android malware is usually grouped in *families*: in fact, given this way of generating Android malware, the codes belonging to the same family share common parts of code and behaviors.

Starting from these considerations, it urges to study new techniques in order to mitigate the problem.

In this paper, we investigate whether short sequences of opcodes (i.e., *opcode $n$grams*) are informative for detecting Android malware. In particular, we focus on the current scenario in which partitions of the malware exist within which the applications share common parts of code. We focus on opcodes because they are closely related to the application code. Indeed, several works in literature showed that opcodes frequency can discriminate a malware software from a trusted one. Bilar [14] obtained evidence that malware opcodes frequency distribution deviates significantly from trusted applications. While Han et al. [21] show that malware classification using instructions frequency can be a useful technique for speeding up malware detection. Rad and Masrom [31] used opcodes for detecting obfuscated versions of metamorphic viruses. The sequences of opcodes could represent a sort of signature of the malware, which is always available and easily extractable from the application to be analyzed. Moreover, as this is a static technique, the analysis has the advantage to be fast and easy to implement.

We designed a method for classifying Android applications as malware or trusted. Our method is built using state-of-the-art classifiers and operates on frequencies of opcode $n$grams extracted from the applications. We experimentally applied the method to a dataset composed of 5560 real mobile malware grouped in 179 families and 5560 mobile trusted applications. We varied method parameters in order to investigate about the method effectiveness with respect to:1) how many consecutive opcodes should be a $n$gram consists, and 2) how many different $n$grams should be considered.

The technique proposed in this paper seems to be much more effective than the existing techniques of static analysis discussed in literature, as we obtained performances that are significantly better than those obtained by the other techniques, that is 96.88% in the recognition of malware. The detection rate among the most spread Android malware families range from 88.6% to 100%.

The paper proceeds as follows: Section II discusses related work; Section III describes and motivates our detection method; Section IV illustrates the results of experiments; finally, conclusions are drawn in the Section V.

II. Related work

In this section we review related literature in two areas: use of opcodes to detect malware; and approaches specific to the Android platform.

Five recent papers investigate the effectiveness of opcodes for detecting malware or characterizing malware.

Bilar [14] proposes a detection mechanism for malicious code through statistical analysis of opcodes distributions. This work compares the statistical opcodes frequency between malware and trusted samples, concluding that malware opcode frequency distribution seems to deviate significantly from trusted applications. Han et al. [21] show that malware classification using instructions frequency can be a useful technique for speeding up malware detection. The major weakness of their method is the rate of false positives. In this paper we investigate the effectiveness of sequences of opcodes, instead of opcodes frequency, in detecting malware.

In references [31, 32] the histograms of opcodes are used as a feature to find whether a file is a morphed version of another. Using a threshold-based method, authors of [31] correctly classify different obfuscated versions of metamorphic viruses; in reference [32] the authors obtain a 100% detection rate using a dataset of 40 malware instances of NGCVK family, 40 benign files and 20 samples classified by authors as other virus files. Compared with our technique, this two works cope with metamorphic malware, while our domain of investigation is Android malware.

Jerome and colleagues [22] proposed a detection mechanism relying on opcode sequences combined with machine learning techniques. They obtain lower performances of detection than our technique and with sequences longer than bi-grams.

Santos et al. [35] propose opcode $n$grams to detect malware using a dataset composed by 1000 malware and 1000 trusted computer applications. They conclude that using 2gram the detection ratio is quite low, achieving a maximum value of 69.66%, thus 2-grams do not seem to be appropriate for malware detection. They achieved best results for detection ratio using 4grams, getting a maximum detection ratio of 91.25%. For the following n values, detection ratio is lower than for $n = 4$, however, the second best results are achieved with $n = 8$.

Also Liangboonprakong and colleagues [24] perform a study on the effectiveness of using $n$gram to discriminate malicious computer applications. They extract four different sizes of $n$grams (with $n \in \{1, 2, 3, 4\}$) and study three classification models (decision tree, artificial neural network, and support vector machine). Using a malware dataset of 12 199 binary files, grouped into 10 families, they obtain the best result in terms of accuracy with 4grams (96.64%).

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

Canfora et al. [15] propose a method for detecting mobile malware based on three metrics, which evaluate: the occurrences of a specific subset of system calls, a weighted sum of a subset of permissions that the application requires, and a set of combinations of permissions. They obtain a precision of 74% using a balanced dataset composed by 200 trusted and 200 real malware applications.

Droid Detective [23] 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 [25] 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 28 548 benign and 1563 malicious applications. Their evaluation obtains a precision equal to 89.8%. The latter technique shows a rate of false positive too high, while the former produce performances that are much poorer than ours.

Sarma et al. [36] investigate the possibility of using both the permissions an application requests, the category of the application, and which permissions are requested by other applications in the same category in order to inform users about the risks of installing a mobile application.

Arp et al. [11] propose a method to perform a static analysis of Android applications based on features extracted from the manifest file and from the disassembled code (suspicious API calls, network addresses and other). Their approach uses support vector machines to produce a detection model, and the dataset used is composed by 5560 malware applications and 123 453 trusted ones obtaining a detection rate equal to 93.9%.

Yerima et al. [39] 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%.

AndroSimilar [17] aims to find regions of statistical similarity starting from the `.dex` files. Authors obtain an accuracy of 72.3% using a dataset of 101 malicious applications.

DroidLegacy [16] classifies Android malware extracting families signatures with a precision rate of 87% from their dataset formed by 1052 malicious applications and 48 benign ones.

Apposcopy [19] identifies class of Android malware using a semantic-based approach, it uses static taint analysis and a call graph inter components; authors evaluate their solution with 1027 malware obtaining an accuracy of 90%.

DroidDolphin [38] performs a static and a dynamic analysis in order to extract features from network access, api calls, achieving a prediction accuracy of 86.1% with a balances dataset composed by 32 000 trusted and 32 000 malicious applications using an SVM classifier. The api calls trace requires the application instrumentation.

Fazeen and Dantu [18] 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 potential malware samples.

Peng et al. [29] introduce the notion of risk scoring and risk ranking derived by the number of permissions requested by an application. They use probabilistic generative models for risk scoring schemes.

Authors of reference [28] 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 [30] 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%.

III. Detection method

We consider a binary classification problem in which an input application a has to be classified as malware or trusted. We propose a supervised classification method for solving this problem in which the features are the frequencies of opcode sequences in the application.

Our method 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 malware or trusted. 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.

A. 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 to extract from the `.apk` the `.dex` file, which is the compiled application file of a (Dalvik Executable); then, with the smali[1] tool, we disassemble the application `.dex` file and obtain several files (i.e., smali classes) which contains the

[1] https://code.google.com/p/smali/

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 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.

B. 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 $n = 5$, 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 proceed as follows. We first compute the average frequencies $\bar{f}_M(o)$ and $\bar{f}_T(o)$ for each $n$gram $o \in \mathcal{O}$ 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 $n$gram 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 $n$grams 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 $n$gram o_x for which another $n$gram 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. For instance, suppose that the $n$gram $o_x = (\texttt{const}, \texttt{iput}, \texttt{move})$ exhibits a high relative difference $d(o_x)$; then, we want to avoid considering the information corresponding to the frequency of $o_y = (\texttt{const}, \texttt{iput})$ if it also exhibits a high relative difference.

Finally, we retain in $\mathcal{O}'$ only the remaining $k < h$ $n$grams 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 $n$grams 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 two classifiers: Support Vector Machines (SVM) and Random Forest (RF). We chose these classifiers because they have proven to be effective in a large set of application scenarios [20]. For SVM we used a Gaussian kernel with the cost $c = 1$, whereas for RF we set $n_{\text{tree}} = 500$.

C. Classification phase

The classification phase consists of determining if an application a is malware or trusted, 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 $\{\text{malware}, \text{trusted}\}$.

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.

IV. Experimental evaluation

A. The Dataset

We built a dataset composed of 5560 trusted and 5560 malware Android applications: we denote with A'_T and A'_M the two partitions of the dataset, respectively.

The trusted applications were automatically collected from Google Play [6], by using a script which queries an unofficial python API [9] to search and download applications from Android official market. The applications retrieved were among the most downloaded from different categories (call & contacts, education, entertainment, GPS & travel, internet, lifestyle, news & weather, productivity, utilities, business, communication, email & SMS, fun & games, health & fitness, live wallpapers, personalization) were downloaded from Google Play [6] (and then controlled by Google Bouncer [27]) from July

Family	Inst.	Attack	Activation	Apps
FakeInstaller	s	t,b		925
DroidKungFu	r	t	boot,batt,sys	667
Plankton	s,u	t,b		625
Opfake	r	t		613
GinMaster	r	t	boot	339
BaseBridge	r,u	t	boot,sms,net,batt	330
Kmin	s	t	boot	147
Geinimi	r	t	boot,sms	92
Adrd	r	t	net,call	91
DroidDream	r	b	main	81

Table I
NUMBER OF SAMPLES FOR THE TOP 10 FAMILIES WITH INSTALLATION DETAILS (STANDALONE, REPACKAGING, UPDATE), KIND OF ATTACK (TROJAN, BOTNET) AND EVENTS THAT TRIGGER MALICIOUS PAYLOAD.

2014 to September 2014, while malware applications of different nature and malicious intents (premium call & SMS, selling user information, advertisement, SMS spam, stealing user credentials, ransom) from Drebin Dataset [11, 37].

We analysed the trusted dataset with the VirusTotal service [10]. This service run 52 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.

Malware dataset is also partitioned according to the *malware family*: each family contains samples which have in common several characteristics, like payload installation, the kind of attack and events that trigger malicious payload [40]. Table I shows the 10 malware families with the largest number of applications in our malware dataset with installation type, kind of attack and event activating malicious payload.

We briefly describe the malicious payload action for the top 10 populous families in our dataset.

1) The samples of *FakeInstaller* family have the main payload in common but have different code implementations, and some of them also have an extra payload. FakeInstaller malware is server-side polymorphic, which means the server could provide different `.apk` files for the same URL request. There are variants of FakeInstaller that not only send SMS messages to premium rate numbers, but also include a backdoor to receive commands from a remote server. There is a large number of variants for this family, and it has distributed in hundreds of websites and alternative markets. The members of this family hide their malicious code inside repackaged version of popular applications. During the installation process the malware sends expensive SMS messages to premium services owned by the malware authors.
2) *DroidKungFu* installs a backdoor that allows attackers to access the smartphone when they want and use it as they please. They could even turn it into a bot. This malware encrypts two known root exploits, exploit and rage against the cage, to break out of the Android security container. When it runs, it decrypts these exploits and then contacts a remote server without the user knowing.
3) *Plankton* uses an available native functionality (i.e., class loading) to forward details like IMEI and browser history to a remote server. It is present in a wide number of versions as harmful adware that download unwanted advertisements and it changes the browser homepage or add unwanted bookmarks to it.
4) The *Opfake* samples make use of an algorithm that can change shape over time so to evade the antimalware. The Opfake malware demands payment for the application content through premium text messages. This family represents an example of polymorphic malware in Android environment: it is written with an algorithm that can change shape over time so to evade any detection by signature based antimalware.
5) *GinMaster* family contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and send to a remote website, as well as install applications without user interaction. It is also a trojan application and similarly to the DroidKungFu family the malware starts its malicious services as soon as it receives a BOOT_COMPLETED or USER_PRESENT intent. The malware can successfully avoid detection by mobile anti-virus software by using polymorphic techniques to hide malicious code, obfuscating class names for each infected object, and randomizing package names and self-signed certificates for applications.
6) *BaseBridge* malware sends information to a remote server running one ore more malicious services in background, like IMEI, IMSI and other files to premium-rate numbers. BaseBridge malware is able to obtain the permissions to use Internet and to kill the processes of antimalware application in background.
7) *Kmin* malware is similar to BaseBridge, but does not kill antimalware processes.
8) *Geinimi* is the first Android malware in the wild that displays botnet-like capabilities. Once the malware is installed, it has the potential to receive commands from a remote server that allows the owner of that server to control the phone. Geinimi makes use of a bytecode obfuscator. The malware belonging to this family is able to read, collect, delete SMS, send contact informations to a remote server, make phone call silently and also launch a web browser to a specific URL to start files download.
9) *Adrd* family is very close to Geinimi but with less server side commands, it also compromises personal data such as IMEI and IMSI of infected device. In addiction to Geinimi, this one is able to modify device settings.
10) *DroidDream* is another example of botnet, it gained root access to device to access unique identification information. This malware could also downloads ad-

ditional malicious programs without the user's knowledge as well as open the phone up to control by hackers. The name derives from the fact that it was set up to run between the hours of 11pm and 8am when users were most likely to be sleeping and their phones less likely to be in use.

B. Experimental procedure and results

We present here the results of a set of experiments we performed in order to assess the effectiveness of our proposal. The experimental procedure was as follows. We built the learning sets A_T and A_M by randomly choosing the 90% of the applications in A'_T and A'_M. The remaining 10% of the applications were used as testing set. We performed the learning phase described in section III-B using A_T and A_M to obtain a classifier C.

After the learning phase, we applied C to each application in the testing set and we measured the accuracy of the classifier, i.e., its detection rate of malware and trusted applications. We repeated the above procedure 5 times, every time by changing the composition of A_T and A_M.

We experimented with different values for k and n, with n varying from 1 to 5 and k in 25–2000. Since the number of different opcodes is 252, we limited the experiment with $n = 1$ to a maximum k equal to 250. We used 2 different kind of classifier—i.e., SVM and Random Forest. We always used $h = 5000$ for the features selection.

Table II reports the results obtained training both an SVM based and a Random Forest based classifier. The table shows the accuracy on training and testing with all the combinations of n and k values. The results are averaged over the 5 repetitions. It emerges that we obtain the best results with $n = 2$ and $k = 1000$. Besides, the Random Forest classifier is better than the SVM one and it reached an accuracy of 96.88%.

Moreover, the table shows that, in order to achieve the same accuracy value, greater values of n need greater values of k. This aspect is consistent with the findings of [22]. A possible interpretation of this finding is that $n$grams greater than 2 may be too specific and thus the classifier tends to overfit. Using larger values of k can reduce the overfit. It is important to note, however, that using of greater values of k and n may make the approach unfeasible in some scenarios.

In order perform the experiments, we used a machine equipped with a 6 core Intel Xeon E5-2440 (2.40 GHz) and 32 GB of RAM. The time spent to perform the features selection with $n = 2$ is about $2063.4\,\mathrm{s}$, where most of the time is needed to compute the $n$grams. This time grows linearly with n. Moreover, the training of an SVM classifier with $n = 2$ and $k = 1000$ took about $239.4\,\mathrm{s}$ whereas a Random Forest classifier took about $1054.8\,\mathrm{s}$. These values are averaged over the 5 repetitions.

Table III shows the detection rate for the 10 families which are most represented in our dataset: the figures appear promising. In particular, we obtain a malware detection rate close to or greater than 90% for most

		SVM		Random Forest	
k	n	Training	Testing	Training	Testing
25	1	85.36	81.77	99.51	93.51
	2	84.78	85.52	87.79	86.64
	3	81.46	85.52	84.75	84.27
	4	81.39	82.77	83.26	83.15
	5	79.25	80.15	80.04	80.27
50	1	86.35	84.39	99.93	93.51
	2	86.90	88.51	91.23	90.14
	3	83.61	84.27	87.29	86.77
	4	82.48	83.15	85.25	84.77
	5	79.55	80.65	81.53	82.02
100	1	89.47	87.14	99.97	93.63
	2	89.26	90.64	94.53	93.38
	3	86.37	86.27	88.79	87.52
	4	83.37	82.90	85.93	85.02
	5	84.55	84.52	85.58	85.02
250	1	93.30	90.89	99.99	95.13
	2	92.99	91.89	97.49	95.13
	3	88.65	86.77	91.15	89.39
	4	85.00	84.89	87.32	86.77
	5	85.25	84.27	87.75	86.02
500	1	-	-	-	-
	2	94.62	92.38	97.54	95.51
	3	90.28	89.64	92.10	90.51
	4	87.40	86.77	90.51	89.39
	5	86.37	84.89	89.78	88.01
750	1	-	-	-	-
	2	95.14	93.01	97.44	96.38
	3	90.78	89.89	93.15	91.14
	4	87.91	87.14	91.39	90.39
	5	86.29	86.14	89.87	88.39
1000	1	-	-	-	-
	2	96.35	94.26	97.35	**96.88**
	3	91.18	90.01	93.49	90.51
	4	88.32	87.64	91.82	90.39
	5	86.79	86.52	89.97	88.51
2000	1	-	-	-	-
	2	95.83	94.13	97.42	94.63
	3	92.60	91.01	95.62	93.01
	4	92.57	89.89	95.89	93.51
	5	91.42	95.67	90.26	94.13

Table II
RESULTS IN TERMS OF ACCURACY (%) ON TRAINING AND TESTING

	Detection Rate			
	SVM		Random Forest	
Family	μ	σ	μ	σ
Malware-FakeInstaller	92.65	2.22	89.71	5.38
Malware-Plankton	88.57	7.80	91.43	2.01
Malware-DroidKungFu	93.23	2.83	88.37	4.79
Malware-GinMaster	91.18	4.91	91.18	3.48
Malware-BaseBridge	93.10	2.97	89.66	2.47
Malware-Adrd	100.00	0.00	100.00	0.00
Malware-Kmin	90.01	3.61	100.00	0.00
Malware-Geinimi	99.28	1.91	100.00	0.00
Malware-DroidDream	100.00	0.00	100.00	0.00
Malware-Opfake	91.67	3.81	94.45	1.53
Malware-Average	94.69	3.35	94.69	2.56
Trusted	93.83	2.53	99.07	0.96

Table III
MEAN AND STANDARD DEVIATION VALUES OF THE DETECTION RATE (%) ON DIFFERENT FAMILIES, WITH $n = 2$ AND $k = 1000$

families. We also highlight that the ability to recognize a malware is similar using SVM or Random Forest, but the second clearly outperforms the first in recognizing trusted applications. This high value is really interesting in a scenario in which it is important to generate a low number of false positives.

V. Concluding remarks and future work

Since previous works showed that opcodes are informative for discriminating a malware from a trusted application, we here investigate the effectiveness of a method operating on opcodes sequences in recognizing the malware targeting Android platform. The experimentation revealed that the sequences of opcodes are a very effective method for detecting Android malware, as this technique produced an accuracy of 96.88%. Moreover, we found that the best accuracy of classification can be obtained by considering just bigrams (i.e., $n = 2$): in that condition, our method needs to take into account 1000 opcodes which, depending on the specific scenario considered, may make feasible the implementation of our method.

Metamorphic malware could escape the proposed technique, but at the moment there are no samples of metamorphic malware in the wild for the Android platform. However polymorphic malware, able to change shape over time so to evade detection by antivirus, is represented in our dataset by Opfake family (613 samples) with a high detection rate: 94% using the Random Forest algorithm.

As future work we are going to compare the performance of opcodes sequences with that of system calls sequences in detecting malware. System calls sequences represent another form of malware signature at a lower level of abstraction, thus in many facets they could provide an information to describe malware that is complementary to that obtained by the sequences of opcodes. Additionally we would apply the opcodes sequences to track the philogenesys of malware in order to characterize which are the known malware an unknown malware descends from. Finally, despite the fact that our findings seem to suggest that long sequence of opcodes are not so informative, it could be interesting to explore the possibility of automatically building pattern-like signatures of opcodes from examples and use corresponding frequencies for detection of malware: indeed, the inference of patterns from examples for security purposes has already been explored (e.g., for intrusion detection [12]).

References

[1] On the effectiveness of malware protection on android. http://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/deutsch/042013-Technical-Report-Android-Virus-Test.pdf, last visit 20 April 2015.

[2] Mobile threat report. https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf, last visit 20 April 2015.

[3] Gartner predicts by 2017, half of employers will require employees to supply their own device for work purposes. http://www.gartner.com/newsroom/id/2466615, last visit 20 April 2015.

[4] Smartphone os market share, q3 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, last visit 20 April 2015.

[5] Update: Mcafee: Cyber criminals using android malware and ransomware the most. http://www.infoworld.com/article/2614854/security/update--mcafee--cyber-criminals-using-android-malware-and-ransomware-the-most.html, last visit 20 April 2015.

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

[7] Gartner. http://www.gartner.com/newsroom/id/2944819, last visit 20 April 2015.

[8] Kindsight security labs malware report - q4 2013. http://www.tmcnet.com/tmc/whitepapers/documents/whitepapers/2014/9861-kindsight-security-labs-malware-report-q4-2013.pdf, last visit 21 January 2015.

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

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

[11] 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.

[12] Alberto Bartoli, Simone Cumar, Andrea De Lorenzo, and Eric Medvet. Compressing regular expression sets for deep packet inspection. In *Parallel Problem Solving from Nature–PPSN XIII*, pages 394–403. Springer, 2014.

[13] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware. In *European Institute for Computer Antivirus Research Annual Conference*, 2006.

[14] D. Bilar. *Opcodes as predictor for malware*. International Journal of Electronic Security and Digital Forensics, 2007.

[15] 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.

[16] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *ACM SIGPLAN on Program Protection and Reverse Engineering Workshop*, 2014.

[17] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal. Androsimilar: robust statistical feature signature for android malware detection. In *International Conference on Security of Information and Networks*, pages 151–159, 2013.

[18] 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.

[19] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: semantics-based detection of android malware through static analysis. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587, 2014.

[20] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research*, 15(1):3133–3181, 2014.

[21] K.S. Han, B. Kang, and E.G. Im. Malware classification using instruction frequencies. In *RACS'14, International Conference on Research in Adaptive and Convergent Systems, pp. 298-300*, 2014.

[22] Q. Jerome, K. Allix, R. State, and T. Engel. Using opcode-sequences to detect malicious android applications. In *IEEE International Conference on Communication and Information Systems Security Symposium*, pages 914–919, 2014.

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

[24] C. Liangboonprakong and O. Sornil. Classification of malware families based on n-grams sequential pattern features. In *8th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 777–782, 2013.

[25] 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.

[26] F. Mercaldo and C. A. Visaggio. Evaluating malwares obfuscation techniques against antimalware detection algorithms. http://www.researchgate.net/profile/Corrado_Aaron_Visaggio/publication/274249693_Evaluating_malwares_obfuscation_techniques_against_antimalware_detection_algorithms/links/551950e80cf273292e71475f.pdf, last visit 24 April 2015.

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

[28] 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.

[29] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *CCS'12, 19th ACM Conference on Computer and Communications Security, pp. 241-252*, 2012.

[30] 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.

[31] B.B. Rad and M. Masrom. *Metamorphic Virus Variants Classification Using Opcode Frequency Histogram*. Latest Trends on Computers (Volume I), 2010.

[32] B.B. Rad, M. Masrom, and S. Ibrahim. Opcodes histogram for classifying metamorphic portable executables malware. In *ICEEE'12, International Conference on e-Learning and e-Technologies in Education, pp. 209-213*, 2012.

[33] 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.

[34] 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.

[35] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas. N-grams-based file signatures for malware detection. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 317–320, 2009.

[36] B. Sarma, N. Li, C. Gates, R. Potharaju, and C. Nita-Rotaru. Android permissions: A perspective combining risks and benefits. In *SACMAT'12, Symposium on Access Models and Technologies, pp. 13-22*, 2012.

[37] 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.

[38] Wen-Chieh Wu and Shih-Hao Hung. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In *Conference on Research in Adaptive and Convergent Systems*, pages 247–252, 2014.

[39] 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.

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

[41] Yajin Zhou and Xuxian Jiang. Android malware, springerbriefs in computer science, 2013.