# Malware Detection Using Unsupervised Clustering of Binary File Control Flow Graphs

K. Liyanage, R. Pearsall, C. Izurieta, B. M. Whitaker

**Abstract**—Detecting malware from binary files is an important task in the research and development of the fields of cybersecurity and machine learning. This paper discusses the viability of unsupervised machine learning clustering techniques to identify differences between graph representations of benign software and malware. This paper evaluates the utilization of graph analysis and unsupervised clustering in separating malware and benign binary files. The binary files are first converted into a control flow graph (CFG) representation. This is carried out by the CFGEmulated() function of the *angr* Python library. Then the graphs are converted to a vector representation using the Graph2Vec algorithm. Finally, several unsupervised clustering techniques were used on the dataset. Preliminary results indicate the viability of using CFGs and unsupervised clustering as a malware detection method.

**Index Terms**—Control Flow Graphs, Clustering, Malware.

## 1 Introduction

In recent years the number of attacks by malware shared through the internet has increased. To avoid damage to computer systems, effective identification and classification of malicious programs from benign programs are necessary. In this research, we focus on analyzing binary files using control flow graphs (CFGs) and exploring the viability of using unsupervised clustering techniques for detecting malware. While it has not been established that there is a significant difference between CFGs belonging to malware and benign programs, we hypothesize that such a difference exists. The research focus is to explore the viability of Machine Learning (ML) as an effective tool in identifying malware through graph analysis of CFGs. This work is an extension of preliminary work done by Veronika et al for the Department of Homeland Security [1].

### 1.1 Goals and Contributions

- **Benign and Malware dataset.** As the initial step, we curate a dataset of benign and malicious programs. The dataset consists of benign operating system files as well as malware provided by *Hoplite industries*[1]. Importantly, this curated dataset of CFGs obtained from malware and benign binary files is publicly shared. Publicly available combined (Benign + Malware) datasets are rare, hence the dataset published with this manuscript will help future researchers to test and evaluate new algorithms.
- **A robust framework for unsupervised clustering.** A robust end-to-end workflow is proposed to minimize model over-fitting and model contamination. The framework is flexible for swapping and trying different methods for research purposes. As with the dataset, the software described in this manuscript is also publicly available.
- **Viability study.** A discussion regarding the acquired results is provided, showing the viability of clustering to separate some benign and malicious files. In addition, we present challenges and possibilities for future expansions.

The overview of the analysis is shown in Figure 1. First, the binary files are converted into CFGs. This is carried out by the *CFGEmulated()* function of the *angr* python library. The graphs are then converted to a fixed-length vector representation using the *Graph2Vec* algorithm [2]. It can be seen that usually a dictionary of rooted sub-graphs is learned and then the similarity is calculated between the sub-graphs and the CFGs of the binaries. A dictionary of rooted sub-graphs is learned and then the similarity is calculated between the sub-graphs and the CFG representations of the binaries. Using the Graph2Vec graph embedding method the graphs are first converted into a document using the Weisfeiler-Lehman graph kernel where each "word" is a rooted sub-graph. A whole graph is represented as a document with a collection of "words". Then convert the documents into a fixed-length vector representation of the graph embedding via Doc2Vec algorithm. The implementations of the *Graph2Vec* is shown in Figure 2, where the graphs are first converted into a word document using the Weisfeiler-Lehman graph kernel [3]. The algorithm then uses *Doc2Vec* [4] to create a fixed-length vector describing the whole graph. Finally, several unsupervised clustering techniques are used on the dataset. This allows for determining the viability of using CFGs as a static malware detection method with unsupervised clustering. The source code associated with this manuscript is hosted on GitHub[2] and the CFG dataset is archived on Zenodo[3].

- *K. Liyanage and B. M. Whitaker are with the Department of Electrical and Computer Engineering, Montana State University, Bozeman, MT, 59719. E-mail: bradley.whitaker1@montana.edu*
- *R.Pearsall and C. Izurieta are with the Gianforte School of Computing, Montana State University, Bozeman, MT, 59719.*

*Funded by the Department of Homeland Security (DHS) Science and Technology (S&T) Directorate under contract number 70RSAT22CB0000005. Any opinions contained herein are those of the authors and do not necessarily reflect those of DHS S&T: Manuscript submitted Feb 15, 2023.*

1. https://www.hopliteindustries.com/

2. https://github.com/MSUSEL/unsupervised-graph
3. https://doi.org/10.5281/zenodo.7630371

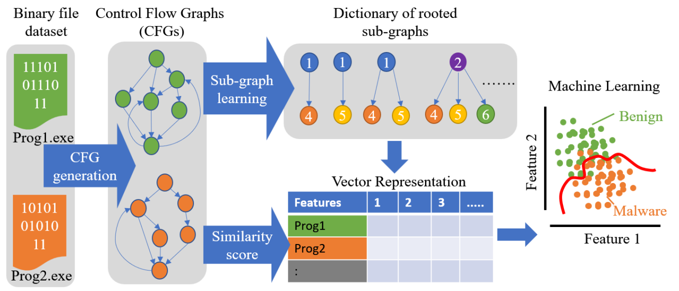


Fig. 1. Overview of binary file analysis using CFGs

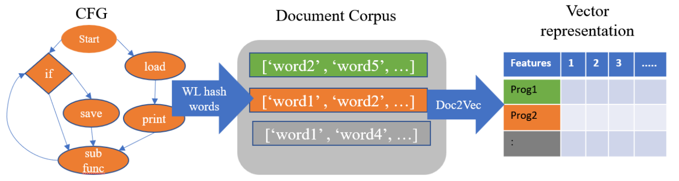


Fig. 2. Graph2Vec implementation overview with Weisfeiler-Lehman hash words and Doc2Vec algorithms.

## 2 Background

### 2.1 Preliminary work summary

Preliminary work in this project was conducted by Veronika et al. [1]. The authors proposed a basic framework with a synthetic program dataset using prospective tools and methods. They curated two datasets, namely the synthetic graph dataset and the distinct benign code (DBC) dataset. The datasets were used to explore the effectiveness of graph embedding techniques and clustering performance.

The synthetic graph dataset is a set of various graph structures to evaluate the effectiveness of the graph embedding methods. The graphs were created in the *networkX*[4] [5] Python package. The dataset consists of 100 graphs per type of *path*, *cycle*, *complete*, *Erodos-Renyi*, *Barabasi-Albert* and, *Neuman-Watts-Strogatz* graphs. The graphs were converted to a vector representation through *Graph2Vec* implementation from the *karateclub*[5] [6] python package. Observation showed a separation between different graph categories when the vector dimensions are 8 and above.

The DBC dataset comprises 49 variations for each of five basic C++ programs *fibonacciSequence*, *isPrime*, *productPrices*, *randomList*, and *repeatingString*. The control flow graphs (CFG) were constructed by using the binary disassembly tool *angr*[6] [7] python package. The tool was chosen considering the compatibility with the existing Idaho National Labs' @DISCO[7] tool. CFGs were created using *CFGFast()*, a static analysis function in *angr*. Three graph embedding methods were considered (*Local Degree Profile (LDP)* [8], *Graph2Vec* [2], and *GL2Vec* [9]) to create vector representations. Then the vectors were clustered using three unsupervised methods: *agglomerative*, *K-means* and, *DBSCAN*. The clustering is evaluated using the *Normalized Mutual Information (NMI)* metric.

4. https://networkx.org/
5. https://karateclub.readthedocs.io/
6. https://docs.angr.io/
7. https://github.com/idaholab/atDisco

The *scikit-learn*[8] [10] python package was used for clustering algorithms and metrics. *Graph2Vec* performed better with *K-means* and *Agglomerative* clustering. Meanwhile, *GL2Vec* performed uniformly for all three methods and outperformed *Graph2Vec* for *DBSCAN* only. *LDP* did not show any significant performance for any clustering method.

The report suggested experimenting with different graph embedding techniques, graph representations, and disassembly tools. Also, it suggested using large-scale of data covering various sources. Further, it suggested trying alternate clustering techniques and anomaly detection methods.

### 2.2 Malware

Malicious ware, or malware, is a general term used to describe an unwanted, unauthorized computer program or script with the intent to cause some kind of damage or harm. System administrators are unaware of the presence and behavior of malware, and if they were to be aware, they would not permit such program to run on the system [11]. Malware typically attempts to gain unauthorized access to some system with the goal of stealing sensitive or financial information, disrupting services, or gaining remote access for later use.

As thousands of new malware strains begin to surface each day, the ability to detect malware before it can cause harm has been an area of focus for many cybersecurity experts. Bazrafshanet et al. define three distinct malware detection methodologies: Signature-based, Behavioral-based, and Heuristic-based [12]. Signature-based detection is the most common method for detecting malware. This technique involves searching for a known digital footprint that has already been detected and recorded in the past. File hashes and byte strings that represent things such as function names, IP addresses, or coding structures are both commonly scanned for in static detection. Unfortunately, static-based detection methods fail to detect new forms of malware or malware that obfuscates itself.

Behavior-based detection uses dynamic analysis for evaluating malware, which is the process of analyzing code or a script by executing it and observing its actions. Dynamic analysis is typically done in a sandbox or virtual environment so malware is not able to cause any damage. Behavioral-based detection includes monitoring Windows registry activity, binary instructions that are executed, and the presence of data in RAM during execution.

Lastly, heuristic-based detection leverages machine learning or data mining in order to make decisions. Features from malware are extracted and then used to train a machine learning classifier. Features that are extracted include operating system API calls, opcode frequency, and program control flow. Such attributes of a program help capture the behavior of a program and provide a set of traits for a machine learning algorithm to learn from.

### 2.3 Control Flow Graphs

Control Flow Graphs (CFGs) are one of the most common graph representations of code used for binary analysis. Several different graph representations were explored for this

8. https://scikit-learn.org/

research, but Control Flow Graphs yielded the best results. The primary focus of a CFG is to illustrate the control flow, or order of executed statements, of a program. A node in this directed graph typically represents a basic block of code, which is a sequence of instructions that are executed sequentially with no jumps or branches to other sections of code. The edges represent transitions from a basic block to a basic block through function calls, return statements, program branches, or looping. A binary can be converted into a CFG first by disassembling it into some intermediate low-level representation, such as an assembly language. The assembly language can then be analyzed and searched for program jumps and branch targets to determine the control flow. Several tools already exist for generating a control flow graph from a binary, such as Ghidra, Binary Analysis Platform, and angr. Several works have been published that utilize machine learning with CFGs [13], [14], [15].

## 2.4 *Graph2Vec* Graph Embedding

Once the CFGs are generated, the graphs have to be represented as a fixed-length vector for machine learning algorithms. Several graph embedding methods can be used for vector representation purposes. In this work, we use the *Graph2Vec* method. As shown in Figure 2, the typical implementation of this algorithm can be divided into several stages.

### *2.4.1 Weisfeiler-Lehman (WL) graph hash*

The WL hash algorithm is a simple algorithm used to identify common sub-tree patterns in a set of graphs. The algorithm initially relabels the graph nodes according to the number of its neighbors. It then iteratively performs a breadth-first search for neighboring nodes. At each iteration, the base node is relabeled by adding information about its neighbors' labels. In each iteration, the appended relabel is converted to a fixed-length unique hash value. These techniques produce deterministic and unique relabeling for each node depending on the number of iterations.

It is possible to use different graph kernels for identifying sub-tree patterns. A recent extensive survey has listed different graph kernels and their comparison [16]. In the survey, the authors provide a helpful "practitioner's guide", a recommendation pipeline for selecting a graph kernel according to a specific application. (See Fig. 10 in [16]. For malware applications, the WL sub-tree kernel is justified through this pipeline. Since program CFGs do not contain edge information and are typically large graphs, WL kernels can be considered. For our study, we only consider the local structures of the graphs. Hence, the WL sub-tree kernel is justified. The motivation for focusing on the local structure is to observe if any coding patterns would emerge as significant features for a different kind of Malware.

### *2.4.2* Doc2Vec

*Doc2Vec* is a technique used in the domain of natural language processing (NLP). Since, the WL hash algorithm uniquely relabels nodes with information about its neighbors, the list of new node labels can be considered as a representation of the graph. If the labels are considered as words, each graph can be considered a document with a set of words. First, a vocabulary is constructed by using the most frequent words. Then *Word2Vec* is used to measure the similarity. *Word2Vec* uses a *skip-gram* approach using a shallow neural network to predict the closeness of the words. Finally, a low-dimensional vector is created for each document (graph).

## 2.5 Unsupervised Clustering

Unsupervised clustering is a machine learning technique where an algorithm tries to group the data points in the detests using some heuristic criteria without using any knowledge about prior existing group information. This is typically a challenging problem since the algorithm does not know how many data clusters (groups) exist. Some common heuristics used are similarities (density) and differences (distances). In this work, we are using several unsupervised clustering algorithms provided in the *scikit-learn* toolbox: *k*-means clustering, spectral clustering, DBSCAN, and agglomerative clustering.

In *k*-means clustering, the algorithm attempts to divide the data into *k* number of groups by minimizing the variance within the cluster. The number of *k* clusters has to be predetermined by the user. In spectral clustering, data is first embedded into a low dimensional manifold then clustering is performed by using *k*-means clustering. Again, the number of clusters has to be pre-determined by the user. In DBSCAN data is clustered by separating high-density areas into clusters. Therefore the final number of clusters is unpredictable and low data points in low-density areas are categorized as noise. The hyper-parameters include the density factor and the minimum number of data points per cluster. In agglomerative clustering, initial clusters are divided and merged as a tree structure using nested clustering. In *scikit-learn* a bottom-up approach is used to combine each data point into clusters using a distance measurement. The distance measurement and the number of clusters have to be pre-determined.

# 3 Experiment

The overall experimental workflow is shown in Figure 3. The workflow consists of two pipelines. The left pipeline is for training purposes and the right pipeline is for testing programs. Clear separation is maintained between the two pipelines to reduce the over-fitting of the models. Only the trained models are used for testing purposes.

## 3.1 Dataset

The dataset associated with this work consists of 3000 malicious files and 3000 benign files. Malware samples were provided for this research by local cybersecurity company Hoplite Industries [9]. This collection of malware is an aggregation of samples collected from VirusTotal.com[10], honeypots, and personal cybersecurity connections. The collection of malware includes several different types of malware, such as trojans, viruses, and droppers. The malware samples consisted of many different file types, but only PE

9. https://www.hopliteindustries.com/
10. https://www.virustotal.com

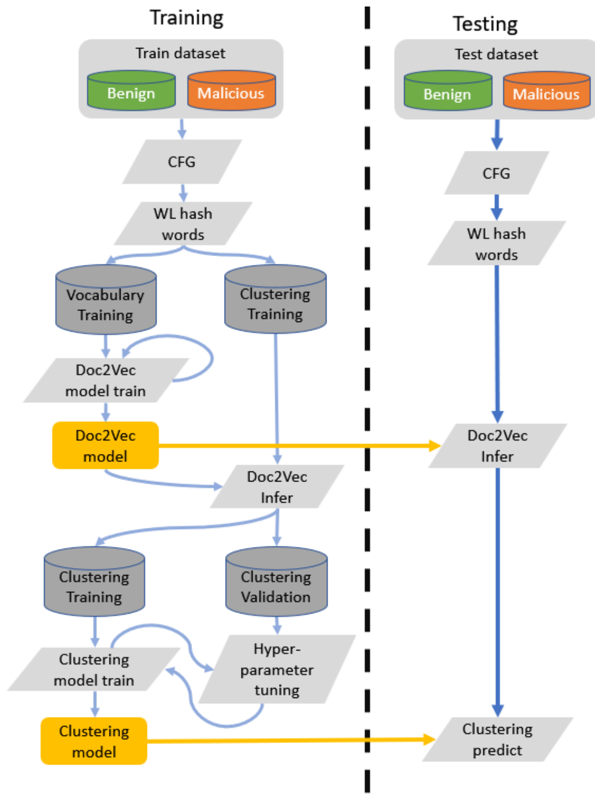


Fig. 3. Overall workflow of the data. The dataset is divided into training and testing sets. The training set is used to train the $Doc2Vec$ model and the clustering models. The trained models are applied to the testing set to evaluate the performance.

executable and ELF binaries were evaluated in this research. For benign programs, operating system files were collected from versions of Windows and Linux. This includes trusted DLL files and benign Linux executables. The summary of categories of malware in the dataset is given in the Github repository.

### 3.2 Control Flow Generation

CFG generation was done using the python library *angr*, which provides two different methods for generating control flow graphs: *CFGFast*, and *CFGEmulated* [11]. *CFGFast* is a static analysis method where the program is evaluated in random positions. *CFGEmulated* is a dynamic analysis method where the program is emulated using symbolic execution to identify the flow paths. Although more accurate than *CFGFast*, this method takes more time. Further the accuracy is bounded by emulation restraints like missing hardware modules, system calls, and the choice of hyperparameters. For this project we have chosen *CFGEmulated* as the CFG generation method. This is because we have prioritized the accuracy of the CFGs that are generated rather than the time it takes to generate them. One of the parameters for the *CFGemulated* is the context sensitivity. In this project we chose context sensitivity level 3[12].

11. https://docs.angr.io/built-in-analyses/cfg
12. https://docs.angr.io/built-in-analyses/cfg#context-sensitivity-level

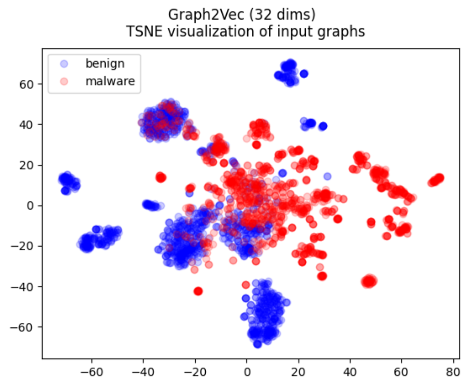


Fig. 4. Two dimensional representation of 32-dimensional *Doc2Vec* representation

### 3.3 Graph embedding

The generated graphs have to be represented with a fixed-length vector. First, WL hash relabelling is conducted for each graph independently. This is computationally expensive since, for each graph, an exhaustive search is conducted. However, this step can be conducted independently for each graph. Two iterations of the WL hash are chosen as a compromise between computational cost and sub-tree structure.

A portion of the training set is then used to build the vocabulary and train the *Doc2Vec* model. The model dimensions considered are $(2, 4, 8, 16, 32, 64, 128, 256)$. Next, the trained *Doc2Vec* model is used to infer the training set for the unsupervised clustering. As an example, Figure 4 shows the 2 dimensional *t*SNE [17] representation, where *t*sne is a method for visualizing high dimensional data in a low dimension with statistical relationships. While some clear groupings of CFGs can be observed in the figure, there is also a significant overlap present. Further investigation is required to identify which features or CFG sub-tree patterns correspond to each grouping.

### 3.4 Unsupervised Clustering

To identify the clusters in the generated vectors, four unsupervised clustering methods were employed. For each method, a grid search is conducted with hold-out validation to determine the hyperparameters.

Several clustering evaluation metrics were measured for each clustering algorithm to determine the “best” hyperparameters. Since this is not an exhaustive search “best” is among the considered options. The considered cluster evaluation metrics are as follows: RAND index, Adjusted RAND index (ARAND), Mutual information score (MIS), Adjusted mutual information score (AMIS), Normalized mutual information score (NMIS), Homogeneity (Hmg), Completeness (Cmplt), and V-measure (V_meas).[13]

Finally the clustering models trained using the best hyperparameters are utilized to predict the clustering for the training test. The best parameter predicted by each metric is shown in Table 1.

13. https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation

TABLE 1
Best hyper-parameters predicted through different metrics

| Metric | K-means | | Spectral | | Agglomerative | | DBSCAN | |
|---|---|---|---|---|---|---|---|---|
| | dim | # Clusters | dim | # Clusters | dim | # Clusters | dim | density |
| RAND | 128 | 5 | 8 | 19 | 128 | 2 | 8 | 0.80 |
| ARAND | 128 | 5 | 8 | 19 | 128 | 2 | 8 | 0.80 |
| MIS | 64 | 29 | 4 | 28 | 128 | 26 | 64 | 0.25 |
| AMIS | 64 | 3 | 8 | 14 | 128 | 2 | 8 | 0.75 |
| NMIS | 64 | 3 | 8 | 14 | 128 | 2 | 8 | 0.75 |
| Hmg | 64 | 29 | 4 | 28 | 128 | 26 | 64 | 0.25 |
| Cmplt | 64 | 3 | 8 | 14 | 128 | 2 | 2 | 0.45 |
| V_meas | 64 | 3 | 8 | 14 | 128 | 2 | 8 | 0.75 |
| **Best** | **128** | **5** | **8** | **14** | **128** | **2** | **8** | **0.75** |

TABLE 2
Contingency matrix for the validation set with 64 dimensions and 3 clusters with K-means

| Clusters No | 1 | 2 | 3 |
|---|---|---|---|
| Benign | 493 | 412 | 34 |
| Malware | 619 | 3 | 359 |
| Cluster label (manual) | Suspicious | Benign | Malware |

TABLE 3
Contingency matrix for the validation set with 128 dimensions and 5 clusters with K-means. (** : manually assigned cluster labels)

| Cluster | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Benign | 44 | 374 | 167 | 14 | 340 |
| Malware | 0 | 0 | 87 | 383 | 511 |
| ** | Benign | Benign | Sus. | Malware | Malware |

### 3.5 Results

The choice of which metric represents the best clustering decision must be studied further. As an example, we will consider the K-means clustering method. Table 2 and Table 3 list the contingency matrices for the validation dataset using 64 dimensions and 3 clusters (Table 2) and 128 dimensions and 5 clusters (Table 3). If we want to incorporate the labels, we can assign manual labels to each cluster to evaluate the test set. As the number of clusters increases, intuitive manual labeling is not feasible. Hence, we limited the maximum number of clusters to 30.

It can be observed that, even with the best parameters, some groupings have significant overlap. This means our vector representation must be further improved to achieve better results. Finally, the clustering models trained using the best hyperparameters are utilized to predict the clustering for the training test. Considering the above predictions 128 dimensions with 5 clusters were chosen for *k*-means clustering. Figure 5 shows the ground truth of the test dataset and the predicted classes with the *k*-means algorithm. Table 4 show the contingency matrix for the test results. As seen the methodology was able to perfectly group cluster 1 and cluster 2 for only benign data, safely grouping about 62% of the benign programs.

Table 5 shows the evaluation scores of the test set for all the clustering algorithms. Unfortunately, the metrics do not have high values. The two-dimensional representation of clustering on the test dataset is shown in Figure 6 (agglomerative clustering), Figure 7 (spectral clustering), and Figure 8 (DBSCAN).

## 4 Discussion

The generated *Graph2Vec* embedding in Figure 4 shows some form of inherent grouping of programs. However when clustering algorithms were employed they were not able perform well.

TABLE 4
Contingency matrix for test set with 128 dimensions 5 clusters with K-means.

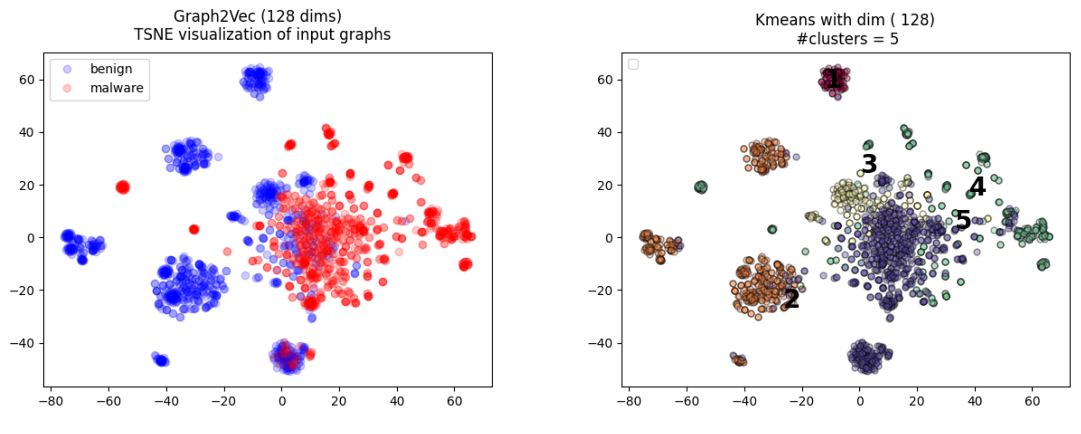
| Clusters No | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Benign | 89 | 531 | 113 | 13 | 254 |
| Malware | 0 | 0 | 90 | 401 | 509 |
| Total | 89 | 531 | 203 | 414 | 763 |

While the visual representation of clusters shows that the methods presented have promise, the numerical performance of unsupervised clustering is not impressive. Supervised clustering and transfer learning have stronger training procedures that result in more accurate classifiers. However, unsupervised methods can be trained without having a priori knowledge of which specific data samples (if any) are malware. Thus, unsupervised clustering may generalize better to never-before-seen types of malicious software.

TABLE 5
Evaluation metric scores for the test dataset with best hyper-parameters

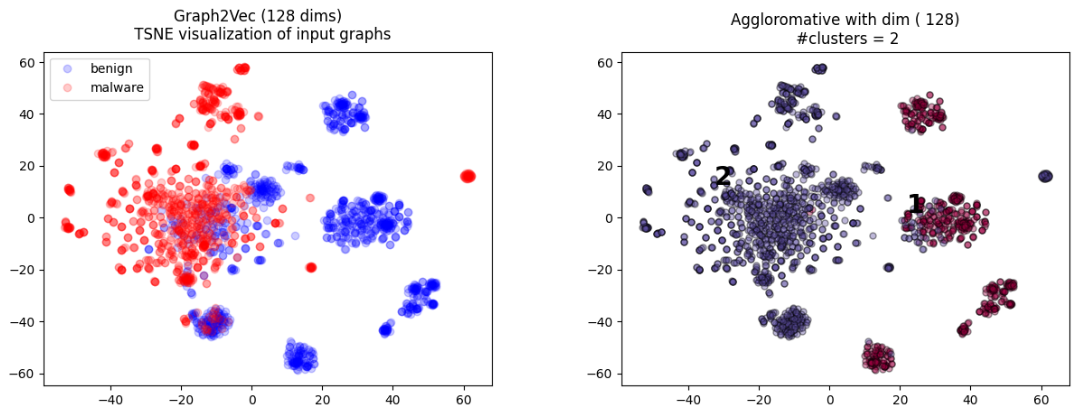
| Metric | K-means | Spectral | Agglomerative | DBSCAN |
|---|---|---|---|---|
| RAND | 0.626 | 0.611 | **0.687** | 0.521 |
| ARAND | 0.252 | 0.221 | **0.374** | 0.042 |
| MIS | **0.352** | 0.333 | 0.282 | 0.323 |
| AMIS | 0.333 | 0.318 | **0.431** | 0.144 |
| NMIS | 0.333 | 0.321 | **0.431** | 0.194 |
| Hmg | **0.507** | 0.480 | 0.407 | 0.467 |
| Cmplt | 0.248 | 0.240 | **0.458** | 0.122 |
| V_meas | 0.333 | 0.321 | 0.431 | 0.194 |

Although the clustering results are not significant, the methods seem promising: some of the malware and benign programs settled into defined groups. However, there are



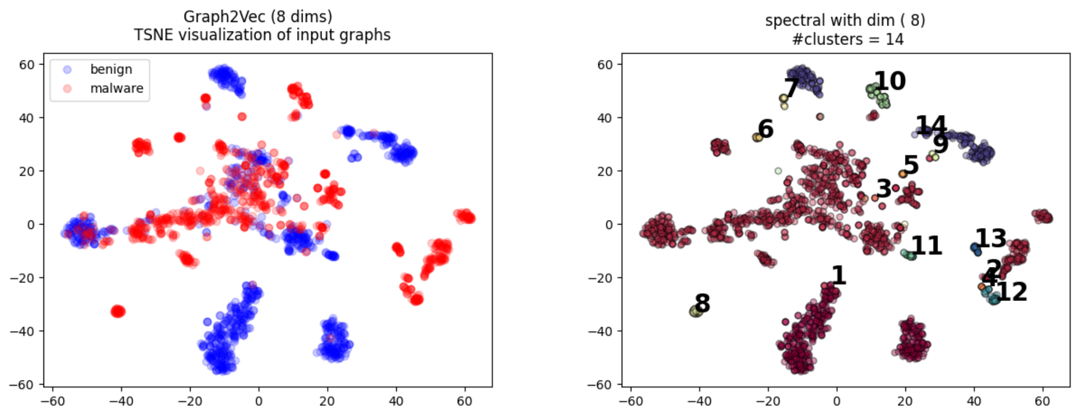
(a) (b)

Fig. 5. K-means clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions.



(a) (b)

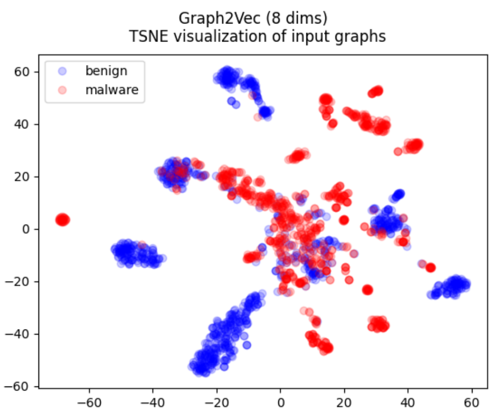
Fig. 6. Agglomerative clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions.



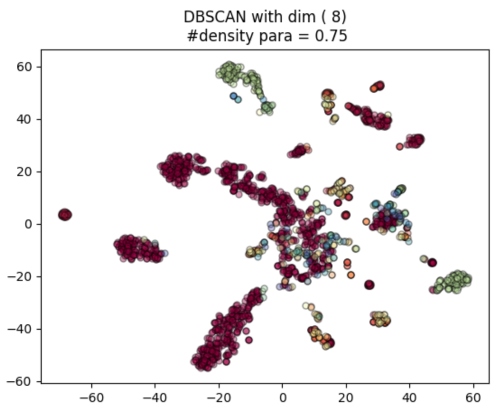
(a) (b)

Fig. 7. Spectral clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions.

groups with significant overlap and there is need of further study to identify the causes. The unsupervised clustering algorithms were only capable of uniquely distinguishing just over half of the programs with high separation. This means the vector representation must be more customized to be able to achieve higher separability between the classes.



(a)



(b)

Fig. 8. Spectral clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions.

## 4.1 Threats to validity

Authors in Arp et el. [18] have recently conducted a study on the effectiveness of the usage of machine learning in the security domain. They have identified ten pitfalls prevailing in the research domain. Our framework is also susceptible to these pitfalls. Some major pitfalls identified according to the criteria are: $(P-1)$ **Sampling Bias**, The dataset we have considered does not represent the whole domain of malware and benign programs. Hence applicability is limited. $(P-3)$ **Data Snooping**, The data is split between training and testing randomly. Since we have not considered temporal distribution or the technological distribution of the programs, it is possible that the training set can contain the latest malware info. Which could reduce the models' performance if unknown or newer programs are presented. $(P-4)$ **Spurious Correlations** The models may learn specific patterns instead of general patterns. Especially due to CFG generation stage, the model might learn artifacts from the data processing stage as its features. $(P-5)$ **Biased Parameter Selection** The selected parameters can be over-fitted to this specific dataset and might not be generalized. Hence the actual performance can be much worse than reported. $(P-6)$ **Inappropriate Baseline** Although we have compared four clustering methods we have only considered a single graph embedding for this dataset. $(P-7)$ **Inappropriate Performance Measures**, The different clustering performance metrics give different scores. Hence further study has to be carried out to identify which metrics work best for this task. $(P-8)$ **Base Rate Fallacy**, The dataset we considered has a balanced number of malware and benign programs and thus might not represent the actual distribution of real-world applications. $(P-9)$ **Lab-Only Evaluation**, The model has not been tested on a real-world setting. Hence further development is required. $(P-10)$ **Inappropriate Threat Model**, Since the machine learning model is susceptible to being exploited by adversaries. An analysis should be conducted regarding the vulnerability of the model.

## 4.2 Future directions

- **CFG generation.** For this project we have used the *angr* python package for CFG generation. However it can be observed that the methods used for generating CFGs are not capable of perfectly constructing CFGs from binary files. Understanding these shortcomings would help better represent binary files as graphs.
- **Node labelling.** In this work, we have replaced the node labels with the number of neighbors the node has. Doing so removes some information about the CFG. However it is challenging to come up with a unifying node labelling system. In CFG generation, when clear function names are not present the *angr* tool assigns a memory location as the node name. However, the memory location is dependent on architecture and the compiler used. Also, node labels can be the function names given by the programmer. Therefore a unifying node labelling procedure is required to label similar functions with similar names. For exmaple, keeping basic function names (eg. *puts()*) intact for WL hash procedure will help preserve some information.
- **Sub-tree structure analysis.** It would be interesting to analyze the relationship between the learned sub-tree structures and the groupings in the vector representations. Since these sub-tree structures correspond to certain coding patterns, it would be interesting to identify and evaluate which patterns are common or have a high occurrence in malware groupings. An example of an idealized backdoor vulnerability is shown in Figure 9 [19][14] that shows a clear graph sub-tree structure. A similar structure may exist in other malware types.
- **Simpler and/or linear graph embedding method.** Since the *Doc2Vec* model is a neural network, it is not intuitive

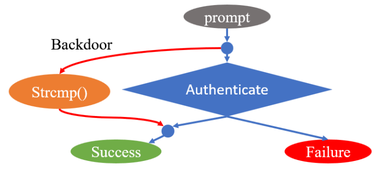


Fig. 9. Conceptual authentication bypass vulnerability

14. https://www.ndss-symposium.org/wp-content/uploads/2017/09/11Firmalice.slide_.pdf

how the final vector representation is related to the original graph sub-tree features. Linear dictionary learning and sparse representation methods could be a candidate for constructing a more intuitive vector representation with a set of known coding patterns.

- **Incorporating graph edge information.** In typical CFGs, the edges do not contain any information or weight, meaning each function call or node connection is considered equal. However, we could incorporate a weight for the edges that represent the calling function's computational complexity. It would help identify if a program keep calling a computationally expensive function. One possibility is to consider the count of assembly code lines in a function as the weight of its incoming edge.
- **Exploring alternate embedding algorithms.** Since the vector representations contain some mixed groupings and many isolated groupings, the evaluated algorithms failed clustering using a low number of clusters. Different embedding or clustering methods could improve performance by maximizing inter-class separation and minimizing intra-class separation in the vector space.

## 5 Conclusion

This paper provides a framework for applying unsupervised clustering for malware detection. The framework employs CFGs and *Graph2Vec* algorithm to represent a program as a fixed length vector. Then several unsupervised clustering techniques were utilized to cluster the vectors. The results are promising in showing that several clusters can be uniquely associated with each class. However, the results are not conclusive and need further evaluation. The paper put forth several challenges in this process, future improvements, and directions. The source code and the dataset is publicly available for further investigation.

## Acknowledgments

The authors would like to thank researchers from Idaho National Lab for their input throughout the manuscript preparation process. We would also like to acknowledge preliminary work done by Veronika Strnadová-Neeley, Daniel Laden, David Opitz, Andrew Rippy, and Shayla Sharma.

## References

[1] V. Strandova-Neeley, D. Laden, R. Pearsall, D. Optiz, A. Rippy, and S. Sharma, "Graph-based analysis of binary code for malware detection and vulnerability identification," *Cyber QR ops report*, 2021.

[2] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017. [Online]. Available: https://arxiv.org/pdf/1707.05005.pdf

[3] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels." *Journal of Machine Learning Research*, vol. 12, no. 9, 2011. [Online]. Available: https://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf

[4] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[5] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[6] B. Rozemberczki, O. Kiss, and R. Sarkar, "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM, 2020, p. 3125–3132.

[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[8] C. Cai and Y. Wang, "A simple yet effective baseline for non-attributed graph classification," *arXiv preprint arXiv:1811.03508*, 2018.

[9] K. Tu, J. Li, D. Towsley, D. Braines, and L. Turner, "Learning features of network structures using graphlets," *arXiv preprint arXiv:1811.03508*, 2018.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[11] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Comput. Surv.*, vol. 52, no. 5, sep 2019. [Online]. Available: https://doi.org/10.1145/3329786

[12] Z. Bazrafshan, H. Hashemi, S. M. Hazrati Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," 05 2013, pp. 113–120.

[13] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. 7, pp. 65 889–65 912, 2019.

[14] Z. Akhtar, "Malware detection and analysis: Challenges and research opportunities," *arXiv preprint arXiv:2101.08429*, 2021.

[15] Z. Liu, C. Chen, A. Ejaz, D. Liu, and J. Zhang, "Automated binary analysis: A survey," in *Algorithms and Architectures for Parallel Processing*. Springer Nature Switzerland, 2023, pp. 392–411.

[16] N. M. Kriege, F. D. Johansson, and C. Morris, "A survey on graph kernels," *Applied Network Science*, vol. 5, no. 1, pp. 1–42, 2020.

[17] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[18] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *Proc. of USENIX Security Symposium*, 2022.

[19] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *NDSS*, 2015.

**Kaveen G. Liyanage** received his BS in Electrical and Electronic Engineering from the University of Peradeniya, Sri Lanka. He is currently a graduate student at Montana State University in Bozeman, MT. His research focuses on signal processing, sparse representation, and machine learning.

**Reese Pearsall** received his BS and MS in Computer Science from Montana State University in Bozeman, MT. He is currently an Instructor of Computer Science and Cybersecurity at Montana State University. His research interests include cybersecurity, malware analysis, malware detection, and cybercrime.

**Clemente Izurieta** received a BS in Mathematics from the University of Wollongong, and an M.S. in Computer Science from Montana State University. Dr. Izurieta received his Ph.D. in Computer Science from Colorado State University and is now a Professor at Montana State University in Bozeman, MT. His research focuses on Empirical Software Engineering, Quality Assurance, and Technical Debt.

**Bradley M. Whitaker** received his BS in Electrical Engineering from Brigham Young University and his M.S. and Ph.D. in Electrical and Computer Engineering from the Georgia Institute of Technology. Dr. Whitaker is now an Assistant Professor at Montana State University in Bozeman, MT. His research focuses on signal processing and machine learning, with applications in healthcare, military surveillance, and remote sensing.