ESANN 2021 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Online event, 6-8 October 2021, i6doc.com publ., ISBN 978287587082-7.
Available from http://www.i6doc.com/en/.

Robust Malware Classification via Deep Graph Networks on Call Graph Topologies

Federico Errica[1], Giacomo Iadarola[2], Fabio Martinelli[2],
Francesco Mercaldo[2,3] and Alessio Micheli[1]

1- University of Pisa, Department of Computer Science

2- Institute of Informatics and Telematics, National Research Council of Italy

3- University of Molise, Department of Medicine and Health Sciences

Abstract. We propose a malware classification system that is shown to be robust to some common intra-procedural obfuscation techniques. Indeed, by training the Contextual Graph Markov Model on the call graph representation of a program, we classify it using only topological information, which is unaffected by such obfuscations. In particular, we show that the structure of the call graph is sufficient to achieve good accuracy on a multi-class classification benchmark.

1 Introduction

Detecting malicious behavior using static analysis is one fundamental process to protect devices, networks and users' personal data. By looking at how the program is developed, the goal is to find known patterns — textual or statistical — that classify a program as either *trusted* or belonging to a specific *malware* family. As anti-malware companies become better at finding known patterns, so do malware writers that rely on *obfuscation* techniques to elude common pattern checks. There are two main categories of obfuscation: intra-procedural, i.e., it modifies procedural code without changing the interaction with the rest of the program, or inter-procedural, i.e., it alters the structure of the program by also adding *call* or *invoke* statements. While the latter is certainly more difficult to detect, it is also much more delicate to use as some mechanisms (e.g., call-return and parameter passing) may rely on information known only at run-time and can introduce concurrency problems. Instead, intra-procedural techniques are widely used and suffice to fool a number of static code analysis tools [1]. Recent works test their approaches on the most common obfuscation techniques [2] or group them by their magnitude of edits on the code [3]. In this context, we investigate the problem of malware classification from a machine learning perspective, where the program is represented as a *Call Graph* (CG), i.e., a graph where nodes are procedures and edges denote calls to other procedures. Differently from the literature, we consider obfuscation techniques based on their influence on the CG topology.

Many malware detection tasks working on CGs exploit graph-signature, similarity algorithms and graph-kernels [4, 5, 6]. In conjunction with formal

This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215 and by MIUR - SecureOpenNets, EU SPARTA, CyberSANE and E-CORRIDOR projects.

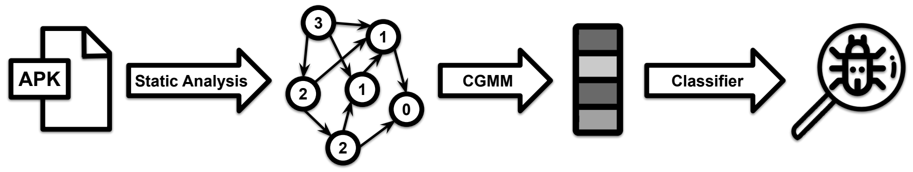

Fig. 1: Given an Android Application Package (APK) w.l.o.g, we apply static analysis to construct a CG, where nodes represent methods and arrows denote how methods are intertwined. For our purposes, the sole node feature we use is the out-degree of each node. Then, the CGMM model transforms the input graph into an embedding that is used for the final classification.

methods, these approaches achieve excellent accuracy, but the analysis is time-consuming and requires domain-level expertise for the temporal logic formulae generation [7]. Instead, most machine learning approaches are adaptive, generally more efficient, and rely on static analysis features included in the graphs, such as opcodes frequencies [8] and control/data dependencies [9]

In this paper, we propose a malware family classification method based solely on the CG topology. This way, it is possible to show that the approach is intrinsically robust to intra-procedural obfuscation techniques. To automatically extract patterns from the CG, we rely on Deep Graph Networks (DGNs); for space reasons, we refer the reader to recent surveys on the topic [10, 11, 12] . In particular, we exploit the Contextual Graph Markov Model (CGMM) [13, 14] to construct graph embeddings that are then fed to a standard machine learning classifier. Note that, while methods exist to certify robustness of DGNs to node perturbations [15], our approach does not need such certificates as it focuses on the structure, i.e., it is robust to any node content perturbation.

2 Methodology

We sketch the overall methodology in Figure 1. For the purposes of this work, we shall define a graph as a tuple $g = (\mathcal{V}, \mathcal{E}, \mathcal{X})$ where $\mathcal{V}$ is the set of entities (or *nodes*), $\mathcal{E}$ is the set of *directed* edges connecting entities, and $\mathcal{X}$ is the set of node features. The feature vector of a node u is referred to with the symbol $\mathbf{x}_u \in \mathcal{X}$. Finally, we define the neighborhood of a node u as $\mathcal{N}_u = \{(v, u) \mid v \in \mathcal{V}\}$.

2.1 Embedding Generation and Classifier

Assuming we have a CG dataset to learn from, we employ CGMM to generate a graph embedding that encodes each CG structural information. To do so, at each layer ℓ, CGMM maximises the likelihood of each graph g conditioned on each node neighbors' posteriors $\mathbf{q}^{\ell-1}_{\mathcal{N}_u}$ computed at the previous layer $\ell - 1$:

$$P(g \mid \{\mathbf{q}^{\ell-1}_{\mathcal{N}_u} \mid u \in \mathcal{V}\}) = \prod_{u\in\mathcal{V}} \sum_{i=1}^{C} P(\mathbf{x}_u|Q_u = i)P(Q_u = i|\mathbf{q}^{\ell-1}_{\mathcal{N}_u})$$

$$P(Q_u = i|\mathbf{q}^{\ell-1}_{\mathcal{N}_u}) \approx \frac{1}{|\mathcal{N}_u|} \sum_{j}^{C} P(Q = i|q = j) \sum_{v\in\mathcal{N}_u} q^{\ell-1}_v(j)$$

where Q_u is the latent categorical variable with C states introduced by marginalization, $P(Q = i|q = j)$ is the probability of transitioning from a neighboring state j to i, and $q^{\ell-1}_v(j)$ refers to the j-th component of the posterior of v computed at layer $\ell - 1$. In this work, $\mathbf{x}_u$ corresponds to the out-degree of node u as done in [14], and therefore the emission distribution $P(\mathbf{x}_u|Q_u = i)$ will be a univariate Gaussian. Using the degree feature is just one the possible choices.

It was shown that this unsupervised training produces useful representations for the subsequent downstream tasks [14]; also, it can significantly accelerate the model selection phase, since graph embeddings need be computed only once and the downstream classifier works on simple vectors. The final graph embedding is the concatenation of the aggregated node posteriors produced at each layer.

3 Experiments

We now describe how we converted a set of Android applications, i.e., `.apk` files in this work, into a CG dataset; nevertheless, provided a static analysis tool is available, it is straightforward to apply this methodology to other environments as well. First, of all, each `.apk` file is decompressed and the Java bytecode is decompiled into Jimple, an intermediate representation language, using the Soot Framework [16]. During decompilation, the code is analyzed to generate a CG[1], where nodes represent methods, i.e., a procedure or function construct, and directed edges denote calls from caller to called nodes, i.e., when an *invoke* or *call* statement is present in the method. Our analysis only considers methods in the application packages, thus discarding calls to library functions or external packages. Notably, the generated CGs do not contain information about the methods statements, e.g., variables, declaration, and dependencies, on the nodes; instead, as already mentioned, we add the out-degree as the sole node feature to be able to train the probabilistic model. Hence, our methodology is intrinsically robust to intra-procedural obfuscations techniques, such as Code Reordering/Removal, Junk code insertion, Instruction substitution, Control Flow modifications, Identifiers and Variables renaming/encryption, and Repacking [1]. Indeed, these obfuscation techniques modify the method's statements but they do not alter the number of *invoke* or *call* statements, so the initial CG is exactly the same as any intra-procedurally obfuscated CG.

Malware samples were collected from the AMD and previous work datasets [17], and the benign samples were downloaded from Google Play. Both the mal-

[1]The Soot transformation code is available at https://github.com/Djack1010/graph4apk

# graphs	**# classes**	**avg $\lvert\mathcal{V}\rvert$**	**avg $\lvert\mathcal{E}\rvert$**	**min deg**	**max deg**	**avg deg**
5669	8	5069	3267	0	618	0.58

Table 1: Dataset statistics. Graphs are large but sparse, and the average degree is low because all calls to external libraries have been removed from the CG.

Model	TR Loss	TR Acc.	VL Loss	VL Acc.	TE Loss	TE Acc.
Baseline	$1.2_{\pm 0.05}$	$55.6_{\pm 0.5}$	$1.1_{\pm 0.01}$	$60.6_{\pm 0.9}$	$1.1_{\pm 0.03}$	$56.7_{\pm 0.5}$
CGMM	$0.01_{\pm 0.01}$	$99.8_{\pm 0.4}$	$0.16_{\pm 0.01}$	$97.9_{\pm 0.2}$	$0.13_{\pm 0.01}$	$96.4_{\pm 0.6}$

Table 2: Malware classification results (mean and standard deviation) on training (TR), validation (VL) and test (TE) sets. We display both the Cross-Entropy loss as well as the multi-class accuracy. Results are averaged over 3 final runs.

ware and the trusted applications were verified with VirusTotal, to ensure either their maliciousness or trustiness. The resulting dataset consists of 5669 samples of real-world malware, split into 8 classes, where one represents the trusted software (1762 samples) and the others stand for different malware families, namely *Airpush* (736 samples), *Dowgin* (1040 samples), *FakeInst* (190 samples), *Kuguo* (879 samples), *Youmi* (959 samples), *Fusob* (73 samples), and *Mecor* (30 samples). Dataset statistics are described in Table 1.

To assess the performance of CGMM on our CG dataset, we split the data according to a stratified hold-out strategy, with 80% of the data for training, 10% for validation and 10% for test.[2] To empirically evaluate the impact of the structure in the dataset, we follow [18] and introduce a structure-agnostic baseline. The baseline applies an MLP to the node features, performs global aggregation and then applies a linear output layer. We performed grid-search model selection for both the baseline and CGMM, with early stopping monitoring the classification accuracy. In the case of the former, we tried: hidden units $\in$ {32,64,128}, 2000 epochs, batch size 128, global aggregation $\in$ {sum, mean}, Adam Optimizer with learning rate $\in$ {0.01, 0.001}, patience $\in$ {50}. For CGMM, instead, we selected the best model across the following configurations: 20 states, layers $\in \{10, 20\}$, 10 EM epochs, posterior version $\in$ {discrete, continuous}, embedding version $\in$ {unigram, unibigram}, global aggregation $\in$ {sum, mean}, batch size 64, 2000 epochs, hidden units $\in$ {32,64,256, 512}, Adam Optimizer with learning rate $\in$ {0.0001} and weight decay $\in$ {0., 0.0005}, and patience 100. We trained CGMM via EM and the neural classifiers using the Cross-Entropy Loss. In the interest of space, we refer to [14] for a thorough explanation of all the hyper-parameters. We relied on the PyDGN library [11] to carry out robust and reproducible experiments.

3.1 Results and Discussion

Results are shown in Table 2. As we can see, the structural variability in the dataset is such that a structure-agnostic baseline cannot accurately classify in-

[2]`https://github.com/diningphil/robust-call-graph-malware-detection`

ESANN 2021 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Online event, 6-8 October 2021, i6doc.com publ., ISBN 978287587082-7. Available from http://www.i6doc.com/en/.

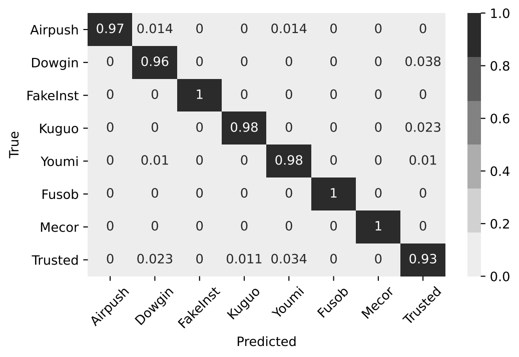

Fig. 2: Row-normalized confusion matrix of CGMM computed on the test set.

stances by merely looking at the out-degree statistics of the graph. Instead, CGMM is able to extract structural patterns that allow the subsequent classifier to achieve a 96.4% accuracy on the test set (and a Macro F1 score of 97.2%). This very good result supports our knowledge that different malware families share detectable topological similarities and hence our hypothesis on the robustness of the approach to intra-procedural obfuscations. In fact, we were able to accurately detect such similarities without relying on non-adaptive procedures, domain expertise, and static analysis' node features that are sensitive to obfuscation techniques. In addition, the confusion matrix of Figure 2 shows how accuracy does not decrease for the most imbalanced classes, e.g., Fusob and Mecor. Rather, the classifier achieves perfect classification on those test samples. Finally, to empirically confirm that the proposed approach is robust to intra-procedural obfuscation methods, we also performed inference on an obfuscated subset of test malwares (261 out of 391, due to intrinsic difficulties in the process, e.g., sometimes the obfuscated code did not compile) using the Code Reordering and Junk Code techniques in [1]. The model achieved a 99.6% accuracy.

4 Conclusions

We have presented a machine learning methodology for malware classification that is intrinsically robust to intra-procedural obfuscation techniques. By classifying program instances based solely on the topology of their call graphs, we have showed that it is possible to get excellent accuracy and F1 score on an imbalanced dataset. In future works, we will extend the evaluation to other malware families, analyse the impact of obfuscation on other classification tools, and assess the robustness to known *inter*-procedural obfuscation techniques.

References

[1] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–9, 2018.

ESANN 2021 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Online event, 6-8 October 2021, i6doc.com publ., ISBN 978287587082-7. Available from http://www.i6doc.com/en/.

[2] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320, 2017.

[3] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[4] Shanhu Shang, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang. Detecting malware variants via function-call graph similarity. In *5th International Conference on Malicious and Unwanted Software*, pages 113–120. IEEE, 2010.

[5] Ammar Ahmed E Elhadi, Mohd Aizaini Maarof, Bazara IA Barry, and Hentabli Hamza. Enhancing the detection of metamorphic malware using call graphs. *computers & security*, 46:62–78, 2014.

[6] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54, 2013.

[7] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Call graph and model checking for fine-grained android malicious behaviour detection. *Applied Sciences*, 10(22):7975–7994, 2020.

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

[9] Akshay Kapoor and Sunita Dhavale. Control flow graph based multiclass malware detection using bi-normal separation. *Defence Science Journal*, 66(2), 2016.

[10] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):25. 18–42, 2017.

[11] Davide Bacciu, Federico Errica, Alessio Micheli, and Marco Podda. A gentle introduction to deep learning for graphs. *Neural Networks*, 129:203–221, 9 2020.

[12] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[13] Davide Bacciu, Federico Errica, and Alessio Micheli. Contextual Graph Markov Model: A deep and generative approach to graph processing. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80, pages 294–303, 2018.

[14] Davide Bacciu, Federico Errica, and Alessio Micheli. Probabilistic learning on graphs via contextual architectures. *Journal of Machine Learning Research*, 21(134):1–39, 2020.

[15] Daniel Zügner and Stephan Günnemann. Certifiable robustness and robust training for graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 246–256, 2019.

[16] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224, 2010.

[17] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Towards an interpretable deep learning model for mobile malware detection and family identification. *Computers & Security*, 105:102198–103012, 2021.

[18] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020.