2019 IEEE International Conference on Cloud Engineering (IC2E)

Detecting IoT Malware by Markov Chain Behavioral Models

Massimo Ficco
Department of Engineering, University of Campania Luigi Vanvitelli, Aversa (CE), Italy.
massimo.ficco@unicampania.it

***Abstract*—Internet of Things (IoT) is become one of the most important technological sector in recent years, and the focus of attention in many fields, including military applications, healthcare, agriculture, industry, and space science, made it very attractive for cyber-attacks. Especially for the wide diffusion of the Adroid platform, the IoT devices are become one of the main targets of malware threats. Considering the great Android market share, it is needed to build effective tools able of detecting zero-day malware. Therefore, several static and dynamic analysis methods have been proposed in the literature. In this work, the sequences of API calls invoked by apps during their execution are modeled by Markov chains, and used to extract features of the apps through the time, needed for malware classification. The considered dataset includes 22K benign applications and 24K malware collected over different shared datasets. Experimental results show that the Markov chain approach detects malware with up to 89% F-measure and outperforms approaches based on API calls frequency.**

***Index Terms*—Internet of Things, Android malware, dynamic analysis, Markov chains.**

I. Introduction

A typical Internet of Things (IoT) scenario can include a wide network of inter-connected smart devices, embedded systems, sensors, mobile devices, connected to Internet and other systems via wireless technologies. These IoT devices and nodes are a valuable target for cyber criminals, whose aim at compromising the security and privacy of IoT data, user's personal information, identity, and financial information [1]. In particular, a huge number of Android based IoT devices, due to Android openness and popularity of this operation system, have become the main targets of malware attacks. Malware are able to perform malicious actions to violate the device integrity, as well as steal in stealthy fashion sensitive data, such as login credentials, contacts, and perform maliciously actions, compromising the system operation, or more simply, subscribing to costly premium services [3]. Million of apps are currently available on third-party market places, which might not perform any malware analysis. Also Google Play Store has hosted malicious apps. During 2016, about 38,000 new malware samples were detected per day [2].

Different static and dynamic approaches have been proposed to extract representative behavioral information of Android apps. Extracted information can be used to define models able to distinguish between benign and malicious apps and identify their belonging class. Static analysis analyzes disassembled code without app execution, in order to capture syntax and semantic information, exploiting extracted API sequences [4], calls graph [5], and opcodes [6], useful to infer a database of signatures needed to identify known attacks. On the other hand, it is affected by the use of obfuscation and encryption techniques. Moreover, it cannot detect the dynamically loaded malicious code. Dynamic analysis approaches monitor the behaviors and actions performed by apps during their execution in virtual environments, including system calls [7], network activity [8], file operations and modification of registry records [9], but it is very computationally expensive.

Several authors have proposed dynamic analysis methods, by exploiting the sequence and/or frequency of API calls performed by an app, in order to model the app behavior. These methods assume that malware may use different calls in a different order with respect to the benign apps [10], and the sensitive API calls may be invoked more frequently in the malicious apps [11], [12]. On the other hand, the sensitive API calls constitute only a small portion of the entire Android API calls [13].

This work exploits the sequence of the API calls invoked by the app to build the sequence of state transitions performed by the considered app. Such dynamic behavioral model is represented as a Markov chain, which is widely used in the literature for modeling sequence of states, assuming that the probability of being in a specific state only depends on the previous states [14], [15]. The derived API calls transitions model is used to extract features need to classify apps as malicious or benign by several machine learning algorithms, including Naive Bayes, Decision Tree (J48/C4.5), Random Forest, and Support Vector Machines.

The considered dataset includes 22K benign applications and 24K malware collected over different shared datasets. The test set is composed by different families of malware, with a mix of older and newer apps, in order to verify how the presented methods are robust with respect to the evolution over time of malware. Experimental results show that the Markov chain approach outperforms approaches based on the API call frequency, and detects malware with up to 89% F-measure.

The remainder of the paper is organized as follows: Sec. II presents an overview of the related work. In Sec. III, a background on the adopted Markov chain approach is provided. The proposed approach and the feature extraction are described in Sec. IV. Sec. V reports the experimental results. Finally, Sec. VI presents some considerations and remarks.

978-1-7281-0218-4/19/$31.00 ©2019 IEEE
DOI 10.1109/IC2E.2019.00037

II. Related work

Existing sophisticated malware are malicious code designed to change the form of each instance of software infection, in order to evade pattern matching detection during the security analysis and investigative processes [16]. For example, the polymorphic malware exploit the encryption and appending/pre-pending data to mute the appearance of the code, as well as the metamorphic malware automatically recode itself each time it propagates, for example, by adding different lengths of NOP and loops instructions within the code segments, permuting user registers, modifying static data structure and reordering functions. Moreover, some sophisticated malware can detect the presence of the security analyzers and virtualized detection environments and evade the detection systems. For example, virtual machine based rootkits could conceal their presence by migrating and infecting other operating systems running in the same virtual environment.

Therefore, in order to face the huge increase of Android malware, several static and dynamic analysis detection methods and tools have been proposed by the research community [11], [17]. Static techniques analyze the program-syntax behavior by code reverse engineering, in order to infer a database of signatures useful to identify known attacks. For example, MaMaDROID presents a malware detection system for Android that uses the static analysis to analyze the sequence of abstracted API calls performed by an app, by abstracting API calls to their packages and families [18]. It builds the app behavioral model in the form of a Markov chain.

On the other hand, the majority of new malware are polymorphic. They adopt mutation engine that generates new decryption routines for the code, without change the underlying code. The main effect of polymorphism is that the signature-based detection techniques become ineffective. Moreover, in static analysis, the applications are analyzed by scanning the code instead of executing them, therefore, it can be affected by the use of the obfuscation and encryption techniques, as well as fails to detect malicious code when it is loaded or determined at run-time. Finally, it is not able to capture the run time environment context, the code that is executed more frequently, or effects derived from user input.

Therefore, in the last years, the dynamic approaches are playing an important role in the analysis of the Android malware. Dynamic techniques analyze the run time behavior of the malicious code while it is being executed, to study the semantics of the evasive malware, by exploiting function calls and parameters monitoring, information flow tracking, instruction traces, etc. [19]. For example, DroidAPIminer uses Android API calls frequency for malware classification. The classification algorithm is based on the most common API calls observed during training process [11]. Andromaly is a tool for extracting execution features. Drebin is a similar tool that exploits execution features and uses different classifiers to detect malicious apps [20]. Several machine learning techniques are adopted to classify malware [21]. Moreover, several works have adopting neural network and deep learning for dynamic malware analysis [22].

In order to reduce the computation and memory consumptions due to the huge number of features extracted during the static analysis, Dahl et al. [23] adopt a random projection process to reduce the feature vector of 179K to a resulting vector of 4K elements, analyzed by a deep classifier, which is pre-trained by a Restricted Bolzmann Machine. A similar approach is adopted by Wang at al. [24], in which the source code is processed through a 5-layer Deep Neural Network (DNN) implemented with stacked denoising Autoencoders. Saxe at al. [25] use different sets of static features, converted in a 1024-length binary vectors and classified by a classic DNN. McLaughlin et al. [26] propose the deep convolutional neural network to analyze the raw sequences of opcode extracted from disassembled Android malware. Regarding dynamic analysis, Kolosnjaji et al. [27] use DNN combined with convolutional layers and recurrent layers to analyze the sequence of system calls extracted during the apps execution. Moreover, they use Long Short Term Memory cells to increase malware detection accuracy. David et al. [22] adopt deep learning implemented through stacked denoising autoencoders, in order to generate the signature, which are processed by a Support Vctor Machine to perform the app classification. A more complex set of features are extracted by Xu et al. [28], through DNNs during the static and dynamic analysis, and then, classified through Multiple Kernel Learning.

In this paper, the same Markov Chain based approach used in MaMaDROID has been adopted, but exploiting dynamic analysis system calls instead of static. Moreover, the adopted Markov based detection approach outperforms a similar approach presented in [15].

III. Markov chain based method

This section describes how to build a Markov chain behavioral model of the monitored app from the extracted sequence of API calls.

Markov chain is a memory-less stochastic process, in which the probability of transitioning from a state to another only depends on the current state. According to the Markov chain model, the sequence of API calls can be represented as a graph, in which each node represents the API and the edges connecting one node to another represent the methods invoked by that node. Moreover, each edge is labeled with the probability of that transition.

Specifically, the set of states of the Markov chain is represented by the all possible software calls invoked by the considered app, and denoted as C. The edge between c_i and c_j denotes the probability P_{ij} that the method i invokes the method j:

$$P_{ij} = \frac{O(c_i -> c_j)}{\sum_{c_z=1}^{c_z=n} O(c_i -> c_z)}, \tag{1}$$

where $O(c_i -> c_j)$ represents the number of occurrences of the method c_j after the method c_i, divided by all the occurrences O_{iz} for all methods z in the chain. Finally, the

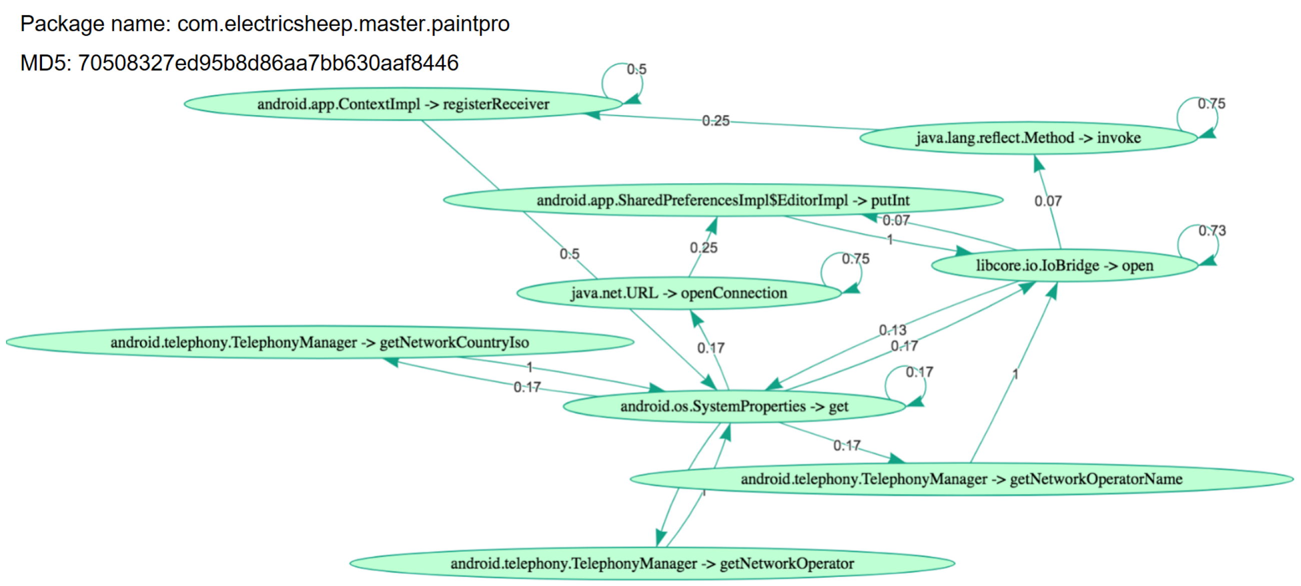

Package name: com.electricsheep.master.paintpro
MD5: 70508327ed95b8d86aa7bb630aaf8446
android.app.ContextImpl -> registerReceiver
java.lang.reflect.Method -> invoke
android.app.SharedPreferencesImpl$EditorImpl -> putInt
libcore.io.IoBridge -> open
java.net.URL -> openConnection
android.telephony.TelephonyManager -> getNetworkCountryIso
android.os.SystemProperties -> get
android.telephony.TelephonyManager -> getNetworkOperatorName
android.telephony.TelephonyManager -> getNetworkOperator
0.5
0.25
0.75
0.07
0.73
0.5
0.25
0.07
1
0.75
0.13
0.17
0.17
0.17
1
0.17
0.17
1
0.17
1

Fig. 1. Markov chain of the com.electricsheep.master.paintpro APK malware

sum of the probabilities associated to all edges from any node is equal to 1:

$$\sum_{c_z=1}^{c_z=n} P_{iz} = 1. \tag{2}$$

The features vector of each app is represented by the probabilities of transitioning from one method to another in the chain. Therefore, this approach can be more robust against attempts to evade signature-based detection methods by inserting useless API calls. In fact, adding calls would not significantly change the transition probabilities P_{ij} between two methods of the app [18].

IV. Features extraction

In order to reproduce the API call patterns generated by each monitored app, two approaches graphically represented by two different Call Graphs have been adopted and compared. In particular, the first approach represents the Call Dependency Graphs by using the Markov chain theory. Fig. 1 shows an example of the Markov chain of the $com.electricsheep.master.paintpro$ APK malware.

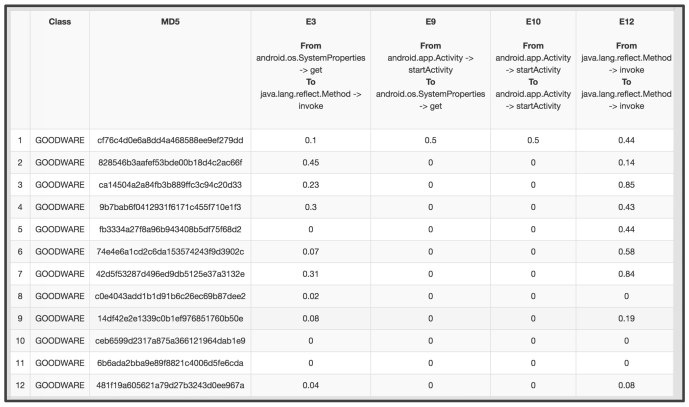
The generated graphs are stored as adjacent matrixs. On the rows and columns are indicated the caller and called methods, respectively, and the probability is represented in the corresponding cell. Moreover, the Principal Component Analysis (PCA) has been applied to perform features selection and reduction of the features space [29]. It has been applied to the extracted feature set in order to select the principal components (i.e., the principal API call transitions), that are a linear combination of the original features. It represents the more significant features that characterize better the analyzed malware. This allows to reduce computation and memory consumptions during the running of machine learning algorithms. In Fig. 2 is reported the Optimized Feature Transition Matrix of a set of malware APKs achieved by the PCA analysis. It is an MxS matrix, in which M is the number of analyzed APKs, and S is the number of API call transitions analyzed with the PCA.

Experimental results have shown that the F-measure obtained using the considered machine learning algorithms and the PCA components is about 2% less than using the full features set.

In the second approach, the API call patterns are represented as a tree, where the root node is the app, and the leaves are the called API methods. The edges represent the number of occurrences of the specific API method. In Fig. 3, the $com.electricsheep.master.paintpro$ APK malware is represented as a calls tree.

V. Experimental results

The dataset used during the experimental evaluation includes malware samples downloaded by the VirusShare [30], Malgenome [31], and Contagio Minidump [32] datasets. Specifically, the exploited VirusShare dataset includes 24K APKs, dating back from Gin. 2013 to Mar. 2014. More recent malware have been downloaded by Malgenome and Contagio including 1.2k and 0.3K samples, respectively. Moreover, about 22k benign apps have been collected from Playdrone dataset [33]. Other 1.2K more recent benign apps have been also obtained by Google Play store, using the Googleplay-api tool [34]. The collected dataset is composed by different families of malware, including Information Steal (IS), Trojan Horse (TH), Escalation of Privileges (EOP), and Premium SMS (SMS), with a mix of older and newer apps.



	Class	MD5	E3 **From** android.os.SystemProperties -> get **To** java.lang.reflect.Method -> invoke	E9 **From** android.app.Activity -> startActivity **To** android.os.SystemProperties -> get	E10 **From** android.app.Activity -> startActivity **To** android.app.Activity -> startActivity	E12 **From** java.lang.reflect.Method -> invoke **To** java.lang.reflect.Method -> invoke
1	GOODWARE	cf76c4d0e6a8dd4a468588ee9ef279dd	0.1	0.5	0.5	0.44
2	GOODWARE	828546b3aafef53bde00b18d4c2ac66f	0.45	0	0	0.14
3	GOODWARE	ca14504a2a84fb3b889ffc3c94c20d33	0.23	0	0	0.85
4	GOODWARE	9b7bab6f0412931f6171c455f710e1f3	0.3	0	0	0.43
5	GOODWARE	fb3334a27f8a96b943408b5df75f68d2	0	0	0	0.44
6	GOODWARE	74e4e6a1cd2c6da153574243f9d3902c	0.07	0	0	0.58
7	GOODWARE	42d5f53287d496ed9db5125e37a3132e	0.31	0	0	0.84
8	GOODWARE	c0e4043add1b1d91b6c26ec69b87dee2	0.02	0	0	0
9	GOODWARE	14df42e2e1339c0b1ef976851760b50e	0.08	0	0	0.19
10	GOODWARE	ceb6599d2317a875a366121964dab1e9	0	0	0	0
11	GOODWARE	6b6ada2bba9e89f8821c4006d5fe6cda	0	0	0	0
12	GOODWARE	481f19a605621a79d27b3243d0ee967a	0.04	0	0	0.08

Fig. 2. Optimized Feature Transition Matrix

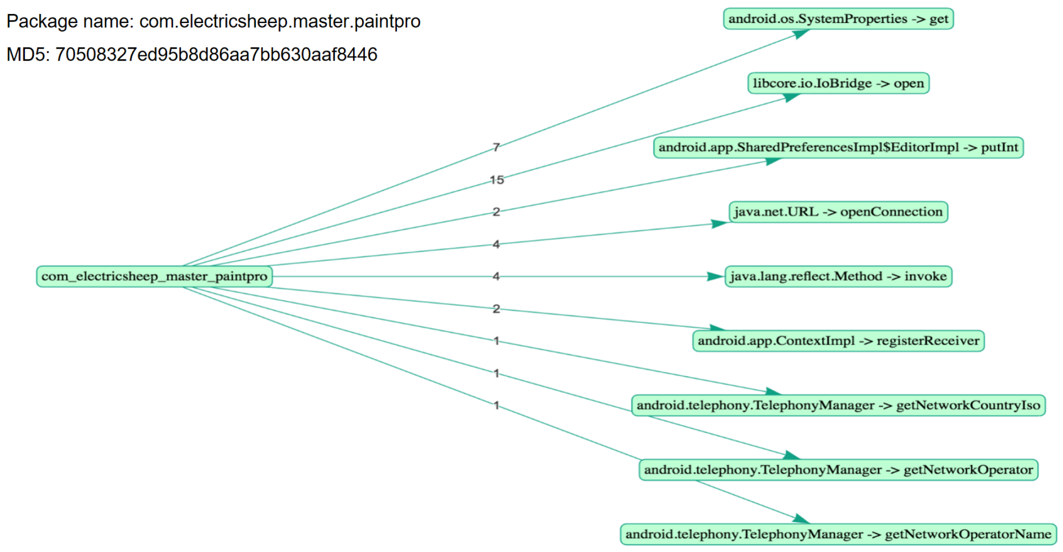

Package name: com.electricsheep.master.paintpro
MD5: 70508327ed95b8d86aa7bb630aaf8446
com_electricsheep_master_paintpro
android.os.SystemProperties -> get
libcore.io.IoBridge -> open
android.app.SharedPreferencesImpl$EditorImpl -> putInt
java.net.URL -> openConnection
java.lang.reflect.Method -> invoke
android.app.ContextImpl -> registerReceiver
android.telephony.TelephonyManager -> getNetworkCountryIso
android.telephony.TelephonyManager -> getNetworkOperator
android.telephony.TelephonyManager -> getNetworkOperatorName
7
15
2
4
4
2
1
1
1

Fig. 3. Calls Tree of the com.electricsheep.master.paintpro APK malware

The dynamic features are extracted using mobSF tool [35]. Each Android APK is executed in the mobSF for a specific period of time and its dynamic behavior is monitored. The extracted system calls are embedded into two vectors space that represent the frequency of call occurrences (NUMCALL) and the transaction probability among components (VAED), which are used to capture the features of the app behavior. The quantitative analysis of the results achieved by the NUMCALL approach shows that the malicious apps invoke specific system calls more frequently than the benign apps, whereas the VAED shows that the sequences of call invocations performed by malware present different patterns with respect to the legitimate applications.

In order to perform a quantitative analysis, the extracted feature vectors are given as input to the machine learning tool, Weka [36]. In particular, four machine learning algorithms have been adopted to perform classification of benign and malicious apps, and compare VAED and NUMCALL results, including: (i) Naive Bayes, (ii) Decision Tree (J48/C4.5), (iii) Random Forest, and (iv) Support Vector Machines (SMO, RBF Kernel). Moreover, results have been validated by means of 10-fold cross validation.

In order to compare the adopted algorithms, the following metrics have been adopted:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + RN}, \tag{3}$$

$$Recall = \frac{TP}{TP + FN}, \tag{4}$$

with

$$TP = True\ Positive,$$
$$TN = True\ Negative,$$
$$FP = False\ Positive,$$

TABLE I
VAED AND NUMCALL ACCURACY

VAED

DATASET	Naive Bayes	Decision Tree (J48)	Random Forest	SVM (SMO)
EOP	92,5	80	87,5	92,5
IS	90,9	86,4	86,4	88,6
PSMS	89,8	84,2	78,9	81,6
TH	84,8	69,6	80,4	87

NUMCALL

DATASET	Naive Bayes	Decision Tree (J48)	Random Forest	SVM (SMO)
EOP	80	87,5	77,5	72,5
IS	79,1	90,9	88,6	88,6
PSMS	84,2	71,9	78,9	78,9
TH	78,4	76,1	84,8	71,7

TABLE II
VAED AND NUMCALL RECALL

VAED

DATASET	Naive Bayes	Decision Tree (J48)	Random Forest	SVM (SMO)
EOP	88,8	66,7	73,3	80
IS	89,5	73,7	78,9	73,7
PSMS	92,3	93	76,9	76,9
TH	87,9	70,4	76,2	75,7

NUMCALL

DATASET	Naive Bayes	Decision Tree (J48)	Random Forest	SVM (SMO)
EOP	91,3	86,7	66,7	68,7
IS	93,5	84,2	78,9	73,7
PSMS	94,1	61,5	69,2	58,5
TH	89,5	70,4	85,7	47,6

$$TN = False\ Negative.$$

For each monitoring approach, Tab. I and Tab. II show the accuracy and recall archived by each classification method, with respect to each dataset used for the validation.

The results show that the best classifier is Naive Bayes, which gives highest accuracy with lowest false positive rate. Moreover, the NUMCALL detection approach, which is computational simplest, provides a better recall, but a lower accuracy. The reason is that, in general, malware perform a high number of calls of a limited set of critical APIs, hence a high recall. On the other hand, several benign apps generate a similar number of calls, but with different transition call patterns, hence, the high number of false negative for the NUMCALL approach, and a high accuracy for the VAED approach.

The standard F-measure metric has been used to measure the accuracy of the compared approaches:

$$F-measure = 2 * \frac{precision * recall}{precision + recall}, \tag{5}$$

where

$$precision = \frac{TP}{TP + FP}. \tag{6}$$

The experimental results show that the VAED approach detects malware with up to 89% F-measure with respect to the NUMCALL approach that achieves 85% F-measure. Moreover, new experiments have been set up, in which an oldest dataset of malware and benign apps have been used as training (i.e., VirusShare from Gin. 2013 to Mar. 2014), and a more recent malware dataset has been used as testing. With this configuration, VAED F-measure drops to 74% when tested on samples two years newest of the training dataset.

VI. CONCLUSIONS AND FUTURE WORKS

In this work has been addressed the malware detection task, modeling the sequences of API calls as Markov chains. Results obtained by experiments performed by using widely used and popular datasets have confirmed the capability of the proposed approach of extracting relevant knowledge from Markov chains representation, which can be used in the malware detection task by observing dynamic behaviors of the apps. In order to reduce the amount of memory and computation consumptions needed to perform classification by machine learning algorithm, a feature selection algorithm has been adopted. It has been compared the Markov chain approach with respectto an approach widely adopted in the literature, which is based on API calls frequently. A large set of Android malware dataset has been analyzed to assess both the accuracy and recall of the achieved classification. Experimental results show that the Markov chain approach detects unknown malware samples with F-measure up to 89%.

In the future work, we intent to investigate and compare the proposed approach with neural network and deep learning approaches.

References

[1] M. Ficco, S. Venticinque, and M. Rak. Malware Detection for Secure Microgrids: CoSSMic Case Study, in Proc. of the *IEEE Int. Conf. on iThings/GreenCom/CPSCom/SmartData*, Jun. 2017, pp. 336-341.

[2] Connect symantec Archives, available at: http://www.symantec.com/connect/blogs/yet-another-bunchmalicious-apps-found-google-play].

[3] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention, in *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1 , Jan.-Feb. 2018, pp. 83-97.

[4] J. Y.-C. Cheng, T.-S. Tsai, and C.-S. Yang. An information retrieval approach for malware classification based on windows API calls, in Proc. of the *Int. Conf. on Machine Learning Cybern. (ICMLC)*, Jul. 2013, pp. 1678-1683.

[5] S. Cesare, Y. Xiang, and W. Zhou. Control flow-based malware variant detection, in *IEEE Trans. Dependable and Sececure Computing*, vol. 11, no. 4, Jul./Aug. 2014, pp. 307-317.

[6] B. Kang, S. Y. Yerima, K. Mclaughlin, and S. Sezer. N-opcode analysis for Android malware classification and categorization, in Proc. of the *Int. Conf. on Cyber Security Protection Digital Services (Cyber Security)*, Jun. 2016, pp. 1-7.

[7] T. Lee, B. Choi, Y. Shin, and J. Kwak. Automatic malware mutant detection and group classification based on the n-Gram and clustering coefficient, in *Journal of Supercomputing*, Dec. 2015, pp. 1-15.

[8] S. Nari and A. A. Ghorbani. Automated malware classification based on network behavior, in Proc. of the *Int. Conf. on Computer Network Communication (ICNC)*, Jan. 2013, pp. 642-647.

[9] G. Cabau, M. Buhu, and C. P. Oprisa. Malware classification based on dynamic behavior, in Proc. of the *18th Int. Symp. on Numerical Algorithms Science Computing (SYNASC)*, Sep. 2016, pp. 315-318.

[10] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on Android, in *IEEE Symposium on Security and Privacy*, May 2015, pp. 915-930.

[11] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android, in Proc. of the *9th Int. ICST Conf. on Security and Privacy in Communication Networks*, 2013, pp. 86-103.

[12] H.Y. Chuang and S.-D. Wang. Machine learning based hybrid behavior models for Android malware analysis, in Proc. of the *9th IEEE Int. Conf. Software Quality, Reliability and Security*, Aug. 2015, pp. 201-206.

[13] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics based detection of Android malware through static analysis, in Proc. of the *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2014, pp. 576-587.

[14] G. Fink. Markov models for pattern recognition: from theory to applications, 2014.

[15] A. Martín, V. Rodríguez-Fernández, and D. Camacho. CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains, in *Engineering Applications of Artificial Intelligence*, vol. 74, Sep. 2018, pp. 121-133.

[16] Chet Hosmer, Polymorphic & Metamorphic Malware, available at: https://www.blackhat.com/presentations/bh-usa-08/Hosmer/BH_US_08_Hosmer_Polymorphic_Malware.pdf [Jen. 2018].

[17] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors, in the *ACM European Workshop on Systems Security (EuroSec)*, April, 2013, pp. 1-6.

[18] MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models, in Proc. of the *24th Network and Distributed System Security Symposium (NDSS 2017)*, Nov. 2017, pp. 1-22.

[19] E. Manuel, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools, in *ACM Computing Surveys*, vol. 44, no. 2, 2012, pp. 1-37.

[20] K. Tam, S.J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: automatic reconstruction of android malware behaviors, in Proc. of the *Symp. on Network and Distributed System Security (NDSS)*, 2015, pp. 1-15.

[21] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. WeissY. Andromaly: a behavioral malware detection framework for android devices, in *Journal of Intellent Informatic Systems*, Vol. 38, no. 1, 2012, pp. 161-190.

[22] O.E. David and N.S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification, in Proc. of the *IEEE Int. Joint Conf. on Neural Networks (IJCNN)*, 2015, pp. 1-8.

[23] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks, in Proc. of the *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 3422-3426.

[24] Y. Wang, W.-D. Cai, and P. Wei. A deep learning approach for detecting malicious javascript code, in *Security and Communication Networks*, vol. 9, no. 11, 2016, pp. 1520-1534.

[25] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. in Proc. of the *10th IEEE Int. Conf. on Malicious and Unwanted Software (MALWARE)*, 2015, pp. 11-20.

[26] N. McLaughlin, et al. Deep android malware detection, in Proc. of the *7-th ACM on Conf. on Data and Application Security and Privacy (CODASPY'17)*, 2017, pp. 301-308.

[27] B. Kolosnjaji, A. Zarras, G. D. Webster, and C. Eckert. Deep learning for classification of malware system call sequences, in *Australasian Conf. on Artificial Intelligence*, vol. 9992, LNCS, 2016, pp. 137-149.

[28] L. Xu, D. Zhang, N. Jayasena, and J. Cavazos. Hadm: Hybrid analysis for detection of malware, in Proc. of the *SAI Intelligent Systems Conf. (IntelliSys)*, 2016, pp. 1037-1047.

[29] Ran He, Bao-Gang Hu, Wei-Shi Zheng, and Xiang-Wei Kong. Robust Principal Component Analysis Based on Maximum Correntropy Criterion, in *IEEE Transactions on Image Processing*, vol. 20, no. 6, 2011, pp. 1485-1494.

[30] Virusshare malware dataset, available at: https://virusshare.com/ (Dec. 2018).

[31] MalGenome malware dataset, available at: http://tinyurl.com/combopx (Dec. 2015).

[32] Contagiominidump malware dataset, available at: https://contagiominidump.blogspot.ca (Mar. 2018).

[33] Playdrone goodware dataset, available at: https://archive.org/details/playdrone-apks (Mar. 2018)

[34] Googleplay-api tool, available at: https://github.com/egirault/googleplay-api [last access: Jen. 2018].

[35] Mobile-Security-Framework-MobSF, available at: https://github.com/MobSF/Mobile-Security-Framework-MobSF [last access: Feb. 2018].

[36] Weka, Open Source Machine Learning Software in Java, available at: https://www.cs.waikato.ac.nz/ ml/weka/ [last access: Feb. 2018].

[37] M. Ficco and F. Palmieri. Introducing fraudulent energy consumption in cloud infrastructures: A new generation of denial-of-service attacks. in *IEEE Systems Journal*, vol. 11, no. 2, 2017, pp. 460-470

[38] G. D'Angelo, F. Palmieri, M. Ficco, S. Rampone. An uncertainty-managing batch relevance-based approach to network anomaly detection. in *Applied Soft Computing Journal*, vol. 36, no. 17, 2015, pp. 408-418.