AN EVALUATION OF GRAPH REPRESENTATION OF PROGRAMS FOR MALWARE DETECTION AND CATEGORIZATION USING GRAPH-BASED MACHINE LEARNING METHODS

by

Reese Andersen Pearsall

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

August 2023

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Bradley Whitaker and Kaveen Liyanage. Without their guidance through the machine learning aspects of this work, this thesis could not have been possible. Next, I would like to thank Dr. Clemente Izurieta for serving as the head of my thesis committee. I would also like to acknowledge Dr. Travis Peters and Dr. Derek Reimanis for serving on my thesis committee. I would like to acknowledge and thank Dr. John Paxton for his support and patience during my time in graduate school. Lastly, I would like to thank my family, friends, and students for always supporting and keeping me afloat during my undergraduate and graduate school career.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With both new and reused malware being used in cyberattacks everyday, there is a dire need for the ability to detect and categorize malware before damage can be done. Previous research has shown that graph-based machine learning algorithms can learn on graph representations of programs, such as a control flow graph, to better distinguish between malicious and benign programs, and detect malware. With many types of graph representations of programs, there has not been a comparison between these different graphs to see if one performs better than the rest.

This thesis provides a comparison between different graph representations of programs for both malware detection and categorization using graph-based machine learning methods. Four different graphs are evaluated: control flow graph generated via disassembly, control flow graph generated via symbolic execution, function call graph, and data dependency graph. This thesis also describes a pipeline for creating a classifier for malware detection and categorization. Graphs are generated using the binary analysis tool *angr*, and their embeddings are calculated using the Graph2Vec graph embedding algorithm. The embeddings are plotted and clustered using K-means. A classifier is then built by assigning labels to clusters and the points within each cluster.

We collected 2500 malicious executables and 2500 benign executables, and each of the four graph types is generated for each executable. Each is plugged into their own individual pipeline. A classifier for each of the four graph types is built, and classification metrics (e.g. F1 score) are calculated. The results show that control flow graphs generated from symbolic execution had the highest F1 score of the four different graph representations. Using the control flow graph generated from symbolic execution pipeline, the classifier was able to most accurately categorize trojan malware.

INTRODUCTION

The creation of new malware continues to surge as internet usage increases and new software vulnerabilities are discovered. New strands of malware are utilized in different types of cyberattacks to cause significant damage or financial impact on an individual or a company. It was estimated in 2018 that damages associated with ransomware were expected to reach $8 billion[3] dollars. As a result, it is vital to detect and quarantine malware before it can cause any damage.

The branch of cybersecurity that focuses on spotting and quarantining malware is referred to as malware detection. According to a study done by Dataprot and AV-Test Institute[1], it was found that 560,000 new pieces of malware were found every day. It is common for new malware to surface when new software vulnerabilities are found. An example of this is the Log4j vulnerability[2], which was discovered in the java-based logging utility, Apache Log4j. The Log4j vulnerability allowed malicious actors to execute arbitrary code on the victim's machine running versions 2.0 to 2.15 of the Log4j library. Many corporations using the Apache Log4j tool were targeted, with the most prominent case being the Belgian Defense Ministry[3]. New groups of malware, particularly ransomware, were created to exploit the Log4j vulnerability. One example is NightSky[4], a ransomware originating from China that exploits the Log4j vulnerability by encrypting the victim's files until a ransom is paid. Pandora[5] is another ransomware that appeared in February of 2022 during the Log4j vulnerability time frame, and is very similar to NightSky. In fact, both

[1]https://dataprot.net/statistics/malware-statistics/

[2]https://www.forescout.com/resources/night-sky-ransomeware-threat-brief/

[3]https://www.zdnet.com/article/belgian-defense-ministry-confirms-cyberattack-through-log4j-exploitation/

[4]https://www.forescout.com/resources/night-sky-ransomeware-threat-brief/

[5]https://www.inceptionsecurity.com/post/an-encounter-with-pandora

Pandora and NightSky are products of an older piece of ransomware called Rook. This means they have very similar source code that is slightly altered to make it look like something new; a very common tactic for malware authors. NightSky and Pandora, while variants of an older malware, are evidence that it is vital to be able detect new pieces of malware before damage can be done.

Software that tries to detect malware, often referred to as "antivirus software," may fail to detect new malware files. Traditional commercial antivirus software falls short due to reliance on signature-based detection methods[20]. Signature-based detection will look for a "signature," or specific pattern in the file and check if the signature matches an already detected malware sample. When new malware files surface, signature-based antivirus software ultimately fails because a known signature will not appear in the file. Therefore, a more creative approach is needed to detect new strands of malware.

One modern method for detecting malware is by leveraging machine learning. There are several different techniques for using machine learning to detect malware, with one being graph-based methods. Graph-based detection is a method that uses graphs, a set of nodes and edges, to train a machine learning model [20]. Converting binary executables into graphs can result in representations such as control flow graphs, call reference graphs, and import graphs. Different graph representations can capture different aspects and relationships of the binary. Graph embedding algorithms are used to capture the topology and features of a graph and output a vector representation, which can then be used as input to a machine learning algorithm. Relevant literature shows that graph-based detection methods have been successful at detecting malware and distinguishing malicious versus benign code [41, 45]. However, there is a gap for using graph-based methods in an unsupervised fashion with clustering. Most relevant literature utilizes supervised methods for detecting malware using graph representations. In addition, many of these studies also used different types of graphs, and an overall comparison of different graph types has never been conducted. Additionally, a measure of how effective graph-based methods are at categorizing malware has not been done in previous work. With the many different types and behaviors of malware, it can be beneficial to know if the malicious file being analyzed is ransomware, or a virus for example.

The scope of this thesis is to evaluate graph-based machine learning methods for malware detection and categorization using different types of graph representations generated from a binary executable. The work done in this thesis leverages both supervised and unsupervised machine learning methods. The graph embeddings are generated using unsupervised machine learning algorithms, as well as the clustering of the embeddings in an N dimensional space. Once clustering has been done, the true labels are assigned to each data point to evaluate how effective the unsupervised algorithms performed at clustering benign data points together and malicious data points together. The goal is to identify if one graph representation performs better at detecting malware and distinguishing between different types of malware over another. Chapter **2** provides relevant background information about malware detection methods, graph embedding, and different graph representations of programs. Existing literature and tools for malware detection that utilize graph-based methods are reviewed in chapter **3** via an informal literature review. Next, in chapter **4**, the motivation for analyzing the effectiveness of graph-based techniques for malware detection and research goals for the thesis are listed. In chapter **5**, the methodology and approach for testing the effectiveness of graph-based techniques and explanation for data collection and data analysis are described. The results of our analysis and experiments are found in chapters **6** and **7**. Lastly, the conclusion and discussion of the results are provided for future work in chapter **8**.

BACKGROUND

Malware

Malware, which is short for **mal**icious soft**ware**, is a general term used to describe an unwanted, unauthorized computer program or script with malicious intent to cause some form of harm. System administrators are most often unaware of the presence and behavior of malware, and if they were to be aware, they would not permit it run on the system [31]. Malware typically attempts to gain unauthorized access in order to steal sensitive or financial information, disrupt services, or gain remote control access for later use [1].

Types of Malware

Malware can be categorized into different families. All malware attempt to cause some kind of damage or harm, but the different families of malware have different strategies for compromising a system. For example, a certain family of malware may only stay isolated and hide itself on a system versus another family of malware that may attempt to spread across several devices on a network. Knowing the family of malware can be vital in responding to an incident and can provide helpful feedback as to how to prevent future attacks. While there are potentially hundreds of different families of malware, the following types of malware appear frequently:

A *trojan* malware, or sometimes referred to as a *trojan horse* is a type of malware that typically disguises itself as a legitimate program or file with the goal to deceive users and gain unauthorized access to their system. Once a trojan is installed, then a trojan can deploy a variety of attacks to cause harm such as, deleting data, stealing data, or installing a backdoor or keylogger. Trojans are a commonly used family of malware, as they are easy

[1]https://www.paloaltonetworks.com/cyberpedia/what-is-malware

to deploy in common attack vectors, such as social engineering.

A *worm* is a type of malware with the distinct characteristic of self-replication. A worm can spread across computer networks without requiring user interaction. Unlike trojans, worms do not need to attach themselves to existing files or programs to propagate. Instead, they exploit vulnerabilities in network protocols or operating systems to independently replicate and spread from one computer to another.

A *virus*, or commonly referred to as a computer virus, is a type of malware designed to infect, replicate itself, and spread from one device to another. Unlike worms, a virus must attach itself to a legitimate programs or files.

A *dropper* is a type of malware that is designed to deliver and install additional malware onto a targeted system. It acts as a carrier or container for other malicious components, which are often more complex and sophisticated in nature. Similar to a trojan, the primary purpose of a dropper is to bypass security measures and facilitate the deployment of the payload onto the victim's computer.

Malware Detection Methods

A variety of different malware detection techniques, both old and new, are practiced in the field of cybersecurity. Bazrafshan et. al [5] classifies malware detection into three distinct categories: Signature-based, Behavioral-base, and Heuristic-based.

Signature-based Detection

The most common method for detecting malware is signature-based detection. This technique involves searching for a known digital footprint that has been detected and recorded in the past. Antivirus software that utilizes signature-based detection will scan a file and look for different types of signatures and then check it against a large database of known signatures. If a signature from the file appears in the malicious signature database, the file will be flagged as malicious and quarantined before it can cause any damage to the system.

A frequently used signature is a file hash, a unique identifier that is calculated from the executable. Common hashing algorithms such as MD5 and SHA256, can be calculated relatively quickly from a binary, and along with their ability accurately detect previously-seen malware, makes them a quick and easy option for detecting malware [17]. VirusTotal [2] is one of many tools that can be used to check the hash of a potentially malicious file. VirusTotal is a community-based website that has a large repository of millions of known pieces of malware. When a user wants to inspect a malicious file, URL, or IP Address, it checks it against this large database of known malware.

Another signature-based technique is scanning a binary for a sequence of bytes that represents a known malware signature. This can be a string such as an IP address, URL, or function name that exists in the malware code. Another byte sequence that can be scanned are known malicious coding constructs. For example, the Nimdra worm, a piece of malware that infected systems in the early 2000s contained the following code[46]:

```
<html> <script language =‘javascript’> window.open(‘readme.eml’) </script>
```

Every Nimdra worm or variants would contain this code block, which is an instruction to open up an internet browser to start spreading itself via email. Signature-based systems can scan the file and look for the byte sequence that represents this signature. This searching process is bounded by the pattern matching problem, which currently cannot be solved more efficiently than O(n + m) [14], where n is the length of the target text being searched and m is the length of the signature that is being searched for. This can be cumbersome when the file being analyzed is large in size or when there are many signatures to look for. Another attractive feature of signature-based detection is the fact it relies solely on static analysis

[2]https://www.virustotal.com/

methods, so executing the malware for analysis and potentially infecting a system is not required when searching by file hash or scanning for a signature.

The major downside of signature-based detection is its inability to detect new malware and how easy it is to avoid being detected by signature-based antivirus software. The most common tactic to avoid signature-based detection is through code obfuscation. This includes adding junk code and NOP instruction sleds. These are bogus functions or operations that will not change the behavior of the malware, but will still alter the structure of the binary, which results in a different file hash, for example. Malware authors will also change function names and order of instructions within the program to avoid detection. A very small change to the source code of malware will result in an entirely different hash, which is known as the avalanche effect [38]. Obfuscating source code or portions of the source code will culminate an entirely new hash value that the antivirus has never seen before and will be classified as benign. A vast majority of malware that is found each day is unique. In 2017, it was found that 93% of malware was unique[10].

Encryption is another evasion method for bypassing static antivirus software. Malicious payloads or sections of malware can be encrypted before being scanned and can decrypt the payload during execution to infect a system. Two common types of viruses that utilize obfuscation to bypass antivirus software are polymorphic viruses and metamorphic viruses[25]. Polymorphic viruses obfuscate its structure and decryption cycle by adding junk instructions, NOP sleds, and program jumps, all while maintaining the semantics of the original virus[5]. It was found that 94% of viruses found in 2017 were polymorphic in nature [10]. Metamorphic viruses, on the other hand, attempt to encrypt the entire virus and behavior of the virus. This includes tactics such as register reassignment, code transposition, and changing of conditional jumps [35]. Detecting these types of viruses using static methods becomes difficult because the malicious signature of the program can change whenever the virus replicates itself. Antivirus systems are able to detect if a program is encrypted or not, and it is possible for an antivirus software to attempt to encrypt files, but it is very unlikely that the virus can be decrypted. Additionally, new mutants of the virus receive a new decryption key after replication, so if a key is able to be retrieved, it may not be able to decrypt all other viruses on a system. An emerging cyber attack method is fileless malware, or non-malware, which is malware that is injected and exists solely in main memory [2]. Fileless malware does not write any data to disk, and the source code does not exist on disk, which makes it nearly impossible for antivirus software or a digital forensics analyst to detect.

Behavioral-based Detection

Due to how simple it is to bypass antivirus software that uses static detection, more creative methods are required for detecting new strands of malware. Behavioral-based detection uses dynamic analyses for evaluating malware, which is the process of analyzing a code or script by executing it and observing its actions [31]. This includes analyses of binary instructions that are executed, monitoring activity in the windows registry while the malware is running, and monitoring of live RAM memory while malware is executing. Because behavioral-based methods execute malware and observe its behavior instead of analyzing static characteristics of the binary file itself, behavioral-based methods are not subject to many of the evasion techniques that cripple static-based methods. However, as malware has evolved and adapted to common antivirus techniques, malware authors have developed ways for hindering dynamic analysis and evading behavioral-based detection methods.

Malware authors can wrap their malicious payloads in a logic bomb, which is a piece of code designed to test various conditions about the environment [31]. This can include checking the presence of a debugger, sandbox environment, and analysis tools, which are all common mechanisms that are used in dynamic analysis. If a piece of malware using a logic bomb detects that it is being executed within a sandbox environment, virtual machine

(VM), or hypervisor, the malicious portions of the code will not be executed and thus the malicious behavior will not be seen. This can be as simple as checking for VM-specific processes and artifacts, other user activity on the system, and system registry values [47]. Another evasion technique, often referred to as DLL injection or DLL hollowing, is to insert code into a trusted, legitimate process[47]. Analyzing a malicious program that utilizes DLL injection or hollowing becomes difficult because the malicious behavior does not come from the malware itself, but rather a trusted process that was spawned by the operating system. Another downside to dynamic analysis methods is that it is expensive, and carries some overhead with it, as it typically requires some sandbox or dedicated environment for the malware to infect.

Heuristic-based

As a solution to overcome the difficulties with static-based methods, Heuristics-based detection uses machine learning and data mining to classify and detect malware. In order to leverage machine learning to classify malware, features must be extracted from a program. Operating System API (Application Programming Interface) calls, or *system calls* as they are known in Unix, are one feature that can be collected. Operating System API calls provide an interface between the operating system and programs, and are used every time an application request some kind of resource that is serviced by the operating system, such as file access, memory access, and network access. The use of API calls for heuristic-based detection has been a heavily researched area, including Iwamoto and Wasaki [21], who used static analysis tools to extract API call sequences which were used for malware classification using a hierarchical cluster analyzer.

Another feature that can be extracted for heuristic-based detection are Operational Codes (OpCodes). Each machine instruction that is executed by a computer has an OpCode, which specifies the operation that needs to be done by a computer. While the names and number of opcodes varies by processor, these generally consist of a small pool of very basic instructions such as *add* (add two numbers), *mov* (move a value), and *jmp* (perform a conditional jump). Studies such as Bilar [6], found that the distribution, frequency, and pairings of OpCodes can be used in heuristic-based malware classification programs as a feature to measure how malicious a binary is.

Malware can also be modeled as an N-gram. N-Grams are a method most commonly seen in natural language processing and modeling, where a String, or sequence of characters can be expressed by a contiguous series of N items. For example, consider the string "The software is not evil". The 1-Grams, 2-Grams, and 3-Grams for this string are modeled in table 2.1 below

Table 2.1: N-grams for string "The software is not evil"

1-Gram	2-Gram	3-Gram
The	The software	The software is
software	software is	software is not
is	is not	is not evil
not	not evil	
evil		

Someone that is evaluating this string simply by looking at just 1-Grams, one would see the word "evil" and make a prediction that the meaning of this text is that the software is evil, which is an incorrect position. By evaluating all N-grams, additional context can be revealed about the text, which will help lead to a more accurate prediction about the meaning of the text. The binary source code of a program can be modeled as an N-gram to reveal context and behavior of program, which can then used by a machine learning algorithm to predict if that program is malicious or benign. Tesauro et al. [1] created a classifier built from N-grams of length 1 to N-grams of length 10 from a series of 65 benign and malicious Windows executable files and was able to achieve up to 98 percent accuracy for identifying benign and malicious executables.

VirusTotal

Determining the family of malware a malicious binary belongs to typically requires static or dynamic analysis. This can be time consuming, especially when done manually. To automate the process of determining malware families, a tool called VirusTotal[3] can be used. A user can provide a hash of a file, and if VirusTotal has seen that file before, it will return back all information and metadata about that file. One piece of information that is returned is the predicted malware family, which has been determined by VirusTotal or another community user that has done analysis on the file. Other pieces of information that is returned is whether or not the file is malicious, header information, file size, and how many times it has been scanned on VirusTotal.

Graph Representation of Programs

A graph is a data structure that is defined by a set of nodes (or sometimes known as vertices) that are connected by edges. Formally, a graph G = (V,E) where V represents a set of vertices or nodes, and E = (u,v) represents some edge between two vertices, u and v. Edges capture relationships between nodes. For example, if a graph is used to model a social network, then an edge would represent a friendship or relationship between two users. Graphs are a very versatile data structure, and can be used to capture the structure of many types of systems, such as road networks, internet routing, and flow networks.

Graphs can also be used to represent the structure of a program. There are many types

[3]https://www.virustotal.com/

of graphs that attempt to model a binary program, and the specifics differ from graph to graph. Typically, the nodes of these graphs represent some code statement or block of code, and the edges represent the flow of control between these statements. Graph representations of programs can be helpful for analysis, deriving program behavior, debugging, and code optimization. Another benefit is that once a graph representation is generated, traditional graph algorithms can be applied to solve a variety of problems, such as generating test paths, shortest paths, and a minimum spanning tree. Additionally, graph representations can be generated without running the actual program. Because of this, graph representations of programs are a useful tool for analyzing the behavior of malware, as it does not require malware execution, which is a risky process.

Control Flow Graph

A control flow graph (CFG) [8] [42] is a graph representation with the primary goal of capturing the control flow of some program. The control flow of a program refers to the sequence of statements that are executed by the program during program execution. The control flow of a program greatly depends on the various program constructs, such as if statements, loops, and function calls. A CFG tries to capture all possible branches of a control flow in a program. In a CFG, the nodes represent some program statement or block of code, and the edges (typically unidirectional), represent the blocks of code that can be reached from other blocks of code. For example, an *if/else* statement may branch off into two edges that connect to two different nodes. A *for* or *while* loop may result in multiple branches, with one edge connecting to a node earlier in the graph. Control flow graphs are often used in program analysis and optimization. They provide a visual representation of the program's control flow, making it easier to reason about its behavior and perform various static analysis techniques. Control flow analysis, for example, can be applied to extract information about the control flow, such as detecting unreachable code, identifying loops, or determining the set of possible paths through the program.

There are various ways to generate a control flow graph using static tools. Two of the most common methods are program disassembly and symbolic execution.

Via Disassembly Disassembly [32] refers to the process of converting executable machine code or binary instruction into human-readable assembly code. Disassembled code provides a low-level representation of the program's instructions, making it easier to analyze and understand how the program operates at a lower level. Disassembly is often used in reverse engineering, vulnerability analysis, debugging, and understanding the inner workings of programs or system software. Disassembly is done using disassemblers, which are software tools or utilities specifically designed to perform the disassembly process. Disassemblers typically read the binary executable file and interpret the machine code instructions, mapping them to corresponding assembly instructions. The resulting disassembled code can be viewed in a human-readable format, usually displaying mnemonic instructions, memory addresses, register values, and other relevant information. Once the assembly instructions for a program are generated, a control flow graph can be created by analyzing the control flow of the different assembly instructions.

Symbolic Execution Symbolic execution[33] is a technique used in software analysis and testing to reason about the behavior of a program without actually executing it with concrete inputs. Instead of using specific input values, symbolic execution operates on symbolic representations of input variables, treating them as variables with unknown values. During symbolic execution, the program's code is executed symbolically, meaning that instead of performing concrete computations, the execution tracks constraints and dependencies on the symbolic input variables. As the program encounters conditional branches, the execution branches into multiple paths, each representing a different branch outcome based on the symbolic inputs. To fully execute a program symbolically, the execution engine needs to handle operations on symbolic values and track the constraints associated with each execution path. Symbolic execution engines often employ constraint solvers to reason about the constraints and generate concrete inputs that satisfy specific program conditions. Symbolic execution offers powerful capabilities for program analysis, and can be used to explore the behavior of a program without executing it. While symbolic execution is happening, a control flow graph can be constructed in parallel while the symbolic execution tool is exploring all possible paths.

Call Reference Graph

While a CFG graph captures the control flow within functions, a Call Reference Graph (CRG) [36], or sometimes known as a invocation graph captures the control flow between functions or methods. This graph provides a visual representation of how functions are invoked or called by each other during program execution. In a CRG, nodes represent individual functions (which may consists of several or many lines of code), and the edges represent a calling relationship between them. If function A calls function B, there will be a directed edge from A to B, which means control of the program is handed from function A to B during program execution. Similar to CFGs, a CRG is a useful tool for analyzing the behavior and control flow of a program between functions without needing to execute a program. A CRG can reveal interesting information about a program, such as the use of recursive calls, unused or bogus functions, and cyclic dependencies. A CRG can be generated through both static and dynamic analysis tools, however when considering the use of a CRG for malware analysis, generating a CRG with static tool will be an ideal option as the user will not have to risk infection from running the program. Static tools can use methods such as disassembly or symbolic execution.

Data Dependency Graph

A data dependency graph (DDG), also known as a data flow graph, is a graphical representation of the dependencies between different data elements or operations in a computational system or program. Rather than capturing the control flow of a program like in a CFG, it illustrates how data is produced, consumed, and transformed within the system. In a data dependency graph, nodes represent individual operations or data elements, while edges indicate the flow of data between them. Each node typically corresponds to a specific task or operation, and the edges show the dependencies between these tasks based on the data they require or produce. One of many types of dependencies that a DDG could capture is a Read-after-write dependency. For example, if node A produces some data, and a subsequent node B reads that data as input, then B would depend on A because it requires the output that was produced from node A. In the DDG, a directed edge would be created that goes from node B to node A.

Data dependency graphs are helpful for understanding the order of operations that need to be followed when while compiling a program. By following a path in a data dependency graph, this will ensure the correct order of operations are followed to ensure correct results and maintain data consistency.

Graph Embedding Algorithms

Graph embedding algorithms [15] [30] are techniques used to represent a graph data structure in a lower-dimensional vector space while preserving the structural information and relationships between nodes in the graph. The goal of graph embedding is to transform the complex graph structure into a continuous vector representation, which can then be used as input for machine learning models or other downstream tasks.

Graph embedding algorithms aim to capture various aspects of the graph structure, such as node similarity, node proximity, or graph connectivity. These algorithms leverage graph topology and potentially other attributes associated with nodes and edges to generate meaningful embeddings.

Unsupervised Clustering

Unsupervised clustering is a machine learning technique where an algorithm tries to group the data points in the datasets 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). There are a variety of different unsupervised clustering algorithms, which include k-means, spectral, DBSCAN, and agglomerate 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. *scikit-learn* is a Python module that has API calls for each of these clustering algorithms. 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. While there are many unsupervised clustering algorithms that exists, in this work, k-means will be the only unsupervised clustering algorithm that is used. This is because k-means is a widely used and popular clustering algorithm. It provides simplicity, scalability, and efficiency, while also generally providing reliable results.

SUPPORTING WORK

Representing Malware as a Graph

Bilot et. al [7] provides a survey on graph representations of programs for malware detection. The authors provide several graph representations that have been used for building a classifier for malware detection with either supervised or unsupervised learning for detecting Android or Windows malware. These graphs include a control flow graph (CFG) [11], function call graph (FCG) [13], program dependency graph (PDG) [43], system call graph [23], network flow graph [24], and system entity graph [26]. Each of these graph representations has been used for malware detection in previous work, however a comparison between these graph representations has not been done.

Graph Embeddings for Capturing Program Structure

Strandova-Neeley et al. [37] provided an initial framework and results for testing the effectiveness of graph embedding methods for capturing program structure. Additionally, the authors also used the graph embeddings in unsupervised machine learning methods to measure the performance of various clustering techniques. One of the curated datasets was the distinct benign code (DBC) dataset, which consisted of 49 variations of five basic C++ programs: *fibonacciSequence*, *isPrime*, *productPrices*, *randomList*, and *repeatingString*. Control flow graphs were also generated using angr[1], a binary analysis tool. Because the variations of each base program were constructed by adding basic statements that do not change the overall functionality of the program, then the control flow graphs that are generated from the variations should also be structurally similar. After clustering the control flow graph embeddings from the DBC dataset, the results showed that variations of the same

[1]https://angr.io/

program clustered together when using the DBSCAN and Agglomerative clustering methods. This study performed by Strandova-Neeley et al. shows that graph embedding methods on control flow graphs do an effective job at capturing the structure of a graph representation of a program, and can be used in unsupervised clustering methods. The work did not address any experiments on how graph embeddings of graph representations of programs can be applied in the field of malware detection and classification, but was suggested as future work.

Graph Embeddings for Malware Detection

There has been a significant amount of work that has shown that graph representations of programs and machine learning can be used for detecting malware. Yan et al. [45] used CFGs of over twenty thousands samples of malware to create a convolutional neural network malware classifier. Using this malware classifier, Yan et al. were able to achieve over 99% accuracy on a dataset of new malware. Jiang et al. [22] used graph embeddings generated from a series of function call graphs to create a malware classifier, and were able to show high accuracy. Both Yan and Jiang's work focused on one graph representation of a program, and did not experiment with multiple graph representations. Additionally, both assessments did not use any unsupervised clustering techniques, relying on other deep learning machine learning methods.

Graph Embeddings for Malware Categorization

Using graph embeddings for categorizing malware into distinct families (virus, worm, adware, etc.) is a well-established area of research. Fan et al. [12] used function call graph embeddings generated from android malware and, in combination with unsupervised clustering methods, were able to cluster malware into similar families based on their graph embedding. Their results showed that their malware familial analysis framework outperformed other state-of-the-art approaches in terms of accuracy and efficiency. Lo et al. [29] generated function call graphs from a set of Android malware and used graph embeddings from the graph to train a graph neural network. This neural network was used in a multi-class classifier and was able to achieve an average F1-Score of 0.97. However, in both research papers, they focused on one graph type and did not exercise their experiments on various types of graph representations.

Conclusion of Supporting Work

It is clear that embeddings of graph representations of programs, such as control flow graphs or function call graphs are an effective structure for use in deep learning algorithms to detect and categorize malware. All reviewed literature only focused on one graph representation of malware, and a comparison between different graph types was not done. While there are many types of graph representations, if no comparison between graph types is done, then there is no guarantee that the work done in the previous literature was using the ”best” graph representation of a program. By addressing this gap, we can help ensure that a comparison between different graph types has been done, and thus can help provide more accurate results for detecting and categorizing malware. After conducting an informal literature review and discussing existing literature of malware detection and categorization using graph embeddings in the previous sections, a comparison between graph representations of programs needs addressing.

RESEARCH GOALS

The goal of this research is to evaluate different graph representations of programs and to determine how effective each is at capturing the structure of a binary executable for detecting and categorizing malware.

Motivation

In today's cyber world, it is vital for endpoint systems or firewalls to be able to detect malware before it can cause any damage. Being able to detect malware in an accurate, timely manner can severely reduce the likelihood of cyber attacks or breaches from occurring, thus saving a company potentially millions of dollars in damage. If a system does become compromised, it is important for the responders to determine what type of malware compromised the system, and the behavior of the malware. If responders know the type or family of malware, that information can help them determine the scope of the attack, and the amount of damage that may have been caused. In chapter 3, it is discussed that using graph representations of programs and graph embedding for malware detection and categorization has already been shown to be an effective method. However, there is no existing literature that compares several graph representations against one another. By comparing different graph represenations of programs, one can determine if there is a "best" graph representation that does a good job at capturing the structure and behavior of a program. Determining which graph representation is "best" for detecting and categorizing malware, those results can help future work that leverages graph-based machine learning methods for malware detection and categorization.

GQM

The Goal-Question-Metric (GQM) [4] is a framework for achieving research goals by answering a set of questions through certain metrics. Metrics will help answer questions, which will in turn help accomplish a set of research goals. The GQM approach is used in this research, and the goals, questions, and metrics and presented below.

Goal: To improve our ability to analyze malware with graph-based machine learning methods by evaluating and comparing different graph representations of programs.

Research Question 1 How do different graph representations of programs compare to one another when used for detecting malware?

Research Question 2: Using that graph representation from question 1, how well does that graph perform at *categorizing* the different types of malware?

The important metrics to help answer the research questions are derived from the predictions that the machine learning classifier makes. When the classifier makes a prediction, there are four possible outcomes:

1. True positive (TP) - the classifier correctly predicts a piece of malware to be malicious
2. True negative (TN) - the classifier correctly predicts a benign program to be benign
3. False positive (FP) - the classifier incorrectly predicts a piece of malware to be benign
4. False negative (FN) - the classifier incorrectly predicts a benign program to be malicious

Metric 1: Accuracy- Accuracy is a metric that measures the proportion of correct predictions made by a model out of the total number of predictions. Accuracy measures the percentage of the benign and malicious programs the model correctly flags, out of all the programs in the data set.

$$Accuracy = (TP + TN)/(TP + TN + FP + FN) * 100$$

Metric 2: Recall- Recall is a metric that measures the proportion of actual positive instances that are correctly identified by the model. This measures the proportion of malware that was actually identified as malware.

$$Recall = (TP)/(TP + FN) * 100$$

Metric 3: Precision- Precision is a metric that measures the proportion of correctly predicted positive instances out of the total instances predicted as positive by the model. This measures the proportion of predicted identified malware that was actually malware.

$$Precision = (TP)/(TP + FP) * 100$$

Metric 4: F1 Score- F1 score is a metric that is a harmonic mean between recall and precision. F1 score is a helpful metric when the cost of false positives and false negative is uneven. Because flagging malware as benign is much more dangerous that incorrectly flagging benign code as malicious, F1 score is going to be the primary metric to evaluate the performance of our classifier.

$$F1Score = 2 * (Precision * Recall)/(Precision + Recall)$$

Metric 5: Confusion Matrix- While a confusion matrix is not a single-value metric, it is a table that summarizes the performance of a classifier by showing the counts of true positives, true negatives, false positives, and false negative predictions. This table is helpful to compute so that metrics such as accuracy, precision, recall, and F1 score can also be computed.

Metric 6: Contingency Matrix- Similarly to a confusion matrix, a contingency matrix is not a single value, but rather provides a table of values. A contingency Matrix is computed once labels have been assigned to data points inside of a cluster. The matrix will break down how many true benign programs are in each cluster, and how many true malicious programs are in each cluster.

Figure 4.1 below shows a diagram of the GQM model.

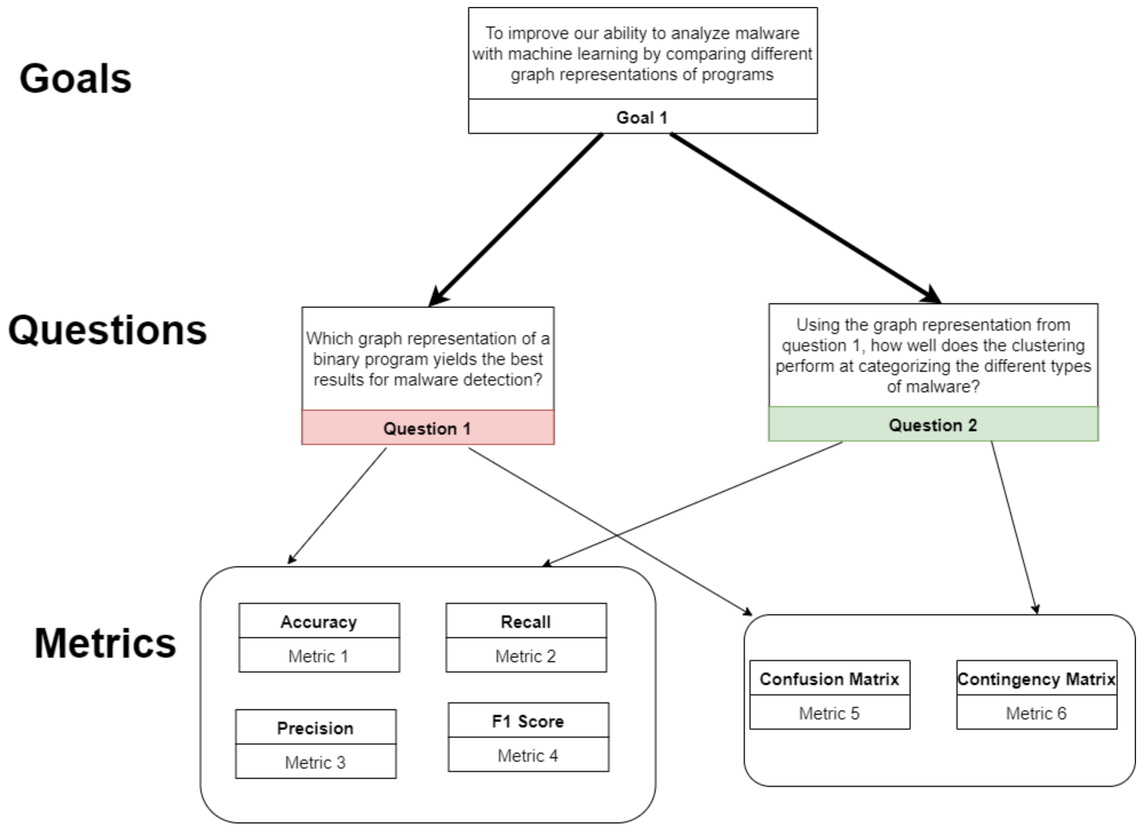

Goals
To improve our ability to analyze malware with machine learning by comparing different graph representations of programs
Goal 1
Questions
Which graph representation of a binary program yields the best results for malware detection?
Question 1
Using the graph representation from question 1, how well does the clustering perform at categorizing the different types of malware?
Question 2
Metrics
Accuracy
Metric 1
Recall
Metric 2
Precision
Metric 3
F1 Score
Metric 4
Confusion Matrix
Metric 5
Contingency Matrix
Metric 6

Figure 4.1: GQM Diagram

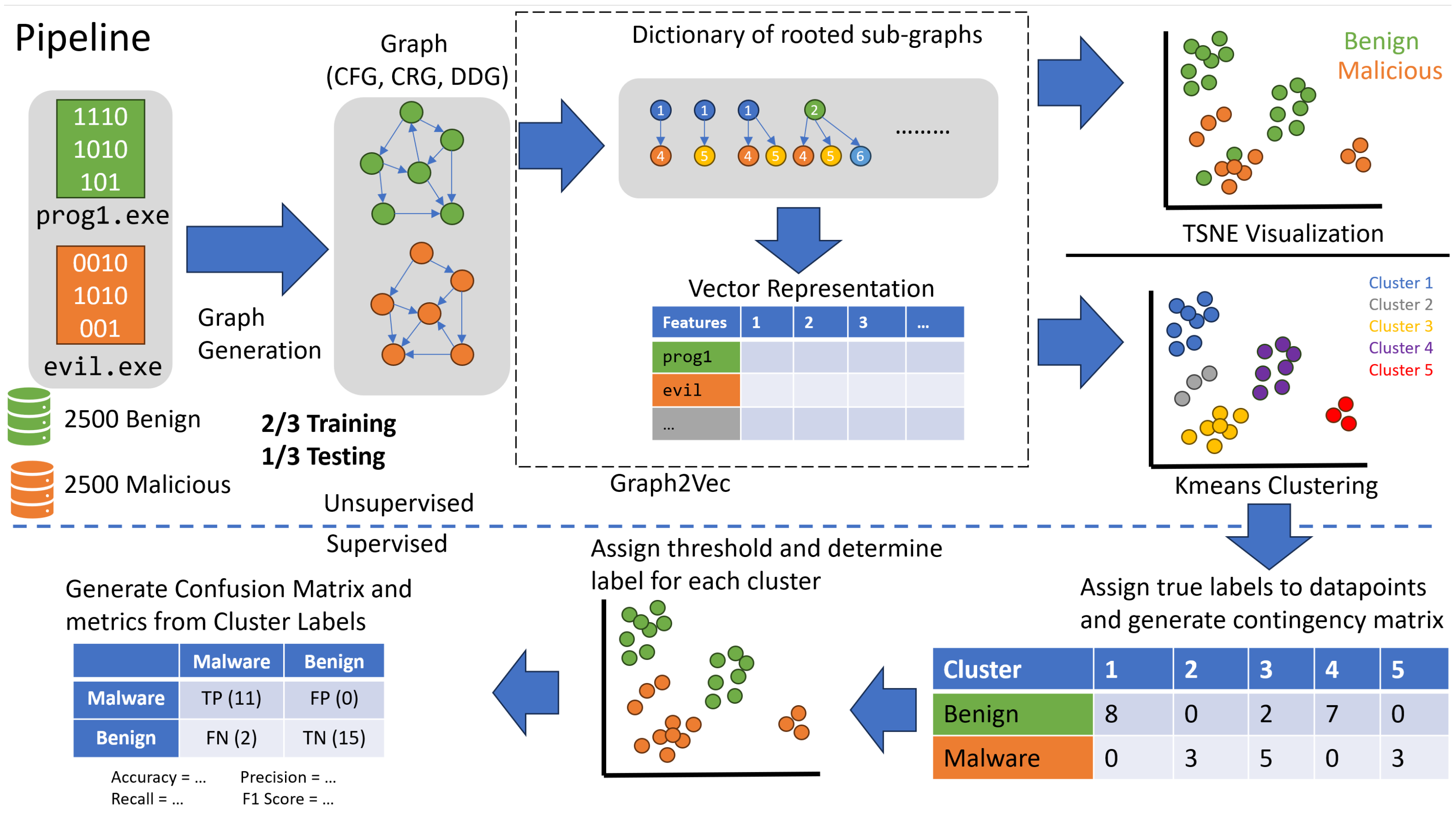
APPROACH

Research Question 1: Which graph representation of a binary programs yields the best results for malware *detection*?

Research Question 2: Using the graph representation from research question 1, how well does the clustering perform at *categorizing* the different types of malware?

Pipeline

A pipeline of the experiment can be seen below in *Figure 5.1.* Binaries are converted into different graph representations using *angr*, and then plugged into Graph2Vec to get their embeddings. Then, K-means clustering is applied and clusters are labeled either benign or malicious based on a threshold value. Then a confusion matrix is generated, and accuracy, recall, precision, and F1 score are calculated.



Features	1	2	3	...
prog1				
evil				
...				

Cluster	1	2	3	4	5
Benign	8	0	2	7	0
Malware	0	3	5	0	3

	Malware	Benign
Malware	TP (11)	FP (0)
Benign	FN (2)	TN (15)

Figure 5.1: Pipeline of Experiment Process

Dataset

The malicious binary programs were donated by a local cybersecurity company, Hoplite Industries [1]. This 14 terabyte dataset is a collection of malware from VirusTotal, honeypot servers, and other cybersecurity firms that have connections with Hoplite Industries. The malware consists of many types of files, such as portable executable, pdfs, java programs, and word documents, however, the scope of this research is binary programs, so only binary executable files (.exe files) were used in this experiment.

The benign binary programs consist of operating system files from both Windows systems and Linux systems. The file type of these programs are .exe files for the Windows files, and ELF executables for the Linux files.

Graph Generation

Graphs were generated using the Angr[2] module for Python. Angr is an open-source binary analysis platform that utilizes both static and dynamic analysis techniques. While Angr has the ability to do a variety of tasks, such as disassembly, decompilation, and symbolic execution, one of the more interesting capabilities angr has is the ability to generate several different types of graphs given a binary file.

Angr can create CFGs using two different methods. The first, CFGFast, uses basic static analysis and disassembly to generate the control flow of a program. Angr converts the binary to an intermediate representation called Vex and then constructs the set of nodes and edges from the Vex code. The other method, CFGEmulated, uses symbolic execution to generate the control flow. Angr can also generate a Data Dependency Graph using the built-in DDG function, and Function Call Graphs using its Function Manager object.

[1]https://hoplite.io
[2]https://angr.io/

Graph2Vec

Once graphs have been generated, a fixed-length vector needs to be generated using a graph embedding algorithm. Graph2Vec[30] is used to convert the graphs to an N-length vector. There are a handful of other graph embedding algorithms that can do the job, however, previous work done by Strandova-Neeley et al. [37] has shown that Graph2Vec to be a reliable graph embedding algorithm and has provided good results in the past. Graph2Vec is an unsupervised learning algorithm that learns embeddings based on rooted sub graphs around each node. An in-depth explanation of Graph2vec can be found in [30], [27], and [28]. The main idea behind Graph2Vec is to treat the graph as a sequence of subgraphs and use techniques from natural language processing to convert these subgraphs into fixed-size vector representations. The graph is first divided into a set of subgraphs. These subgraphs can be extracted using various techniques, such as random walks, node neighborhoods, or predefined patterns. Each subgraph represents a local structural pattern within the graph. Each subgraph is then treated as a sequence of nodes and edges. Just like words in a sentence, nodes and edges are treated as tokens in a sequence. These sequences of nodes and edges are then processed using techniques from natural language processing, such as skip-gram or continuous bad of words, which are commonly used for word embeddings. During the training process, Graph2Vec learns embeddings for each node by predicting the surrounding nodes or subgraphs within a certain context window. The goal in this step is to capture the contextual information of nodes within the graph. Once the node embeddings are learned, they are aggregated to obtain a graph-level embedding. This can be done using various aggregation functions, such as mean, sum, or pooling. The resulting graph-level embedding represents the entire graph in a lower-dimensional vector space. The learned graph embeddings can then be used as feature vectors for downstream machine learning tasks, such as classification.

t-SNE

t-distributed stochastic neighbor embedding, or t-SNE [39] visualizations are generated once Graph2vec is finished. t-SNE is a dimension reduction method to visualize high dimensional data in a low dimensional space. t-SNE calculates a probability distribution that represents similarities between neighbors, and then tries to generate a low-dimensional space with the same number of data points with a similar probability distribution. t-SNE is particularly effective at capturing the nonlinear structure and relationships within the data, allowing for the discovery of meaningful patterns and clusters.

Clustering

Once the n-length embeddings are generated from Graph2Vec, the embeddings can be mapped in a n-dimensional space, and placed into similar groups, or clusters. There are many different clustering algorithms that can be used, but **K-means** [19] will be the sole clustering algorithm that is used in the scope of this work. This is because K-means is widely popular and efficient clustering algorithm that can handle large datasets, and generally provided accurate and reliable results. The centroid-based algorithm works by minimizing the within-cluster sum of squares. K clusters are generated where K is generally an input the algorithm, and each point is placed inside the nearest cluster based on some distance metric.

Research Question 1: Cluster Labeling and Classification

Once clustering has been performed, a contingency matrix can be calculated. A contingency matrix will measure the number of malicious data points and benign data points in each cluster. Then, a threshold value, x is used to label each cluster as benign or malicious. If the percentage of malware in a cluster is greater than x, then the cluster is labeled as malicious, and each data point inside that cluster is classified as malicious. If the percentage of malware in a cluster is less than x, then the cluster is labeled as benign, and each data point inside that cluster is classified as benign.

Once each cluster has been labeled and all datapoints have a predicted label, the true labels are revealed and and a confusion matrix is generated. Once the confusion matrix reveals the number of true positives, true negatives, false positives, and false negatives, then accuracy, recall, precision, and F1 score are generated. The goal is to see which graph representation yielded the best F1 score.

Hyperparameters

There are three important hyperparameters that need to be considered during this experiment. The first is the number of dimensions for the embeddings, which is an input to Graph2Vec. The second is the number of clusters, which is an input to K-means. The last is the threshold percentage, which is used during cluster labeling. It is important to select the hyperparameter values that yield the best results, otherwise there is the possibility that a different value yields better results. Graph2Vec was run using 2, 4, 8, 16, 32, 64, 128, and 256 dimensions. K-means was run trying 1 cluster up to 100 clusters. A threshold value of 0 (all clusters are labeled as malicious) to 100 (most clusters are labeled as benign) were tried. This creates 120,000 different combinations, and the optimal combination (in terms of F1 score) needs to be used to ensure the best possible results. To deal with this, all possible combinations were tried for each graph type, and the combination that has the best F1 score was used.

Research Question 2: Cluster Labeling and Categorizing Malware

Using the graph representation that had the best F1 score from research question 1, the malicious datapoints are broken down into their different families. The malware families for the scope of this research are trojan, adware, worm, virus, and *other*. The *other* category consists of several smaller families of malware that did not have a high enough frequency to make its own class. Once again, the goal is to assign each cluster a label, and all data points within that cluster are predicted to be that label. A threshold can not be used in multi-class classification, because there is not a guarantee that a class will reach that threshold. For example, if the threshold was 33%, but a cluster has 20% trojans, 20% adware, 20% worm, 20% virus, and 20% benign, none of the classes reached the 33% threshold, so a label cannot be assigned to the cluster. Instead of a threshold, a simple majority-rules policy was assigned during cluster analysis.

A contingency matrix is calculated again, and the cluster was labeled to be the malware family that made up the majority of the cluster, or the majority of the points in the cluster were benign, the cluster was labeled as benign. Once this has been done, the true labels are revealed, and a confusion matrix is calculated. Then, the accuracy, recall, precision, and F1 score are calculated for each class.

RESULTS

Research Question 1

As discussed in the previous sections, the optimal hyperparameters for dimensions, number of clusters, and threshold percentage were used. The optimal hyperparameters for each graph type can be seen below in table 6.1

Table 6.1: Optimal Hyperparamaters used for each Graph Type

Graph Type	Number of Dimensions	Number of Clusters	Threshold Percentage
CFG Fast	64	84	43
CFG Emulated	64	98	34
Call Reference Graph	16	76	35
Data Dependency Graph	32	96	39

Figures 6.1, 6.2, 6.3, and 6.4 show t-SNE visualizations of the raw embeddings for each graph type. Embeddings are plotted in the optimal dimension from table 6.1, and then reduced to two dimensions using t-SNE. t-SNE is unsupervised, so the labels are not applied until after the dimensions have been reduced. Distinct groupings of similar data points are good to see, because making decisions during the cluster labeling step will be easier if there are clusters with all benign points or all malicious points. Each graph representation has some distinct clusters of malicious or benign data points, however CFG emulated seems to have the most distinct clusters of benign or malicious points.

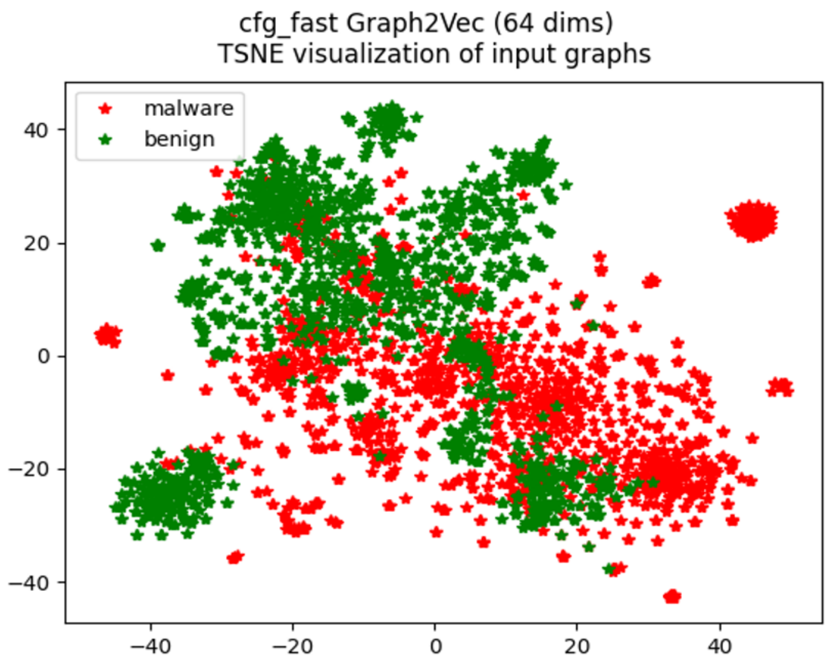


Figure 6.1: t-SNE Visualization of CFG Fast Embeddings

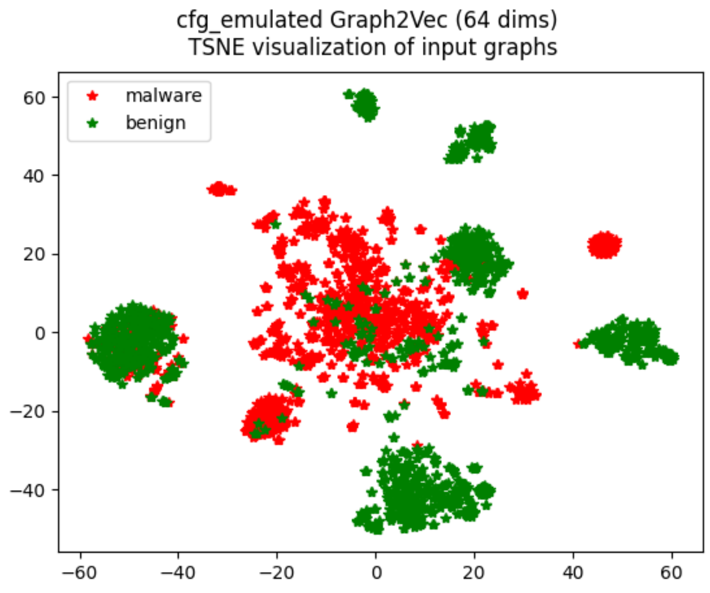


Figure 6.2: t-SNE Visualization of CFG Emulated Embeddings

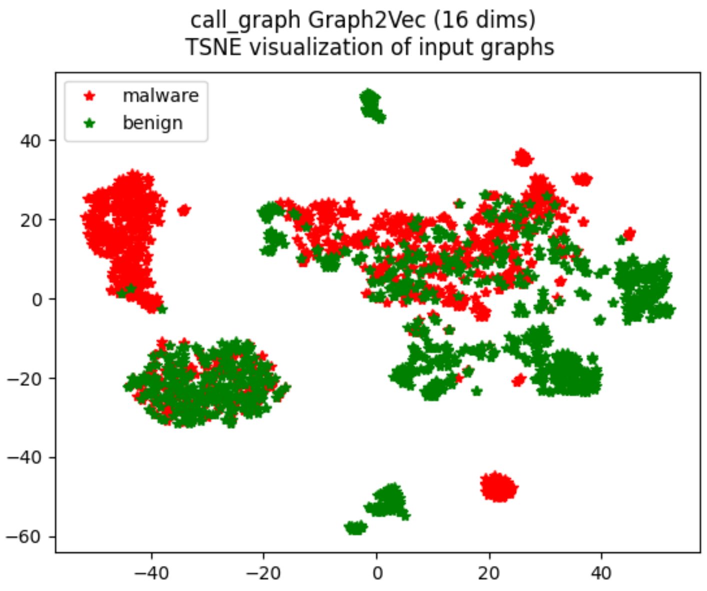


Figure 6.3: t-SNE Visualization of Call Graph Embeddings

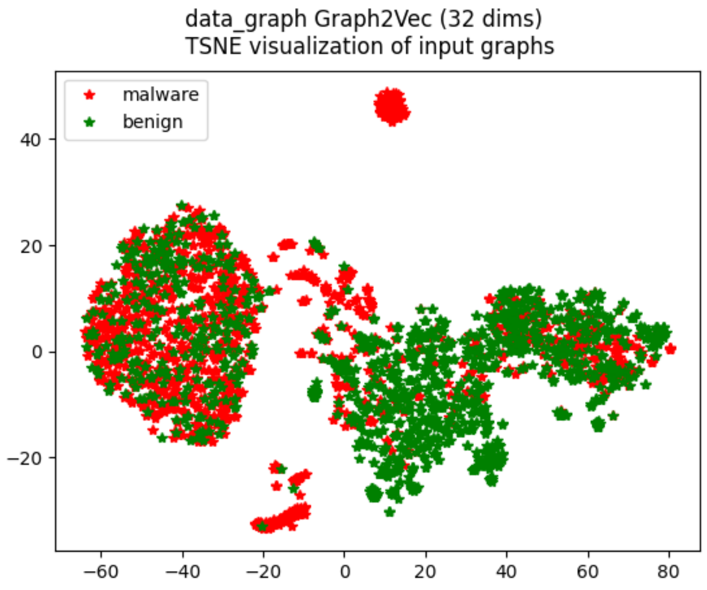


Figure 6.4: K-means clustering of Data Dependency Graph Embeddings

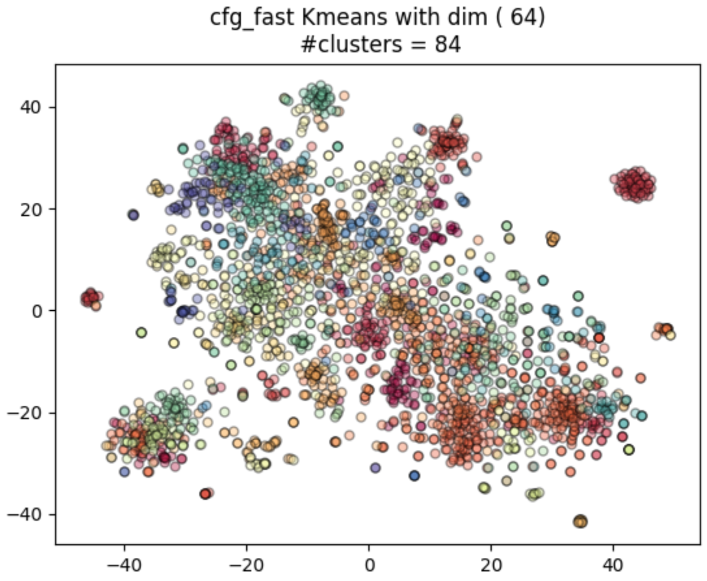


Figure 6.5: K-means clustering of CFG Fast Embeddings

Figures 6.5, 6.6, 6.7, and 6.8 are the K-means cluster for each graph representation. Colors represent the grouping decisions made by K-means. Close data points that are the same color are the groupings made by K-means. The number clusters provided to K-means was pulled from table 6.1. Clustering happens in a higher number of dimensions, and then the embeddings are reduced to two dimensions for visualization.

Tables 6.2, 6.3, 6.4, and 6.5 are the generated confusion matrices after cluster labeling and predictions have been applied. The columns represent the decision the classifier made, and the rows represent the true value of a data point. The top left number represents the number of true positives. The top right number represents the number of false positives. The bottom left number represent the number of false negatives. The bottom right number represent the number of true negatives. The scope of this work is more focused on correctly identifying malware, so we are looking for a high number of true positives, and a low number of false positives. CFG Emulated had the highest amount of true positives while also having the lowest number of false positives. Data dependency graph has had the lowest number of

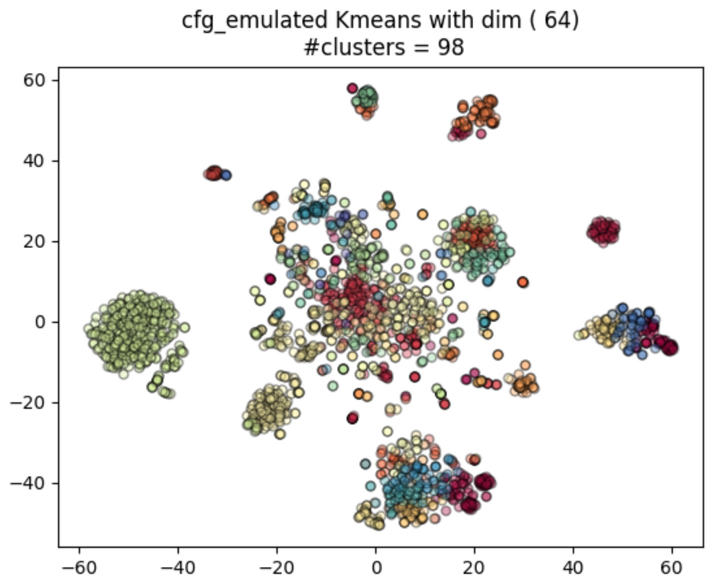


Figure 6.6: K-means clustering of CFG Emulated Embeddings

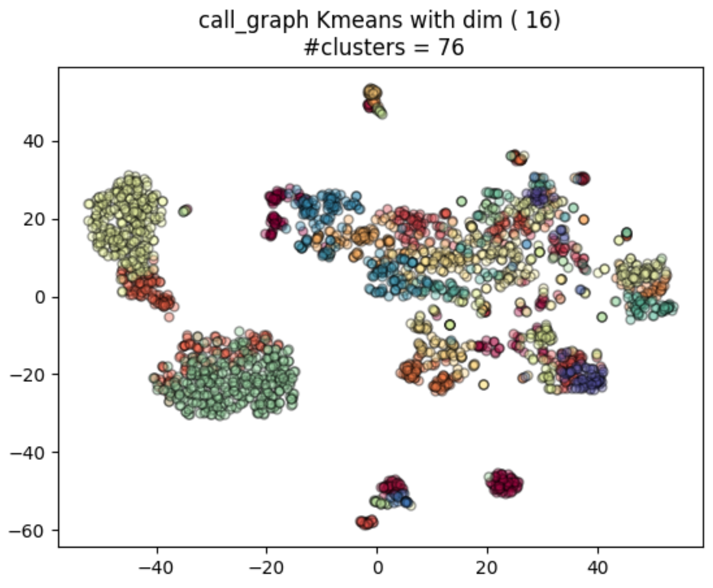


Figure 6.7: K-means clustering of Call Graph Embeddings

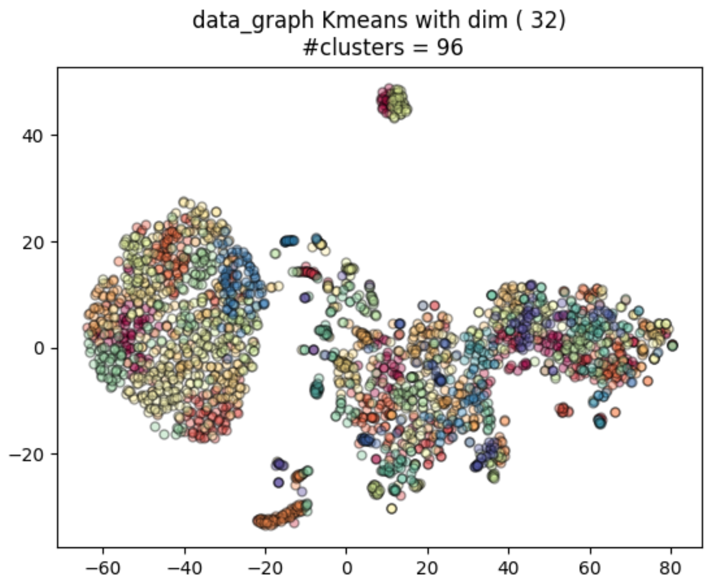


Figure 6.8: K-means clustering of Data Dependency Graph Embeddings

true positives and the highest number of false positives.

Table 6.2: Confusion Matrix for CFG Fast Graph Classification

		Predicted Values	
		Malware	Benign
Actual Values	Malware	683	69
	Benign	105	743

Table 6.3: Confusion Matrix for CFG Emulated Graph Classification

		Predicted Values	
		Malware	Benign
Actual Values	Malware	714	48
	Benign	74	764

With a confusion matrix for each classifier, accuracy, recall, precision, and F1 score are calculated for each graph type. These metrics are recorded below in table 6.6. CFG Emulated had the highest accuracy, precision, recall, and F1 score.

Figure 6.9 shows how the F1 score changed while holding the optimal dimensions and threshold value, but changing the number of clusters for each graph type. Figure 6.10 shows how F1 score changed while holding the optimal dimensions and number of clusters, but changing the threshold value for each graph type.

Table 6.4: Confusion Matrix for Call Reference Graph Classification

		Predicted Values	
		Malware	Benign
Actual Values	Malware	690	186
	Benign	98	626

Table 6.5: Confusion Matrix for Data Dependency Graph Classification

		Predicted Values	
		Malware	Benign
Actual Values	Malware	698	291
	Benign	90	521

Research Question 2

Labels for the malware family are applied to each data point. The frequencies of each malware family is listed below in table 6.3. The dataset is highly imbalanced, with trojan malware making up roughly 75% of the dataset. According the the experiment done in the previous research question, CFG Emulated had the highest F1 score, therefore CFG Emulated will be the only graph representation that is used for research question 2.

Figure 6.15 shows t-SNE visualization of the testing data with the malware family labels applied. Figure 6.16 shows the same visualization, but with benign data points excluded. With benign points being excluded, it is more difficult to see distinct clusters of similar malware types.

Table 6.6: Classification Metrics for Each Graph Type

Graph Type	Accuracy	Recall	Precision	F1 Score
CFG Fast	0.89125	0.866751	0.908245	0.887013
CFG Emulated	0.92375	0.906091	0.937008	**0.92129**
Call Graph	0.8225	0.875635	0.787671	0.829327
Data Dependency Graph	0.761875	0.885787	0.705763	0.785594

Table 6.7: Frequencies of Malware Type in Testing Data

Family	Count
Trojan	1871 (74.8%)
Adware	163 (6.5%)
Virus	159 (6.4%)
Worm	180 (7.2%)
Other	128 (5.1%)

Table 6.8 shows the confusion matrix for classifying the different malware families, and figure 6.18 shows the individual confusion matrix for each malware family. Trojan had the highest number of true positives, but also the highest number of false positives. This is due to the imbalance of classes in the dataset.

Table 6.9 shows the the calculated metrics (accuracy, recall, precision, and F1 score) for each of the malware families. Classifying trojan malware had the highest F1 score of .8000. This table also shows the downsides of using accuracy as a metric when using an imbalanced dataset. Adware, virus, worm, and other all had very low F1 scores, but had high accuracy values. The high accuracy values are simply due to the fact that adware, virus, worm, and other were a small minority and made up a small amount of the overall dataset. In some

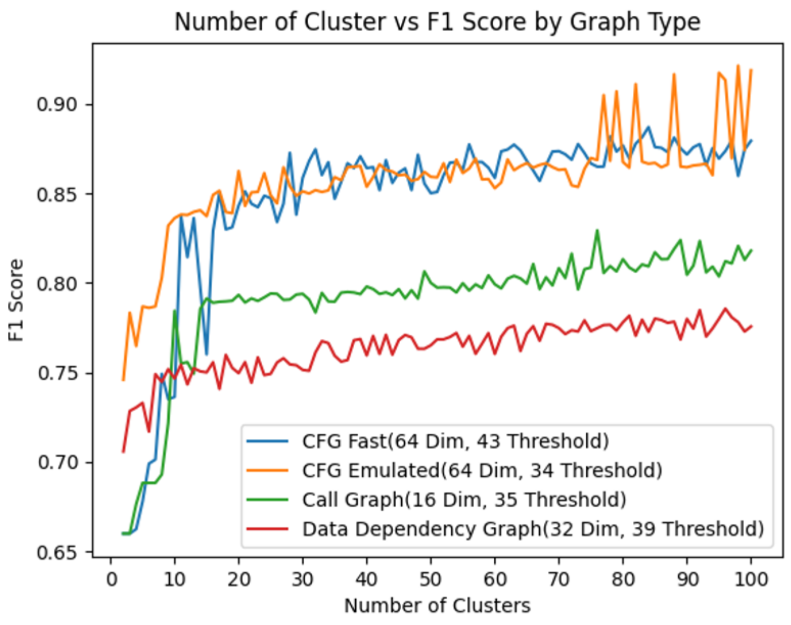


Figure 6.9: F1 Score vs Number of Clusters for each Graph Type

cases, some metrics were unable to be calculated because there were no true positives.

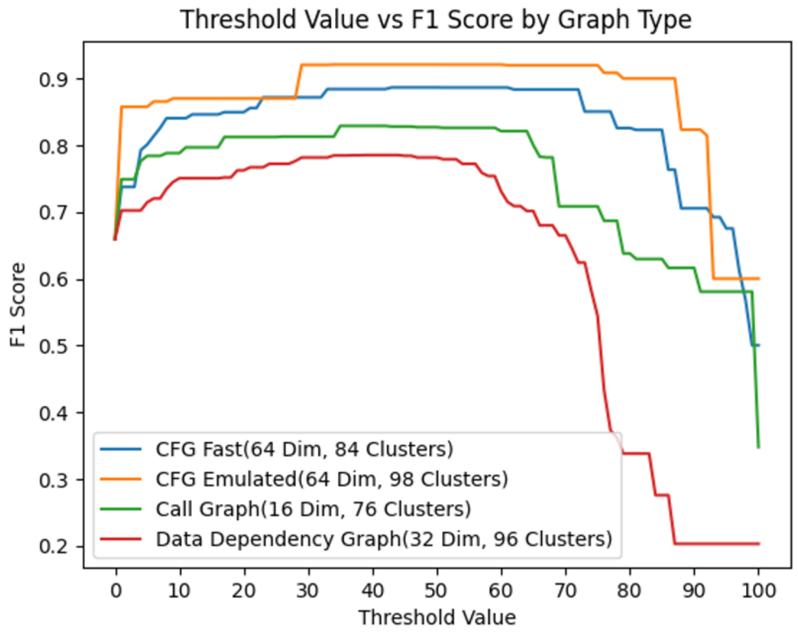


Figure 6.10: F1 Score vs Threshold Value for each Graph Type

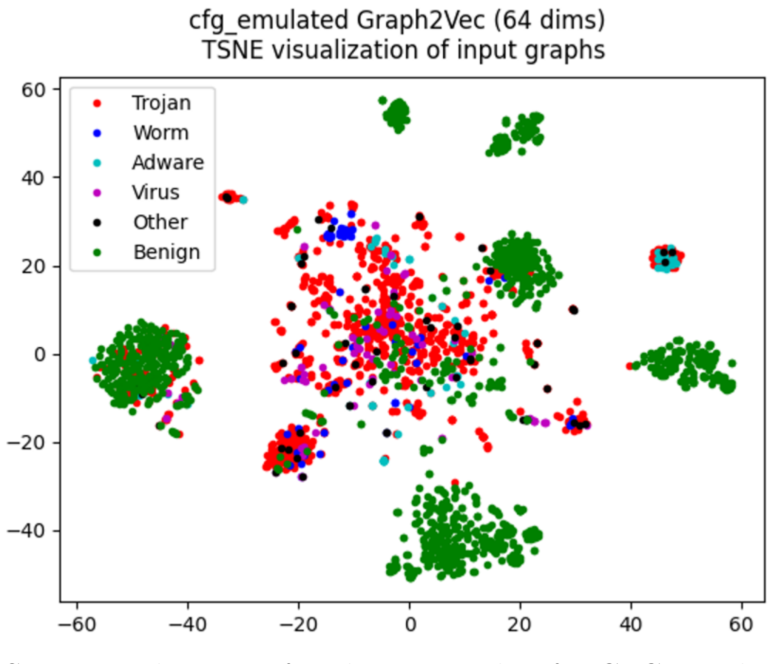


Figure 6.11: t-SNE Visualization of Malware Families for CFG Emulated Embeddings

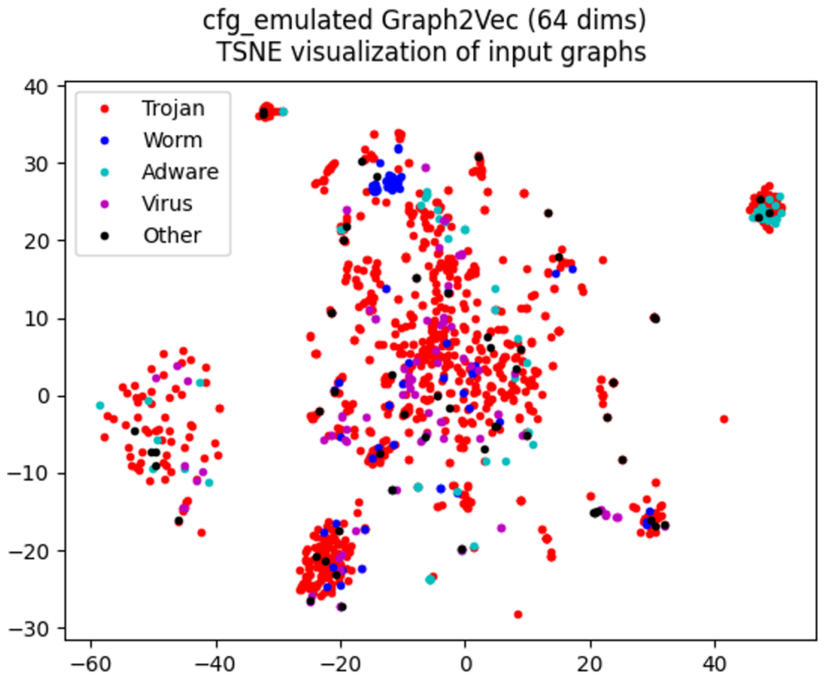
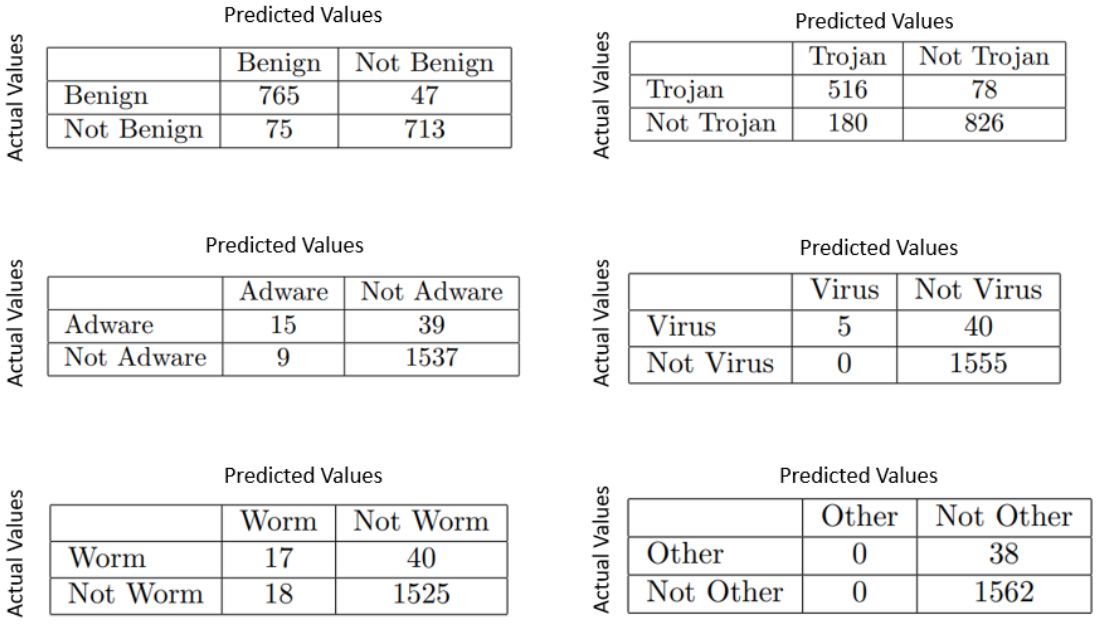


Figure 6.12: t-SNE Visualization of Malware Families for CFG Emulated Embeddings (No benign)

Table 6.8: Confusion Matrix for Malware Type

		Predicted Values					
	Malware Type	Benign	Trojan	Adware	Virus	Worm	Other
	Benign	765	47	0	0	0	0
	Trojan	57	516	5	0	16	0
Actual Values	Adware	11	28	15	0	0	0
	Virus	1	35	3	5	1	0
	Worm	2	38	0	0	17	0
	Other	4	32	1	0	1	0



Predicted Values

Actual Values

	Benign	Not Benign
Benign	765	47
Not Benign	75	713

Predicted Values

Actual Values

	Trojan	Not Trojan
Trojan	516	78
Not Trojan	180	826

Predicted Values

Actual Values

	Adware	Not Adware
Adware	15	39
Not Adware	9	1537

Predicted Values

Actual Values

	Virus	Not Virus
Virus	5	40
Not Virus	0	1555

Predicted Values

Actual Values

	Worm	Not Worm
Worm	17	40
Not Worm	18	1525

Predicted Values

Actual Values

	Other	Not Other
Other	0	38
Not Other	0	1562

Figure 6.13: Individual Confusion Matrices for each Malware Family

Table 6.9: Classification Metrics for each malware family

Malware Type	Accuracy	Precision	Recall	F1 Score
Benign	0.9238	0.9107	0.9421	0.9261
Trojan	0.8388	0.7414	0.8687	0.8000
Adware	0.9700	0.625	0.2778	0.3846
Virus	0.975	1	0.1111	0.2
Worm	0.9638	0.29825	0.4857	0.3696
Other	0.9763	NaN	0	NaN

DISCUSSION

Research Question 1: Which graph representation of a binary programs yields the best results for malware *detection*?

Research Question 2: Using the graph representation from research question 1, how well does the clustering perform at *categorizing* the different types of malware?

Research Question 1

Table 6.6 shows the important classification metrics for each of the graph representations. CFG Emulated had the highest F1 score of all four graph types with an F1 score of 0.92129. CFG Fast was close behind it, with an F1 score of 0.887013. Function call graph and data dependency graph were both had a significantly lower F1 score, which function call graph taking third place with an F1 score of 0.829327, and data dependency graph being the worst of the four with an F1 score of 0.785594. Table 6.3 also shows that CFG Emulated had the lowest amount of false positives, which is important to minimize when failing to flag malware can have dangerous consequences. These results show that CFG Emulated seems to be the best graph representation for detecting malware.

The fact that CFG Emulated performed the best shows the benefits and future potential for symbolic execution. CFG Emulated may have performed better than CFG Fast due to some of the downsides of standard disassembly during CFG generation. Junk code and unreachable code are still processed and become part of the CFG graph, which creates extra nodes, edges, and sub graphs. Symbolic execution attempts to simulate execution of a program, so constructs such as unreachable code, or junk code will not be captured in the CFG. This leads to a more accurate graph representation, and possibly a more accurate learning process during the Graph2Vec step. CFG emulated likely performed better than the call reference graph due to the fact that CFG emulated captures both the control flow between functions, as well as the control flow within a function. CFG emulated is a more detailed call reference graph, so this may have lead to a more accurate learning process during the Graph2Vec step. A data dependency graph does not capture a control flow, and due to this seems to perform much worse than the other the other graph representations when it comes to malware classification.

Research Question 2

Table 6.9 shows the important classification metrics for each malware type, and figure 6.18 shows the individual confusion matrices for each malware type. Trojan had an F1 score of 0.8, which was the highest across all other malware types. Adware, virus, worm, and other had higher accuracies, however this is misleading due the testing data being significantly unbalanced. Table 6.7 shows the breakdown of the frequencies of the five different classes. A single class, trojan makes up about 75% of the testing data. Because a simple majority-rules was applied during the cluster labeling step, the trojan class will frequently be the majority in most of the clusters, therefore the smaller classes, such as worm, virus, and adware, will rarely be flagged correctly, which effects metrics such accuracy, recall, and precision.

The classifier was able to correctly categorize Trojan the must, but this is most likely due to the unbalanced dataset. If a more balanced dataset were to be used, different results may be found. The classifier was not able to correctly categorize adware, virus, worm, and the other category, but this is once again likely due to the unbalanced dataset. Therefore, it is difficult to say for sure how effective CFG emulated was at categorzing malware.

CONCLUSION

Threats to Validity

Dataset

A large hurdle of this work is the fact that the dataset was unbalanced when it comes to the different malware families. Because the dataset was massively dominated by trojans, it is difficult to draw meaningful outcomes from the experiment. Additionally, there may be some sampling bias. The dataset that was used is not be representative of all different kinds of malware. A challenging part of this work was finding executables that were able to generate the four graph types. There were many instances where angr would error out while generating graphs, so the executable had to be thrown out. Because of the difficulties of using angr to generate graphs, the focus was finding a balanced dataset of benign and malicious executables, but not a balanced dataset for malware types. This work did not evaluate other forms of malware, such as word documents, mobile malware, and PDFs, but this is because we focused only on binaries. The scope of this research did not touch obfuscated malware, which may reveal shortcomings in this classification method.

Graph Representations

This work only focused on four different graph representations that were all generated using the same binary analysis tool. These were the only graphs that were studied because these were the only four that angr could generate. There are more graph representations, such as Interprocedural Control Flow Graphs [34] and other forms of dynamic graphs [18]. Angr was the only binary analysis tool that was also used. There exists many other binary analysis tools that can generate graphs, such as *Ghidra* [40] and *Binary Analysis Platform* [9]. Some of these tools may be able to generate more accurate graphs, or also generate more types of graphs.

Clustering and Embedding Algorithms

K-means and Graph2vec were the only clustering and graph embedding algorithms used, respectively. There are many other clustering algorithms, and using a different clustering algorithm [44] may yield better results. The same goes for graph embedding. There may be a different graph embedding algorithm[16] that provides better results

Future work

Evaluating more graph representations, graph generation tools, clustering algorithms, and graph embeddings algorithms to see if results can improve. Additionally, this pipeline should be translated to work on much larger datasets, and possibly implemented into some kind of useable endpoint tool to scan or detect malware. Additionally, this work did not test on obfuscated malware. Because obfuscated malware is commonly seen in the wild, the pipeline should be tested on malware that deploys a variety of obfuscation techniques. Lastly, more work

Conclusion

This thesis provided a comparison between different graph representations of programs to see which one provided the best results for both malware detection and malware categorization. This work also proposed a pipeline for generating graphs, computing their embeddings, and creating a classifier for malware detection and categorization based on their embeddings.

We began by providing relevant background information about malware detection methods and defining different ways to represent a program with a graph. Next, we gave an overview about other relevant work in the area. While it has been shown that different graph representations of programs can be combined with machine learning to detect malware, a comparison between the different representations of programs has not been done before, which was the main motivating factor for my research goals, which were driven by the goal-question-metric (GQM) model [4].

We then described an approach for comparing the different graph representations. Once graphs were generated, their embeddings were gathered using the Graph2Vec graph embedding algorithm, and their embeddings were then plotted and clustered using K-means. Then, clusters were labeled and points within the clusters were given the identical label as the cluster, and a classifier was built.

The results of this research showed the control flow graphs that were generated using symbolic execution did the best at detecting malware. For the other research question, the classifier was able to most accurately categorize trojan malware, however more work needs to be done, and a more balanced dataset needs to be used.

As malware attacks continue in the cyber world, there is a dire need for the ability to detect and categorize malware before damage can be caused. Machine learning continues to show promise in fighting the battle against malware. Graphs, particularly control flow graphs are one of many graph representations that can fed to machine learning algorithms to improve machine learning's ability to detect and categorize malware.

REFERENCES CITED

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 41–42 vol.2, 2004.

[2] Asad Afreen, Moosa Aslam, and Saad Ahmed. Analysis of fileless malware and its evasive behavior. In *2020 International Conference on Cyber Warfare and Security (ICCWS)*, pages 1–8, 2020.

[3] Quist D. Neil J. Storlie C. Lane T. Anderson, B. : Graph-based malware detection using dynamic analysis. *J. Comput. Virol*, 7(4), 2011.

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. 1994.

[5] Zahra Bazrafshan, Hashem Hashemi, Seyed Mehdi Hazrati Fard, and Ali Hamzeh. A survey on heuristic malware detection techniques. pages 113–120, 05 2013.

[6] Daniel Bilar. Opcodes as predictor for malware. *Int. J. Electron. Secur. Digit. Forensic*, 1(2):156–168, jan 2007.

[7] Tristan Bilot, Nour Madhoun, Khaldoun Agha, and Anis Zouaoui. A survey on malware detection with graph representation learning, 03 2023.

[8] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, may 1966.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 463–469. Springer, 2011.

[10] Nicholas Duran, Jurijs Girtakovskis, Ken Jacobi, David Kennerley, Justine Kurtz, Grayson Milbourne, Tyler Moffitt, Cameron Palan, and Steve Snyder. 2018.

[11] Jeffrey Fairbanks, Andres Orbe, Christine Patterson, Janet Layne, Edoardo Serra, and Marion Scheepers. Identifying attck tactics in android malware control flow graph through graph representation learning and interpretability. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5602–5608, 2021.

[12] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. Graph embedding based familial analysis of android malware using unsupervised learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 771–782, 2019.

[13] Pengbin Feng, Jianfeng Ma, Teng Li, Xindi Ma, Ning Xi, and Di Lu. Android malware detection based on call graph via graph neural network. In *2020 International Conference on Networking and Network Applications (NaNA)*, pages 368–374, 2020.

[14] Hossein Gharaee, Shokoufeh Seifi, and Nima Monsefan. A survey of pattern matching algorithm in intrusion detection system. pages 946–953, 09 2014.

[15] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151, 05 2017.

[16] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[17] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. Automatic generation of string signatures for malware detection. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, pages 101–120, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[18] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[19] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[20] Hashem Hashemi et al. Graph embedding as a new approach for unknown malware detection. *Journal of Computer Virology and Hacking Techniques 13. 3*, 3, 2017.

[21] Kazuki Iwamoto and Katsumi Wasaki. Malware classification based on extracted api sequences using static analysis. In *Proceedings of the 8th Asian Internet Engineering Conference*, AINTEC '12, page 31–38, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Haodi Jiang, Turki Turki, and Jason T. L. Wang. Dlgraph: Malware detection using deep learning and graph embedding. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1029–1033, 2018.

[23] Teenu S. John, Tony Thomas, and Sabu Emmanuel. Graph convolutional networks for android malware detection with system call graphs. In *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, pages 162–170, 2020.

[24] Teenu S. John, Tony Thomas, and Sabu Emmanuel. Graph convolutional networks for android malware detection with system call graphs. In *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, pages 162–170, 2020.

[25] Xufang Li, Peter KK Loh, and Freddy Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European intelligence and security informatics conference*, pages 149–154. IEEE, 2011.

[26] Chen Liu, Bo Li, Jun Zhao, Ziyang Zhen, Xudong Liu, and Qunshi Zhang. Fewm-hgcl : few-shot malware variants detection via heterogeneous graph contrastive learning. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2022.

[27] Kaveen Liyanage, Reese Pearsall, Clemente Izurieta, and Bradley M. Whitaker. Dictionary learning on graph data with weisfieler-lehman sub-tree kernel and ksvd. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, June 2023.

[28] Kaveen Liyanage, Reese Pearsall, Bradley Whitaker, and Clem Izurieta. Malware detection using unsupervised clustering of binary file control flow graphs. 2023.

[29] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. Graph neural network-based android malware classification with jumping knowledge. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–9, 2022.

[30] Annamalai Narayanan, Chandramohan Mahinthan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. 07 2017.

[31] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), sep 2019.

[32] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851, 2021.

[33] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851, 2021.

[34] Matt Revelle, Matt Parker, and Kevin Orr. Blaze: A framework for interprocedural binary analysis. 01 2023.

[35] Neha Runwal, Richard Low, and Mark Stamp. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, 8, 05 2012.

[36] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979.

[37] Veronika Strandova-Neeley, Daniel Laden, Reese Pearsall, David Optiz, Andrew Rippy, and Shayla Sharma. Graph-based analysis of binary code for malware detection and vulnerability identification. *Cyber QR ops report*, 2021.

[38] Drashti Vadaviya and Purvi Tandel. Study of avalanche effect in aes. 05 2015.

[39] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[40] Daniel Votipka, Mary Nicole Punzalan, Seth M. Rabin, Yla Tausczik, and Michelle L. Mazurek. An investigation of online reverse engineering community discussions in the context of ghidra. In *2021 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 1–20, 2021.

[41] Fok Kar Wai and Vrizlynn L. L. Thing. Clustering based opcode graph generation for malware variant detection. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pages 1–11, 2021.

[42] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep*, pages 14–23, 2009.

[43] Peng Xu. Android-coco: Android malware detection with graph neural network for byte- and native-code, 2022.

[44] Rui Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[45] Jiaqi Yan, Guanhua Yan, and Dong Jin. 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)*, pages 52–63, 2019.

[46] Yanfang Ye, Lifei Chen, Dingding Wang, Tao Li, Qingshan Jiang, and Min Zhao. Sbmds: An interpretable string based malware detection system using svm ensemble with bagging. *Journal in Computer Virology*, 5:283–293, 10 2009.

[47] Katsunari Yoshioka, Yoshihiko Hosobuchi, Tatsunori Orii, and Tsutomu Matsumoto. Vulnerability in public malware sandbox analysis systems. pages 265–268, 07 2010.