Adding semantic to level-up graph-based Android malware detection

Roxane Cohen, Florian Yger, Fabrice Rossi

Adding semantic to level-up graph-based Android malware detection

Roxane Cohen[1,2], Florian Yger[1], and Fabrice Rossi[2]

[1] LAMSADE, CNRS, Université Paris-Dauphine, PSL Research University
[2] CEREMADE, CNRS, Université Paris-Dauphine, PSL Research University

1 Introduction

Polymorphic malwares operate changes in their source code, while their semantics remain the same. Thus a given malware ends up having numerous different binary versions which impair detection capabilities of static analysis methods, especially signature based ones. However, the general control-flow of a malware remains rather stable under many obfuscation techniques as it relates strongly to the semantics of the program. In particular, a Function call graph (FCG) can be extracted from a program: it is an oriented graph, where vertices represent functions of the program and edges denote interprocedural calls. FCG have been shown to provide reliable descriptions of their programs, see e.g. [7]. Inspired by those results, a very large database of FCG was released, MalNet [4] providing a test bend for graph based machine learning methods.

For security reasons, MalNet graphs are reduced to FCG structure only: they do not provide any information about the represented functions, like name, class, source code, etc. We show in this paper that this information loss is possibly damaging to the practical relevance of MalNet as malware recognition is strongly reduced by this loss.

2 Labelled MalNetTiny and feature extraction

To limit the computational burden of our analysis, we focused on MalNetTiny, a small version of MalNet, that contains 5000 FCGs, evenly distributed over 5 classes: addisplay, adware, benign, downloader, trojan. We reproduced the process described in [4] to obtain the FCG but we kept some node labels (i.e. full function names with class and package information). We used in particular Androzoo [2] and Androguard [3].

We extracted two types of features from the FCG: only structural ones for the unlabelled MalNetTiny and features that combine structure and semantics for the labelled version. More precisely, the structural features are:

1. Features based on the directed call graph: as [8], we collected structural features[3];
2. Features based on the undirected version of the FCG: to assess the importance of the orientation in the call graph, we used an undirected version of the FCG and compute the same features as previously (when possible);

[3] order, size, number of strongly connected components, weakly connected components, attracting components, density, cyclomatic complexity, number of selfloops, entrypoints, isolated vertices, average degree, number of cycles, degree assortativity, degree centrality (min, max, mean), betweeness centrality (min, max, mean), average shortest path length, radius, center, diameter, average clustering coefficient, node connectivity, graph clique number, powerlaw parameters (alpha, cutoff, p-value) on in-degree and out-degree, percentage of nodes in the largest weakly connected component

3. Directed graphlets: as pointed out in [5], obfuscating graphlets in a FCG is much more complicated than e.g. simply adding fake edges, mainly because of the strong correlation between graphlet counts. We tested therefore as features the size 3 and 4 oriented graphlets density vector, using [5] algorithm;
4. Undirected graphlets: as for structural features, we considered graphlet counts for the undirected version of the FCG (using size 3 and 4 undirected graphlets).

It is well known that the rich available API in Android can be used to characterise to some extent the semantics of a program (see e.g. [1]) and in MalNetTiny, on average, FCGs contain more than 40% of external vertices (i.e. functions defined by the API and not by the program itself) and downloader FCGs reach nearly 79%. Notice that we can only extract from the apk calls from so-called *encoded nodes* (i.e. functions whose bytecode are contained in the apk) to *external nodes* (i.e. functions from the Android API). We use external calls to design features for encoded nodes as follows.

We first notice that MalNetTiny uses more than 42,000 classes from the Android API with vastly different uses from a program to another. To capture this variability at a reasonable computational cost, we represent each FCG by its top 3 functions/encoded nodes in terms of API usage diversity, with a one-hot encoding over the 42,000 classes. We use PCA to reduce the dimensionality of the representation to 32 components, explaining 75 % of the variance. Each encoded node is then represented by the projection of its external connections to those 32 coordinates. To obtain a fixed size representation of arbitrary FCG, we use two different methods:

1. Average call pattern: a FCG is represented by the average of the vector representations of all of this encoded nodes;
2. Average call and covariance: a FCG is represented by the average embedding all of this encoded nodes and the associated covariance matrix.

In addition, we complement those simple representation by graph features computed either on the full graph or on the sub graph of the encoded nodes only.

3 Results

We compared the quality of the features by using them to classify the programs into the five classes with a random forest. We split the data set into a training (80 %) and test sets. We use 10 fold cross validation on the learning set to set the hyperparameters of the classifier and report the accuracy on the test set. The process is repeated 5 times. Results are shown in Table 1.

These baselines are robust, even though the average PCA vector and associated covariance are simple and coarse features. In addition, the results are stable through repeated train-test splits. We compare to Graph Neural Networks (GNN) only on a single split -noted with (*) in Table 1- as the results were uniformly worse than these simple baselines. We used Spektral [6] to implement a dozen of different models and tested various types of GNN, such as GCN, GraphSAGE and GIN. The hyperparameters of the GNN are selected on a validation set which represents 25% of the training set. The weaker results for GNN can be explained by the particular structure of the reduced graphs after applying PCA. Since FCGs come from Android, reduced graphs contain many isolated nodes, which may prevent GNN from properly propagating information between nodes.

	Model	Averaged accuracy
Structure only	Features on non-oriented graphs	0.8692 ± 0.0068
	Features on oriented graphs	0.87 ± 0.0078
	Graphlet densities	0.7272 ± 0.0074
	Oriented graphlets densities	0.8322 ± 0.012
	GNN + degree	0.6743 (*)
	GNN + in-degree, out-degree	0.7525 (*)
Extra semantic	PCA features (external call)	0.9076 ± 0.0068
	PCA features + covariance	$\mathbf{0.9214} \pm 0.0067$
	PCA features + covariance + reduced-graph features	0.9183 ± 0.0035
	PCA features + covariance + graph features	0.9208 ± 0.0049
	GNN + reduced-ACP features	0.8958 (*)

Table 1. Comparison of the test accuracy achieved by each model

4 Conclusion and future work

In this work, we demonstrate that while structural features on MalNetTiny can be used to classify malwares, significant improvements of the classification rate can be obtained by representing each graph by a low dimensional feature vector extracted from external calls to the Android API. Such a representation can be efficiently processed by simple shallow models. We also study the impact of the orientation of the FCG and recommend the use of this information if structural features only are to be used. Finally, we show that simpler shallow models can have worthwhile performances and we advocate for their systematic use as baselines in graph classification tasks.

Finally, there are several possible ways to improve this work (e.g. validating the number of vertices used to construct the API-based features or the number of components kept in the PCA). Since we only consider external class API calls, it is promising to increase the feature granularity by taking into account external methods calls.

References

1. Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: Security and Privacy in Communication Networks (2013)
2. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories. MSR '16 (2016)
3. Desnos, A., Gueguen, G.: Android: From reversing to decompilation. Proc. of Black Hat Abu Dhabi (2011)
4. Freitas, S., Dong, Y., Neil, J., Chau, D.H.: A large-scale database for graph representation learning. NeurIPS (2021)
5. Gao, T., Peng, W., Sisodia, D., Saha, T.K., Li, F., Al Hasan, M.: Android malware detection via graphlet sampling. IEEE Transactions on Mobile Computing (2019)
6. Grattarola, D., Alippi, C.: Graph neural networks in tensorflow and keras with spektral (2020)
7. Hu, X., Chiueh, T.c., Shin, K.G.: Large-scale malware indexing using function-call graphs (2009)
8. Ikram, M., Beaume, P., Kaafar, M.A.: Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling (2019)