Deep Graph Neural Networks for Malware Detection Using Ghidra P-Code

Rinaldo Iorizzo and Bo Yuan
Rochester Institute of Technology, USA

rpi1809@g.rit.edu
Bo.Yuan@rit.edu

Abstract: This work examines the effectiveness of using Ghidra P-Code as semantics-based features in a graph neural network-based malware detection system. A preliminary model exhibits a function level precision of ~70% and a recall around ~60%, and a precision and recall of ~55% and ~80% respectively for the program level detection task on a dataset of ~50,000 control flow graphs extracted from functions of malicious and benign programs. Future improvements to this ongoing project include, but are not limited to, collecting dynamic control flow graph information as opposed to static graphs to provide the model with resilience to advanced malware obfuscation and encryption schemes.

Keywords: Malware Detection, Deep Learning, Neural Network, Graph Neural Network, Ghidra

1. Introduction

As malware authors employ ever more sophisticated evasion techniques, it has become apparent that classical approaches such as signature detection leave much to be desired. Malware detection remains difficult in the presence of obfuscation techniques developed by malware authors.

In many cases, deep learning approaches that make use of static features are assumed to fail in the presence of obfuscation techniques. Detection techniques such as those shown in MalIMG (Nataraj et al. 2011) rely upon the bytes of compiled programs, which are easily changed through obfuscation. This has led many researchers to use dynamic features in their malware detection systems, as dynamic features are assumed to be more closely aligned with the behaviour of the program as opposed to the form of the program. By this logic, we extrapolate that the intention of using dynamic features is the alignment of the representation of the samples with the potential behaviour of the sample program. Then, we direct our attention to producing such a representation through the exploration of program representations that avoid over-reliance on the exact form of a given program.

Inspired by vulnerability detection techniques (Cao et al. 2021), we consider the classification of Control Flow Graphs (CFGs) derived from the functions in compiled programs. In this work we explore the relationship and feasibility of the binary classification of function CFGs from malicious and benign software to the task of binary classification on whole programs.

We propose a deep learning malware detection system based on the classification of function control flow graphs with Ghidra P-Code node-level features. Through analysis of our model's performance upon a sample dataset of malicious and benign software under a variety of circumstances we aim to explore the relationship between function classification and malware detection as well as the use of P-Code as a semantic feature for augmenting a GNN malware detection system for resistance to obfuscation techniques.

We have identified several pertinent research questions:

1. How effective is detection at the function level with respect to malware detection at the program level?
2. How does the classification of P-Code derived CFGs for malware detection compare to more traditional malware detection techniques? (Both graph based and otherwise.)
3. Is P-Code suitable for malware detection in the presence of obfuscation?

2. Background and Related Works

2.1 Ghidra and P-Code

Ghidra was selected for decompilation and analysis of programs in this work to make use of Ghidra's internal language for program representation: P-Code. Ghidra internally represents analysed programs using P-Code. P-Code represents computation using registers in a generic way by breaking known instructions on known CPU architectures into one or more P-Code instructions (or P-Code nodes). According to Naus et al. (2023), P-Code is

not semantically complete nor executable, however we consider that the application of P-Code to reverse engineering is indicative of the potential for useful representations for deep learning systems.

2.2 Graph Neural Networks

Graph neural networks (GNNs) are a kind of deep learning architecture that takes graphs as input. GNNs have been used across many domains ranging from molecular analysis (Gilmer et al. 2017) to abstract mathematics (Xu et al. 2019) and facilitate tasks on graph data on the node level, edge level, and or whole graph level. GNN models take graph data in the form of matrices of node features, edge features, and an adjacency matrix. In our case, we make use of GNN models for a graph-level classification task where the features of nodes in the input graphs are used to classify the entire graph into one of two classes, malicious or benign.

2.3 Related Works

The following is a discussion of various related works in malware detection. Many of these works informed our decisions made during the creation of our malware detection system. While this work focuses on Windows malware detection, works focused on Android malware detection were also considered.

2.3.1 Static Approaches

Many malware detection systems make use of static features. Examples such as Nataraj's work on the Malimg dataset often make use of machine learning or DL tools from other domains such as image classification (Nataraj et al. 2011). Palma Salas et al. (2023) made use of transfer learning to improve detection on a Malimg based program representation. However, Palma Salas made use of the Microsoft Malware Classification Challenge Dataset for their experiments which does not contain benign software and includes obfuscated samples as a separate class. While their approach was successful, it inherits the vulnerability to obfuscation techniques from their inspiring work, Malimg.

In Pei et al. (2020), the authors used graph representations of android programs to train a malware detection system. While their approach enjoyed high performance, it was not directly applicable to Windows malware detection as Pei's work relies upon Android specific features and obfuscation techniques. Pei's work supports the use of GNNs with semantic attribution at the node level for graph level classification tasks. While Pei's work is demonstrated on obfuscated data, the performance of their approach when only a small portion of the dataset exhibits obfuscation techniques was drastically reduced.

Yan et al. (2019) demonstrated a powerful graph convolution network approach on statically derived CFGs for malware classification. Yan's work supports the creation of CFGs from compiled programs and the attribution of information to nodes in the CFG to retain semantic information. Yan's work strongly supports the use of CFGs for malware detection using static information and provides a basis for future adaptation. However, Yan's work also makes use of the Microsoft Malware Classification Challenge Dataset for one set of their experiments and otherwise does not address obfuscation.

2.3.2 Dynamic Approaches

Dynamic approaches are popular for their ability to capture the behaviour of samples, avoiding obfuscation techniques in the best case. It is common for these approaches to make use of system API calls as in Li et al. (2022), Xiao et al. (2019), and Nguyen et al. (2023). All three of these works make use of a sandbox for the collection of dynamic information, as well as the use of graph structures for representation of their samples. Li's work provides competitive performance with graph convolutional networks, but not to an overwhelming degree. Nguyen's work similarly performs well but not more-so than approaches such as the application of CNNs to Malimg samples. Xiao's work demonstrates the use of stacked auto encoders (SAEs) for eventual classification of graph structured information. However, Xiao's work is intended for use in IoT environments and performs graph compression techniques which may remove structural information. Of these three works, only Nguyen discusses obfuscation at length.

3. Methodology

To address our first research question "How effective is detection at the function level with respect to malware detection at the program level? ", we will compare our GNN model's performance on several datasets and training configurations that differ in their relationship between functions and programs overall.

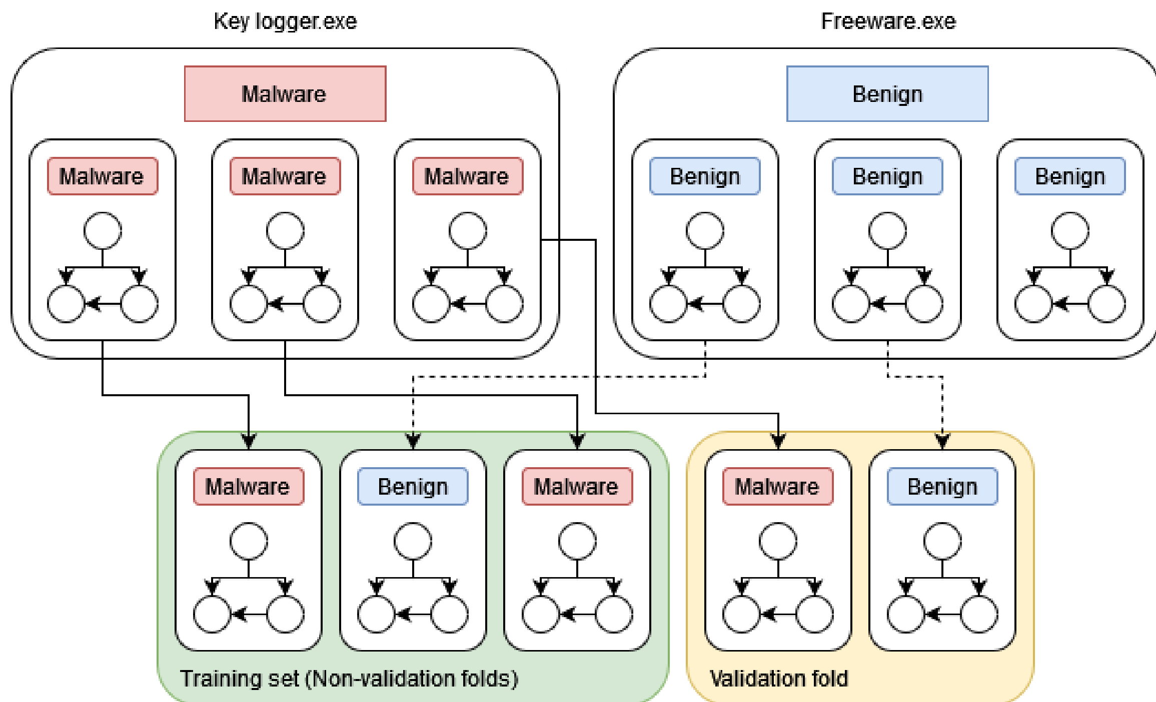

Figure 1: Our first dataset configuration is constructed by selecting function CFGs by sampling the total data generated with Ghidra. Note that not all functions from a program are necessarily selected for inclusion in the dataset.

We first construct several datasets that are balanced with respect to the number of function CFGs from each class (Figure 1). The objective of our model is to predict the class of the program the function CFG originates from. To collect information on program level malware detection, we look at the programs represented in the validation set. If any of a program's functions are considered malicious, we consider the whole program to be malicious.

Next, we create new datasets that are balanced by number of programs from each class and a maximum number of function CFGs (Figure 2). The datasets select programs by their function count and select all functions from each program. We then apply the same training objective and train our model. We account for duplicate programs in this dataset by comparing MD5 hash values. We additionally control for variants of the same malware by considering the number of functions of each program. Programs that have the same number of functions and similar file sizes are considered duplicates. The relationship of program function count to file size can be seen graphically in figure 3. Note the vertical lines and dark points that malware samples form, indicating programs with the same number of extracted functions. Manual analysis of these artifacts indicated that they strongly correlate with variations of the same program through consultation of Virustotal. All malware programs that are not present on Virustotal are excluded from sampling.

Lastly, we consider a similar representation as in the previous datasets where we include all functions from represented programs but do not allow programs to be split between training and validation folds (Figure 4). In doing so, we enforce that the program level detection task is not skewed by partial representation in the validation set. Samples are selected in the same manner as in the previous paragraph.

In this work, we do not provide experimental results to explore our third research question but instead lay the groundwork for doing so. To properly address our third research question, "Is P-Code suitable for malware detection in the presence of obfuscation?" we consider in our future work the extraction of P-Code through dynamic means and the construction of a dataset with known obfuscation techniques. To do this however, it was first necessary to create a proof of concept for P-Code's use in malware detection.

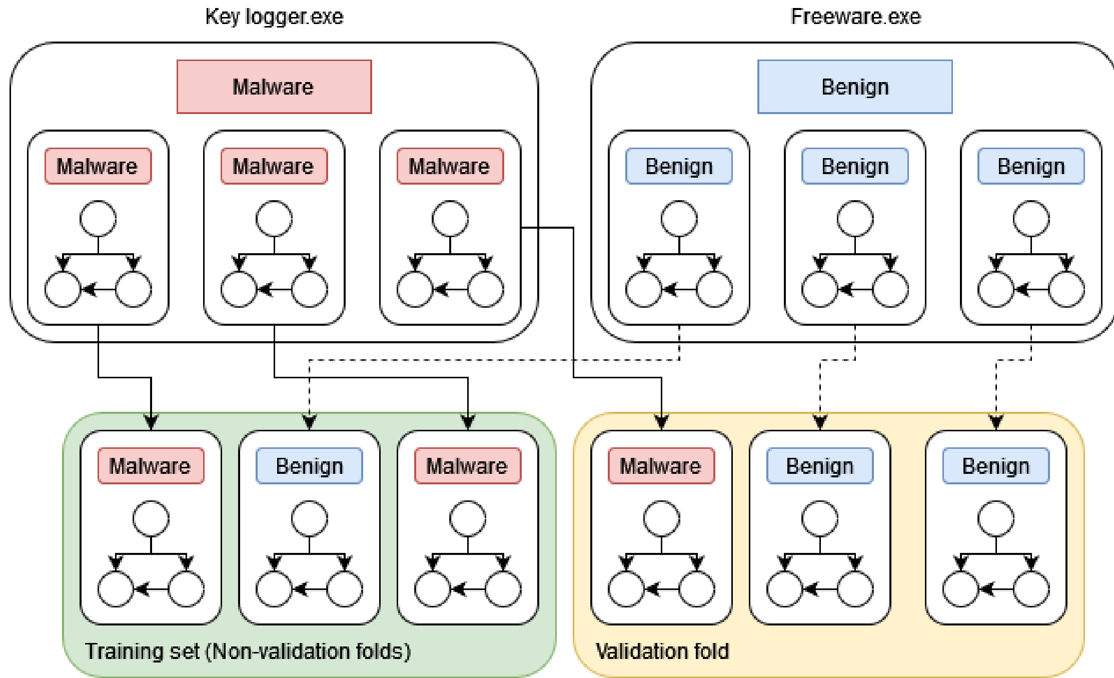

Figure 2: The second configuration enforces that all functions from a program must be included in the total dataset as well as attempts to balance the number of programs and functions from each class.

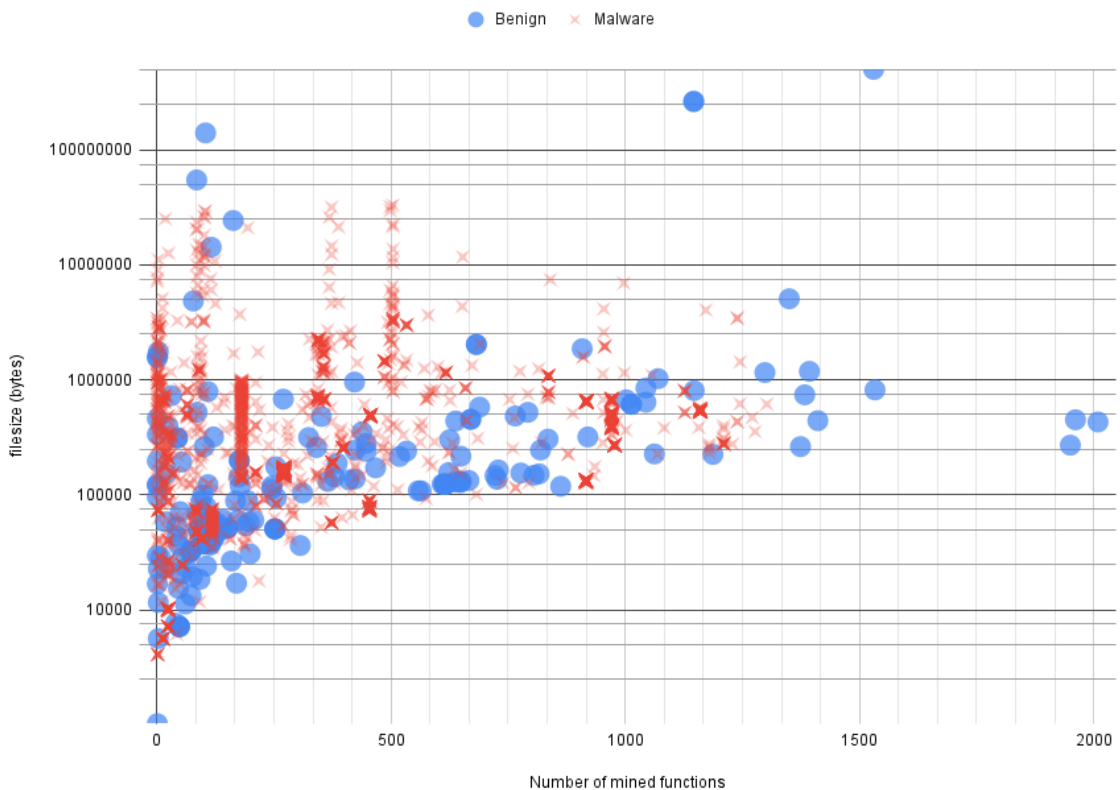

Figure 3: This chart compares the number of mined functions to the file size of the sample program in bytes. The vertical axis is a logarithmic scale due to high disparity in file sizes.

4. Experimental Setup

Spektral was used in tandem with Keras to construct and apply GNN models using Tensorflow as the backend. The system was tested on an Ubuntu Server with a 32 core Xeon E5 2620 CPU and Tesla P100 GPU with 16GB of memory. Our malware detection system was trained on a collection of malware samples from a variety of sources such as Virustotal and Virusshare, and a collection of benign samples from common open-source and freeware projects.

Samples were processed using Ghidra to produce P-Code CFGs for every function of the sample program and were exported as JSON files. The CFG data was processed using node2vec (Grover and Leskovec, 2016) to provide artificial node-level features, and additional features of node degree and P-Code node-kind were concatenated to each node feature vector to complete the initial node representations. Training datasets were created by sampling the total set of programs processed by Ghidra.

A Spektral GeneralGNN model is trained and tested across 10-fold cross validation. Precision and recall are collected for each fold and averaged to obtain performance metrics. This process is performed for each dataset.

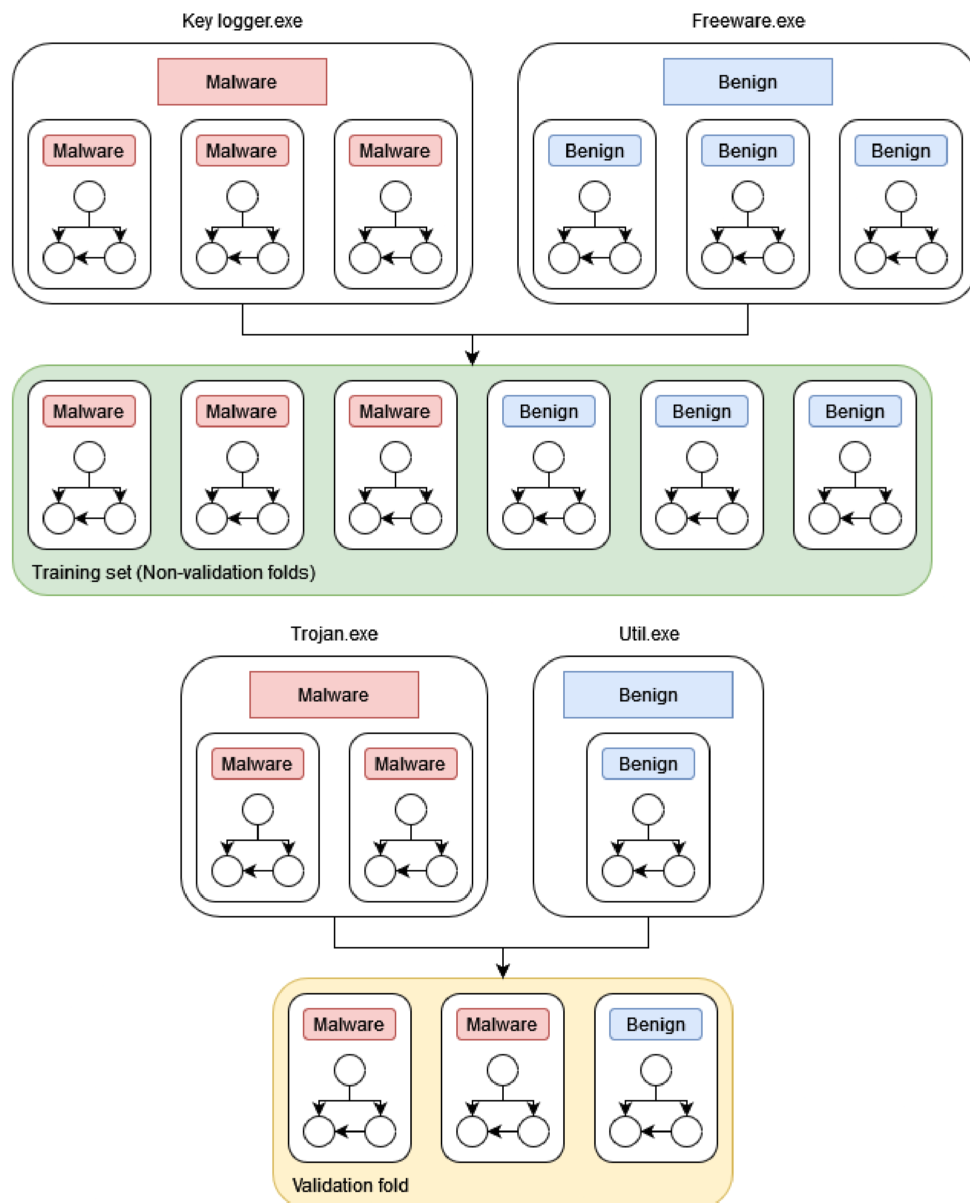

Figure 4: In addition to balancing the number of programs and number of functions from each class in the dataset, we perform train and validation fold splitting along program lines instead of by function count. The fold selection process only considers the number and class of program.

5. Results

The GeneralGNN model was built into the Spektral GNN library and was selected as a general baseline for GNN models. This model was an implementation of the work in You et al. (You, Ying, and Leskovec 2021). Results are shown in table 1. Dataset size indicates maximum number of function CFGs in the dataset. The column "Split on Program" indicates the splitting of training and validation sets on the program level. "Has all Functions?" indicates if all functions from a program are present in the dataset. "Class Weighted" indicates the use of the class weight parameter used by the focal cross entropy loss function. "Pruned" indicates that duplicates are removed. The rows in red reinforce that no pruning has occurred and are derived from the first dataset description in the methodology section.

Table 1: GeneralGNN model results collected across the datasets described in the methodology section.

Dataset Size	Split on Program?	Has all Functions?	Class Weighted	Pruned	Function Precision	Function Recall	Program Precision	Program Recall
10k	no	yes	no	yes	0.79	0.61	0.56	0.70
25k	no	yes	no	yes	0.71	0.58	0.53	0.79
50k	no	yes	no	yes	0.71	0.64	0.55	0.86
10k	yes	yes	no	yes	0.77	0.62	0.66	0.93
25k	yes	yes	no	yes	0.71	0.41	0.61	0.90
50k	yes	yes	no	yes	0.77	0.44	0.61	0.92
10k	no	yes	yes	yes	0.75	0.58	0.55	0.68
25k	no	yes	yes	yes	0.69	0.46	0.52	0.74
50k	no	yes	yes	yes	0.80	0.46	0.54	0.77
10k	yes	yes	yes	yes	0.77	0.53	0.66	0.87
25k	yes	yes	yes	yes	0.70	0.44	0.62	0.94
50k	yes	yes	yes	yes	0.76	0.46	0.60	0.91
10k	no	no	no	no	0.58	0.66	0.71	0.61
25k	no	no	no	no	0.59	0.74	0.81	0.73
50k	no	no	no	no	0.64	0.72	0.85	0.73

6. Discussion

When considering the relationship between function-level prediction and program-level prediction, our results indicate that including entire programs and not splitting them between training and validation sets is beneficial. The low program precision indicates that false positives are likely, but a high recall indicates that our model is unlikely to miss malicious software. This aligns with the intuition that our representation may indicate program behaviour to some degree and that the broad labels applied to the function level task are insufficient to differentiate non-malicious behaviour within malware from non-malicious behaviour in benign software. When considering the red rows present in table 1 that do not include pruning or exclusivity to training/ validation sets, it is important to consider the number of programs represented in the validation set increases with the size of the dataset overall. Therefore, the performance increase shown in program metrics for the red rows falsely indicate high performance in malware detection at the program level.

To address our second research question, "How does the classification of P-Code derived CFGs for malware detection compare to more traditional malware detection techniques? (Both graph based and otherwise.).":

Our model performs better than guessing and achieves a high recall on the program level task when trained on datasets that are controlled for duplicates and include all relevant functions. Models like those in the related works section often boast over 90% precision and recall; our model is far below state of the art at present.

However, if we assume our semantic representation closely aligns with the behaviour of the original program it is possible our results are consistent with the common intuition that malicious software's functions and behaviours may only constitute a small proportion of malware's codebase. If this is true, investigation into characterization of program behaviour from such features may allow for higher performance with labels that are derived from program behaviours. It is an ongoing research. The performance of our model is expected to improve.

Acknowledgements

This work was partially supported by the National Science Foundation under Award 1922169.

References

Cao, S. *et al.* (2021) 'BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection', *Information and Software Technology*, 136, p. 106576. Available at: https://doi.org/10.1016/j.infsof.2021.106576.

Gilmer, J. *et al.* (2017) 'Neural Message Passing for Quantum Chemistry'. arXiv. Available at: http://arxiv.org/abs/1704.01212 (Accessed: 13 December 2023).

Grover, A. and Leskovec, J. (2016) 'node2vec: Scalable Feature Learning for Networks'. arXiv. Available at: http://arxiv.org/abs/1607.00653 (Accessed: 2 August 2023).

Li, S. *et al.* (2022) 'Intelligent malware detection based on graph convolutional network', *The Journal of Supercomputing*, 78(3), pp. 4182–4198. Available at: https://doi.org/10.1007/s11227-021-04020-y.

Nataraj, L. *et al.* (2011) 'Malware images: visualization and automatic classification', in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*. *VizSec '11: 2011 International Symposium on Visualization for Cyber Security*, Pittsburgh Pennsylvania USA: ACM, pp. 1–7. Available at: https://doi.org/10.1145/2016904.2016908.

Naus, N. *et al.* (2023) 'A Formal Semantics for P-Code', in A. Lal and S. Tonetta (eds) *Verified Software. Theories, Tools and Experiments.* Cham: Springer International Publishing (Lecture Notes in Computer Science), pp. 111–128. Available at: https://doi.org/10.1007/978-3-031-25803-9_7.

Nguyen, M.T., Nguyen, V.H. and Shone, N. (2023) 'Using deep graph learning to improve dynamic analysis-based malware detection in PE files', *Journal of Computer Virology and Hacking Techniques* [Preprint]. Available at: https://doi.org/10.1007/s11416-023-00505-x.

Pei, X., Yu, L. and Tian, S. (2020) 'AMalNet: A deep learning framework based on graph convolutional networks for malware detection', *Computers & Security*, 93, p. 101792. Available at: https://doi.org/10.1016/j.cose.2020.101792.

Xiao, F. *et al.* (2019) 'Malware Detection Based on Deep Learning of Behavior Graphs', *Mathematical Problems in Engineering*, 2019, p. e8195395. Available at: https://doi.org/10.1155/2019/8195395.

Xu, K., Hu, W., Leskovec, J. and Jegelka, S., 2018. How powerful are graph neural networks?. *arXiv preprint arXiv:1810.00826*.

Yan, J., Yan, G. and Jin, D. (2019) 'Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network', in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 52–63. Available at: https://doi.org/10.1109/DSN.2019.00020.

You, J., Ying, R. and Leskovec, J. (2021) 'Design Space for Graph Neural Networks'. arXiv. Available at: http://arxiv.org/abs/2011.08843 (Accessed: 13 December 2023).