Using Opcode-Sequences to Detect Malicious Android Applications

Quentin Jerome, Kevin Allix, Radu State and Thomas Engel
Interdisciplinary Center for Security Reliability and Trust
University of Luxembourg
4 rue Alphonse Weicker, Luxembourg L-2721
Email: firstname.name@uni.lu

***Abstract*—Recently, the Android platform has seen its number of malicious applications increased sharply. Motivated by the easy application submission process and the number of alternative market places for distributing Android applications, rogue authors are developing constantly new malicious programs. While current anti-virus software mainly relies on signature detection, the issue of alternative malware detection has to be addressed. In this paper, we present a feature based detection mechanism relying on opcode-sequences combined with machine learning techniques. We assess our tool on both a reference dataset known as Genome Project as well as on a wider sample of 40,000 applications retrieved from the Google Play Store.**

***Keywords*—Android malware, opcode-sequences, machine learning**

I. Introduction

Over the last year, Android became the most used mobile Operating System around the world with more than 500 million devices already activated and around 1.3 million activations every single day [1]. This context makes Android attractive for users, developers and also the attackers who can develop and distribute malware easily. While anti-virus vendors do not agree on the market shares of malware owned by the Android OS, they acknowledge that this is the favourite target of rogue authors to spread mobile malware[2] [3]. Google reacted in setting up Google Bouncer, which scans applications before submission. However, according to a Karspersky security bulletin[4], no significant change has been observed. Researchers [1] have found ways to bypass the Google Bouncer in fingerprinting the Android emulator used by the service. In order to thwart those threats, anti-viruses vendors adapted their detection mechanisms to Android applications. Since it uses signature based detection, this approach is designed to catch only known threats [2]. Detecting Android malware is not an easy task. Dynamic approaches must take into account the multi-entry points issue due to the component-based paradigm of Android, whereas static approaches must deal with known obfuscation techniques[5]. In this paper, we propose a static approach combining opcode-sequences and machine learning techniques. To the best of our knowledge no previous approach tackled Android malware detection using this technique. We sum up here the contributions of this paper:

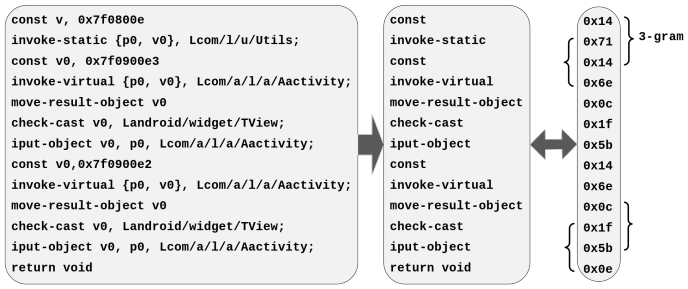

Figure 1. A method translated into a 3-grams vector

- We present an approach based on opcode-sequences that we assessed on a reference Android malware dataset available for the research community;
- We propose a realistic assessment on a snapshot of the Google Play Store - retrieved in the first six months of 2012 – in order to test the validity in a real life scenario. ;
- We compare our approach with several well-known anti-virus products;
- We give access to a dataset of applications that our tool detected as malicious while anti-virus packages did not.

In the section II we present our approach and continue then with the results on a reference dataset in section III. We review the related works in section IV and concluded the paper in the section V.

II. Approach

A. Feature Extraction

Known feature sets that have already been used in the past to detect malicious programs: n-grams [4], opcodes [5], Android permissions combined with Control Flow Graphs [6] and

[1] http://news.cnet.com/8301-1035_3-57510994-94/google-500-million-android-\devices-activated/: accessed on 2013-03-20

[2] http://thehackernews.com/2013/03/google-f-secure-can-say-that-anything.html accessed on 2013-03-20

[3] http://www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012 accessed on 2013-03-20

[4] http://www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012 accessed on 2013-03-20

[5] http://www.dexlabs.org/blog/bytecode-obfuscation accessed on 2013-03-20

several others. Finding the feature set that generalizes the most our observable is the most challenging task. Opcode-sequences that we will refer as k-grams, have already proven their efficiency to classify Windows binaries [7], [8], [9]. However, we opted for a slightly different approach based on opcode-sequence occurrence, introduced further. It is worth noting that we are using opcodes without their related operands. The reason is that operands can be easily changed in altering register indexes. In addition, this can be done without affecting the control flow of execution. Opcodes are extracted from the *classes.dex* file and must be translated into opcode-sequences. Figure 1 depicts how a method is translated into opcode-sequences of length three.

B. Overall Architecture

The classification mechanism is split into two parts, the first aims at building the model according to the machine learning algorithm and the second uses this model as input to classify unknown applications. Since our approach relies on supervised learning, we need only labelled instances to build a model. To build a tool able to classify Android applications we process as follows:

- Collect all the possible k-grams in all the set of applications as an initial feature set
- Apply a selection algorithm in order to determine the most relevant features;
- Create a model using these relevant feature set
- Use this model to classify unknown applications.

Instead of using the weighted frequency of opcode-sequences, as used in [7], [8], [9], we collect binary occurrences of k-grams. In using binary count, we characterize the minimal functionalities required by a program to function properly. On the contrary, if extracting weighted opcode-sequence frequencies, the whole structure of a program is represented. The advantage of using binary count is that the total number of opcode-sequences is not required.

C. Classification Mechanism

Machine-learning-based detection has already been used for detecting malicious Android programs [10], [6]; nevertheless, as far as we know, none of these approaches was assessed on a wide dataset as ours. In order to find the best suited algorithm, we tested some well-known implementation of machine learning algorithm such as libsvm [11] or C 5.0[6]. As a consequence, we opted for a linear implementation of SVM – Support Vector Machine – classification known as liblinear. According to [12], this implementation is adapted to process both a large amount of instances and features.

We used a smart feature selection so that only significant features remain to build the model. The initial number of features depends on the k parameter that we choose for opcode-sequences. We could be tempted to evaluate the maximum of possible k-grams to $M_{th} = N^k$ where $N = 224$ is the number of permitted opcodes. However, in a real scenario this number varies due to opcode semantics. Hence, for $k = 5$ we could expect to count about 564 billions different k-grams but we observed $M_{obs} = 5,998,223$ on a set of 40,000 applications. We used a well-known feature selection based on the information gain computed for each feature. It aims at computing the information brought by each feature compared to the information brought by the labels. The formal definition of the information gain for a feature $f \in F$, where F is the set of all features, is :

$$IG(Ex, f) = H(Ex) - H(Ex|f) \quad (1)$$

$$H(Ex|f) = \sum_{v \in vals(f)} \frac{|\{x \in Ex | val(x,f)=v\}|}{|Ex|} \quad (2)$$

$$\cdot H(\{x \in Ex | val(x, f) = v\}) \quad (3)$$

where Ex represents the set of all instances to study, $val(x, f)$ is the value of the attribute $f \in F$ of the instance x and H is the Shannon's entropy function.

Once computed for each feature, we can rank opcode-sequences according to their information gain. This ranking is used afterwards to reduce our feature set in keeping only the most significant features and thereby reduce the information loss.

III. Experimental Results

We run our experiments on the Genome Project's dataset introduced in [13]. We choose to use this dataset as a ground truth since it contains only malicious programs that were checked manually by analysts. We are making our work comparable with other approaches since the dataset is openly available.

A. Experiments Description

As no goodware dataset has been published, we are running our experiments on ten different subsets of 1,246 Android applications randomly picked from the Android market. We use ten different datasets in order to work with balanced datasets and thus avoid overfitting problems. We define a goodware dataset as being a collection of applications that does not contain any instance of malicious applications present in the malicious set. We define the metrics that need to be computed to evaluate classification performances. The true positives rate TP_{rate} and true negatives rates TN_{rate} are defined as $TP_{rate} = \frac{TP}{TP+FN}$ and respectively $TN_{rate} = \frac{TN}{TN+FP}$. These metrics represent percentages of well classified instances over each class – malign and benign in our case . Assuming that *positive* stands for malign applications and *negative* for benign instances, the recall is defined as $Recall_{malign} = TP_{rate}$ and $Recall_{benign} = TN_{rate}$. Another interesting measure is the precision where $Precision_{malign} = \frac{TP}{TP+FP}$ and $Precision_{benign} = \frac{TN}{TN+FN}$. Precision measures the likelihood of good prediction while recall measures probability of good retrieving. Finally, F-measure is defined as F-measure$_c = \frac{2 \cdot Precision_c \cdot Recall_c}{Precision_c + Recall_c}$, where c denotes the class considered. We choose the average F-measure to evaluate tool performances because it takes into account both *Recall* and *Precision* while

[6]http://www.rulequest.com/see5-info.html

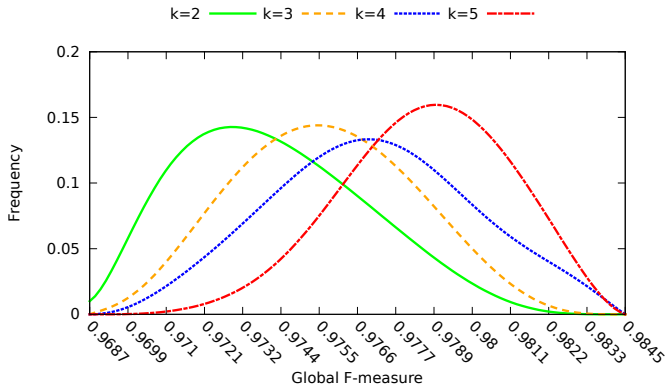

Figure 2. F-measure distribution over datasets

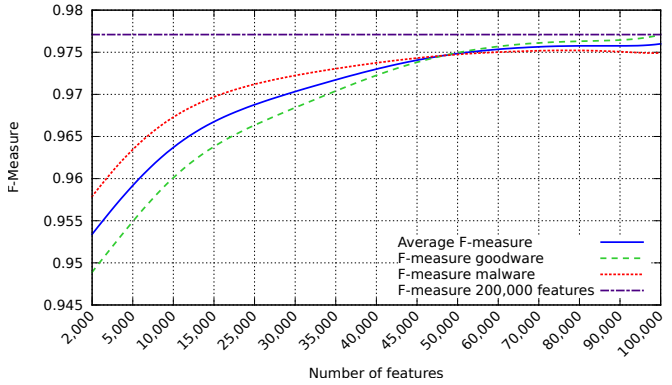

Figure 3. F-measure variation for dataset number one

classification accuracy only considers the *Recall* values of both classes.

In order to find the best value for k, we run ten-fold cross-validation experiments on each combination of malware/goodware datasets. In addition, we limited the number of features to the 200,000 highest information-gain features. We needed to do this for performance reasons since for $k = 5$ we extracted $N = 1,305,511$ different features for only one malware/goodware combination. Figure 2 shows the results for different combinations of malware/goodware datasets. These distributions have been plotted considering the average F-measure over classes for each fold of each run. We can see that the best results are obtained for $k = 5$. Another good property for $5 - grams$ is the squeezed shape of the distribution. This means that the results are more steady among the several runs. The choice of the benign dataset has a small impact on classification results. Indeed, we observed a low standard deviation of the average F-measure computed per dataset. The standard deviation among ten datasets is F-measure$_{stdev} = 0.0027$ for $k = 5$. This means that our work can be easily compared with other approaches using the Genome Project's dataset.

Figure 2 shows that the length $k = 5$ provides the best results in term of classification. The classification capabilities are not affected by the choice of the benign dataset. The best classification performances that can be expected are the same as those obtained with 200,000 features. As a consequence, we minimize the number of features in targeting an average F-Measure as close as possible to F-Measure$_{target} = 0.9771$. Figure 3 depicts the F-measure obtained among classes as well as the average when the number of features varies. This curve was obtained with ten-fold cross-validation. Good classification results with only few features can be achieved. Indeed with 2,000 features, we are able to get good classification performances since F-Measure > 0.95. We also observe that classification results become steady around 70,000 features. This number of features is sound since F-Measure$_{70,000} = 0.9759$, which is the closer value to F-Measure$_{target} = 0.9771$. Figure I shows the the ten most significant opcode-sequences. Some sequences are specific to a certain class of applications, where opcode-sequences two,six,nine and ten are seen only in malware.

Table I. TOP TEN OPCODE-SEQUENCES

0x13 0x6e 0x6e 0x6e 0x0c	0x12 0x12 0x3c 0x0e 0x22
const/16	const/4
invoke-virtual	const/4
invoke-virtual	if-gtz
invoke-virtual	return void
move-result-object	new-instance
0x1c 0x6e 0x6e 0x0c 0x6e	0x71 0x54 0x62 0x6e 0x28
const-class	invoke-static
invoke-virtual	iget-object
invoke-virtual	sget-object
move-result-object	invoke-virtual
invoke-virtual	goto
0x1d 0x54 0x6e 0x0a 0x39	0x6e 0x28 0x20 0x38 0x1f
monitor-enter	invoke-virtual
iget-object	goto
invoke-virtual	instance-of
move-result	if-eqz
if-nez	check-cast
0x21 0x01 0x35 0x46 0x1a	0x0d 0x07 0x6e 0x28 0x9c
array-length	move-exception
move	move-object
if-ge	invoke-virtual
aget-object	goto
const-string	sub-long
0x0c 0x1a 0x6e 0x0a 0x33	0x16 0x31 0x3d 0x74 0x0e
move-result-object	const-wide/16
const-string	cmp-long
invoke-virtual	if-lez
move-result	invoke-virtual/range
if-ne	return void

We trained our detection tool on a labelled dataset obtained from VirusTotal[7]. We evaluate the model on a training set made of a snapshot of the Android Market retrieved during the first six months of 2012. We further compare our results with the results given by some anti-virus software available in the VirusTotal engine. As far as we know, no previous academic work concerning Android malware detection gave access to tools or datasets. This is the reason why we do not compare our approach with previous works done in this area of research.

B. Dataset Introduction

The dataset used in this experiment contains three parts. We gathered two datasets from VirusTotal, one containing 25,476 malware and another containing 15,670 benign applications. We also got access to 42,062 Android applications downloaded from the official Google Play Store market. In order to build a relevant model, we define hereafter which applications we

[7]https://www.virustotal.com

Table II. DATASETS SUMMARY

	Benign	Malign
Retrieved from VirusTotal		
Total	15,670	25,476
Unique by SHA 256	15,670	25,476
After filtering	12,905	11,960
Retrieved from Google Play		
Total	35,657	6,405
unique by SHA 256	35,700	6,135
Overlap with files in model	389	31

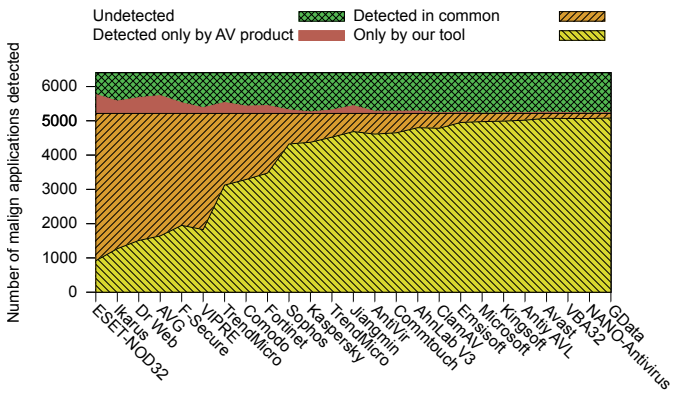

Figure 4. Comparison with top 25 anti-virus packages

Table III. CONFUSION MATRIX

$Rec._{ben.}$	$Rec._{mal.}$	$Prec._{ben.}$	$Prec._{mal.}$	F-measure
97.33%	81.40%	83.95%	96.83%	0.8931

consider as malicious for the training phase. Among a preliminary study on this dataset we observed many detections such as adware like Airpush, Leadbolt or Wooboo[8]. However, when we looked at such adware description, none of it states clearly that this family can be fully considered as malware. To deal with this issue, we deleted each application detected as adware. To reduce noise in the training set we choose to use only files that raised at least three alarms out of 46, after adware filtering. On our benign dataset we applied a filtering step as well, consisting in removing applications being signed with a certificate that signed at least one application detected as malware. We started the filtering process from respectively 15,670 benign and 22,476 malign instances. After filtering, 12,905 instances from the benign class and 11,960 malign applications remained. To avoid balancing issues between the two datasets, we opted for a down-sampling strategy regarding the benign class. Table II summarizes both the training and testing datasets.

C. Results

The *precision malicious* means that an unknown application scanned by our tool will have 96.83% of likelihood to be also detected as such by at least one anti-virus product embedded in VirusTotal. Similarly, 83.95% of apps classified as benign by our tool will remain undetected by VirusTotal. As the total number of malign and benign instances are different, we weighted the results in order to compute the precision metrics. The true negatives rate respectively *Rec.ben.* in table III means that when VirusTotal does not raise any alert, we classify applications as benign in 97.33% of cases. The true positives rate *Rec.mal* corresponds to the likelihood of agreeing with VirusTotal when it raises at least one alert for an application. We conclude that in spite of the same knowledge basis as VirusTotal, our tool has different results. This is another drawback when we want to scale up such a detection mechanism. The fundamental problem is that we do not really now if VirusTotal is right in 100% of cases. Indeed when we assume an application as malicious if it raises at least one alert, we are exposed to the false positives rate of each anti-virus package. As comparing directly our tool to VirusTotal is biased by anti-virus software false positives rates, we propose to compare it directly to each software embedded in VirusTotal. Figure 4 shows a comparison, in term of number of detections, between our approach and the top 25 anti-virus packages available in VirusTotal. We define the top 25 anti-virus packages as being the 25 software that detected the most malicious applications among our testing set. For the chart, these were ordered by detection rate in descending order. We can consider as first example the comparison with ESET-NOD32[9].

- Left to right stripes (yellow): our tool detected 918 applications that ESET-NOD32 did not;
- Right to left stripes (orange): 4296 applications were detected by both tools;
- No stripes (red): 554 were detected only by ESET-NOD32;
- Crossed stripes (green): 677 were not detected by both tool, but at least once by another anti-virus software.

In spite of its poor classification results when compared with VirusTotal, the tool detects more rogue applications than all anti-virus products taken individually. We can also notice that our approach does not mimicry neither VirusTotal nor a given anti-virus product.

We relied on digital certificates to gather some hints about misclassified applications. When building a classification mechanism we must deal with false positives and false negatives. As we observe in Table III, we have 2,143 misclassified instances when compared with VirusTotal. To get the exact classification results, one should analyse each of them. However, completely reverse engineering Android applications for analysis is a long and tedious process. This is the reason why we used a lightweight method to reduce this amount of applications to analyse and thus avoid reverse engineering. Throughout a preliminary analysis of the datasets, we noticed interesting signature patterns for applications. As depicts Figure 5, certificates signing malware, tend to sign more applications than other certificates do. In this histogram, we plot on the x axis the number of applications signed by a given certificate and on the y axis the frequency of such a signature pattern over the datasets. We rely on a method to

[8]http://www.networkworld.com/news/2012/102212-trendmicro-android-malware-263542.html accessed on 2013-03-21

[9]http://www.eset.com/us/

quickly identify potentially malicious sets of applications. We extracted certificates from all applications presented in Table II. Concerning false negatives, we see on Figure 5 that the signing pattern is closer to the pattern observed for malware. This is not surprising since we discarded adware from our training set. In this false negative set we noticed that only 90 were not adware. However, this is not this set of applications which is problematic since these have already been identified as malicious by analysts.

Table IV. AVERAGE ESTIMATOR FOR DIFFERENT CONFIDENCE INTERVALS

	N	sample mean	sample stdev	CI 99%
benign	8319	1.60	2.92	$1.51 < \mu < 1.68$
malign	5486	4.63	35.12	$3.41 < \mu < 5.85$
false positives	420	1.81	4.10	$1.29 < \mu < 2.33$

Applications that would really need to be analysed are in the false negative set. We searched for applications that were signed by certificates that already signed a malign application. We found that among these 952 apps, 194 were signed by a certificate that had already signed at least one application detected by VirusTotal. Thus, we can isolate this set of applications for a future analysis. In addition, we estimated the average number of applications signed by a given certificate in Table IV. In this table μ estimates the average number of applications signed by certificate, for each population. These estimations confirm that we can consider the signature pattern as being different between malign and benign applications. Knowing this, we propose a method to find applications with a high likelihood of maliciousness. We isolate suspicious certificate according to a threshold $T = f(\mu, \sigma)$ characterizing the number of applications signed. Once suspicious certificates are identified, we can find the applications signed and isolate them. As a result in taking the upper bound of the estimated mean μ with a confidence interval of 99%; we can isolate 128 applications in considering certificates that signed more than 11 applications. This value is obtained in setting the threshold to $T = \mu + 3\sigma$ for the benign population. Considering a normal distribution, over $T = \mu + 3\sigma$ only 0.135% of the population is represented. Hence, in selecting only certificates over this threshold, we isolate only outliers. In using this method, combined with the malicious certificate check, we can divide the set of applications to analyse by three since we isolated 322 applications over 952. Another advantage of this method is that we increase the likelihood of finding malware inside this subset. In studying signature pattern we also observe malicious behaviour that may explain why these applications were detected as malicious by our classification approach.

IV. RELATED WORK

In [15] the authors present JuxtApp, an Hadoop-based approach that extracts k-grams from basic blocks in order to detect similarities between two applications. DroidMoss [16] is another approach dealing with the similarity detection issue in using opcode. Firstly, all opcodes are extracted from applications and then a piecewise hashing is computed to compare applications. In [17] the authors present a dataset of 46 malware ranging from malware, personal spyware and grayware. They also present the current incentives of rogue authors as well as future motivations. The dataset of malware that we use

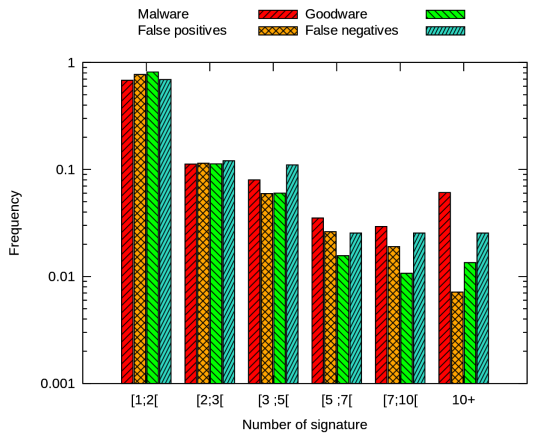

Figure 5. Applications signed by certificate

in this paper has been introduced in [18], this collection gathers 1,260 malicious programs ordered in 49 families. Andromaly, a machine learning-based detection technique, is introduced in [19]. The tool monitors memory use, calls, SMS as well as many other dynamic features. In [20] RiskRanker is presented, this approach aims at detecting applications having suspicious. This is performed through Control Flow Graph analysis as well as in looking for suspicious API use. Another static detection mechanism is presented in [13] where the authors present DroidRanger. This tool uses heuristics extraction and dynamic execution monitoring in order to detect unknown threats. In addition, the approach leverage a signature based detection to identify known threats. A dynamic approach is presented in [21] where the authors proposed a syscall monitoring approach further used to identify malicious signatures. Batyuk *et al.* introduce in [22] a solution aiming at disassembling code and looking at malicious API use. DroidMat [10] extracts information about Intent, API calls and permissions in order to classify applications in using clustering techniques. In [6], the authors use Permissions and Control Flow Graph in order to detect malicious pattern user the One Class SVM algorithm. Two remote analysis approaches are introduced in [23] and [24]. Crowdroid[23], the former approach, firstly collects syscalls on the phone and then it uses these feature on a remote server to detect anomaly. Walldroid[24] is an application based firewall aiming at detecting and blocking communications between smartphones and malicious servers.

In [25], Grace *et al.* observed strange behaviours in several in-app advertisement libraries. Some of them are loading code dynamically, using code obfuscation in using the Java Reflection API, reading SMSs, accessing contacts or even starting GPS. Addressing the same topic, [26] shows that 56% of applications with ads access location through the ad library and that 23% of applications are using less privileges than the advertisement library it embeds.

V. FUTURE WORK AND CONCLUSION

We have presented in this paper an efficient approach to classify Android applications and thus to detect Android malware. Our approach has the same limits that any supervised machine learning approach. It will not detect completely

different malware. The evasion that could completely fool our approach is advanced bytecode-level obfuscation - for instance, in transforming the bytecode into a semantically equivalent program - since it would alter directly opcode-sequences. There are several commercial solutions capable of obfuscating Android applications. Nevertheless, as far as we know, only Proguard the default Android obfuscator is freely available. Our approach is not sensible to the Proguard obfuscation process since only method names and class names are modified by this obfuscation tool. Although advanced obfuscation techniques have been reviewed in [14] we do not know any tool capable of doing this automatically. We provide an open access to the datasets used for the experiments[10] presented in this paper. In addition, we give access to our dataset of applications identified as malicious in this paper and not detected by VirusTotal. This access can be given in contacting us by email.

REFERENCES

[1] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.

[2] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, no. 2, pp. 6:1–6:42, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/2089125.2089126

[3] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-26, Feb 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-26.html

[4] I. Santos, Y. Penya, J. Devesa, and P. Bringas, "N-grams-based file signatures for malware detection," in *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS*, 2009, pp. 317–320.

[5] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, "Unknown malcode detection using opcode representation," *Intelligence and Security Informatics*, pp. 204–215, 2008.

[6] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Intelligence and Security Informatics Conference (EISIC), 2012 European*. IEEE, 2012, pp. 141–147.

[7] I. Santos, F. Brezo, J. Nieves, J. K Penya, B. Sanz, C. Laorden, and P. G Bringas, *Opcode-sequence-based Malware Detection*. The Institute of Electrical and Electronics Engineers, Inc, 2010, vol. 5965, pp. 35–43.

[8] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0020025511004336

[9] I. Santos, B. Sanz, C. Laorden, F. Brezo, and P. G. Bringas, *Opcode-sequence-based Semi-supervised Unknown Malware Detection*, 2011, vol. 6694, pp. 50–57.

[10] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[11] C. Chang and C. Lin, "Libsvm: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, no. 3, p. 27, 2011.

[12] R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin, "Liblinear: A library for large linear classification," *The Journal of Machine Learning Research*, pp. 1871–1874, 2008.

[13] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," *of the 19th Annual Network and*, no. 2, 2012. [Online]. Available: http://www.csd.uoc.gr/~hy558/papers/mal_apps.pdf

[14] S. Patrick, "Code protection in android," Rheinische Friedrich-Wilhelms-Universität Bonn, Germany, Tech. Rep., 2012.

[15] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications."

[16] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[17] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org.proxy.bnl.lu/10.1145/2046614.2046618

[18] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16

[19] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, no. 1, pp. 161–190, 2011. [Online]. Available: http://www.springerlink.com/index/10.1007/s10844-010-0148-x

[20] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 281–294. [Online]. Available: http://doi.acm.org.proxy.bnl.lu/10.1145/2307636.2307663

[21] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," pp. 1011–1015, 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6128277

[22] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," *Malicious and Unwanted Software, International Conference on*, pp. 66–72, 2011.

[23] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," *Science*, pp. 15–25, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2046619

[24] C. Kilinc, T. Booth, and K. Andersson, "Walldroid: Cloud assisted virtualized application specific firewalls for the android os," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE, 2012, pp. 877–883.

[25] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 101–112. [Online]. Available: http://doi.acm.org.proxy.bnl.lu/10.1145/2185448.2185464

[26] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of AsiaCCS*, 2012.

[10] http://secan-lab.uni.lu/~qjerome/downloads/datasets.tar