Robust Malware Classification via Deep Graph Networks on Call Graph Topologies

Federico Errica\textsuperscript{1}, Giacomo Iadarola\textsuperscript{2}, Fabio Martinelli\textsuperscript{2}, Francesco Mercaldo\textsuperscript{2,3} and Alessio Micheli\textsuperscript{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 $x_u \in \mathcal{X}$. 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 $\ell$, CGMM maximises the likelihood of each graph $g$ conditioned on
each node neighbors’ posteriors $q^{\ell-1}_u$ computed at the previous layer $\ell - 1$:

$$
P(g | \{q^{\ell-1}_u | u \in V\}) = \prod_{u \in V} \sum_{i=1}^{C} P(x_u | Q_u = i) P(Q_u = i | q^{\ell-1}_u)
$$

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

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, $x_u$ corresponds to the out-degree of node $u$ as done in [14], and therefore the emission distribution $P(x_u | Q_u = i)$ 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
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.

<table>
<thead>
<tr>
<th># graphs</th>
<th># classes</th>
<th>avg</th>
<th>avg</th>
<th>min deg</th>
<th>max deg</th>
<th>avg deg</th>
</tr>
</thead>
<tbody>
<tr>
<td>5669</td>
<td>8</td>
<td>5069</td>
<td>3267</td>
<td>0</td>
<td>618</td>
<td>0.58</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>MODEL</th>
<th>TR LOSS</th>
<th>TR ACC.</th>
<th>VL LOSS</th>
<th>VL ACC.</th>
<th>TE LOSS</th>
<th>TE ACC.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>1.2 ± 0.05</td>
<td>55.6 ± 0.5</td>
<td>1.1 ± 0.01</td>
<td>60.6 ± 0.9</td>
<td>1.1 ± 0.03</td>
<td>56.7 ± 0.5</td>
</tr>
<tr>
<td>CGMM</td>
<td>0.01 ± 0.01</td>
<td>99.8 ± 0.4</td>
<td>0.16 ± 0.01</td>
<td>97.9 ± 0.2</td>
<td>0.13 ± 0.01</td>
<td>96.4 ± 0.6</td>
</tr>
</tbody>
</table>

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. 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 ∈ {32, 64, 128}, 2000 epochs, batch size 128, global aggregation ∈ {sum, mean}, Adam Optimizer with learning rate ∈ {0.01, 0.001}, patience ∈ {50}. For CGMM, instead, we selected the best model across the following configurations: 20 states, layers ∈ {10, 20}, 10 EM epochs, posterior version ∈ {discrete, continuous}, embedding version ∈ {unigram, unibigram}, global aggregation ∈ {sum, mean}, batch size 64, 2000 epochs, hidden units ∈ {32, 64, 256, 512}, Adam Optimizer with learning rate ∈ {0.0001} and weight decay ∈ {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-

\[\text{https://github.com/diningphil/robust-call-graph-malware-detection} \]
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 Merca


