Complexity-Based Graph Attention Network for Metamorphic Malware Detection

by

Kenneth Brezinski

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfilment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Electrical and Computer Engineering
University of Manitoba
Winnipeg

Complexity-Based Graph Attention Network for Metamorphic Malware Detection

Kenneth Brezinski
DOCTOR OF PHILOSOPHY

Electrical and Computer Engineering
University of Manitoba
2024

Abstract

This thesis work presents a new approach to malware analysis by creating a specialized sandbox environment for executing and monitoring malware on a host operating system. Over 200 malware samples, along with benignware, were tested in this environment. Application Programming Interface (API) calls were traced of how these executable interacted with the host system, which including registry changes, file system access, and thread activity. Two new methods to measure complexity, Mass Radius and Radius of Gyration Fractal Dimension (FD), were developed and added to a Deep Learning model. These complexity methods helped the model converge faster. Tests showed that the Mass Radius FD measure improved convergence and accuracy over the Radius of Gyration FD in identifying malware. The complexity models performed better than standard models across different datasets. The study also found that shorter sequences of API calls and file events were more likely to indicate malicious behavior. Using GNNExplainer, linked API sequences were linked to specific malware techniques, providing deeper insights into the model's predictions. The complexity models identified flaws in traditional methods and successfully flagged malware that other commercial sandbox methods were not able to identify, lending credence to the sophistication and applicability of this work.

Contents

List of Tables

List of Figures

List of Abbreviations

ARP	**A**ddress **R**esolution **P**rotocol
APC	**A**synchronous **P**rocedure **C**all
API	**A**pplication **P**rogramming **I**nterface
APP	**A**ddress **R**esolution **P**rotocol
AUC	**A**rea **U**nder the **C**urve
AV	**A**nti-**V**irus
BoW	**B**ag-**o**f-**W**ords
CFG	**C**ontrol **F**low **G**raph
CnC	**C**ommand a**n**d **C**ontrol
CNN	**C**onvolutional **N**eural **N**etwork
DL	**D**eep **L**earning
DLL	**D**ynamically **L**inked **L**ibrary
DNS	**D**omain **N**ame **S**erver
DOS	**D**enial **O**f **S**ervice
EAT	**E**xport **A**ddress **T**able
ELF	**E**xecutable and **L**inkable **F**ormat
EXE	**EXE**cutable
FD	**F**ractal **D**imension
FP	**F**alse **P**ositive
FTP	**F**ile **T**ransfer **P**rotocol
GAT	**G**raph **AT**tention Network
GUI	**G**raphical **U**ser **I**nterface
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HTTPS	**H**yper**T**ext **T**ransfer **P**rotocol **S**ecure
IAT	**I**mport **A**ddress **T**able
ICMP	**I**nternet **C**ontrol **M**essage **P**rotocol
IDS	**I**ntrusion **D**etection **S**ystem
IDT	**I**nterrupt **D**escriptor **T**able
I/O	**I**nput/**O**utput
IP	**I**nternet **P**rotocol
IT	**I**nternet **T**echnology
k-NN	**k**-**N**earest **N**eighbors
LAN	**L**ocal **A**rea **N**etwork

LCS	**L**ongest **C**ommon **S**ubsequence
LSTM	**L**ong **S**hort-**T**erm **M**emory
ML	**M**achine **L**earning
MR	**M**ass **R**adius
NAT	**N**etwork **A**ddress **T**ranslation
NGVCK	**N**ext **G**eneration **V**irus **C**reation **K**it
NLP	**N**atural **L**anguage **P**rocessing
OS	**O**perating **S**ystem
PE	**P**ortable **E**xecutable
PCA	**P**rinciple **C**omponent **A**nalysis
PSO	**P**article **S**warm **O**ptimization
RAM	**R**andom **A**ccess **M**emory
RMS	**R**oot **M**ean **S**quared
RMSE	**R**oot **M**ean **S**quared **E**rror
RNN	**R**ecurrent **N**eural **N**etwork
RoG	**R**adius **o**f **G**yration
rpm	**r**otations **p**er **m**inute
SA	**S**imulated **A**nnealing
SP	**S**ervice **P**ack
SSDT	**S**ystem **S**ervice **D**escriptor **T**able
SSH	**S**ecure **SH**ell Protocol
SVM	**S**upport **V**ector **M**achine
TCP	**T**ransmission **C**ontrol **P**rotocol
tf-idf	**t**erm **f**requency-**i**nverse **d**ocument **f**requency
TP	**T**rue **P**ositive
UDP	**U**ser **D**atagram **P**rotocol
URL	**U**niform **R**esource **L**ocator
USD	**U**nited **S**tates **D**ollars
VM	**V**irtual **M**achine
XGAN	Comple**X**ity-based **G**raph **A**ttention **N**etwork

List of Symbols

M	set of malware samples
C	set of benignware samples
c	subset of benignware samples
i	iteration / trial
C_s	number of classes
D	dataset
D_{val}	validation set
G	graph
$\mathcal{V}$	set of graph vertices
$\mathcal{E}$	set of graph edges
$\mathcal{N}$	set of neighbor nodes
$\mathcal{D}$	set of training examples
$\mathbf{h}$	node feature matrix
$\bar{h}$	node feature vector
$\bar{h}'$	final node feature embedding
m	number of features
α	attention coefficient
$\mathbf{W}$	weight matrix
$\bar{w}$	weight vector
w	weight
E	embedding dimension
$E[\cdot]$	expectation
σ	standard deviation / sigmoid function
N	number of processes
d	dimension
δ	time interval
ϵ	small value (e.g. 10^{-14})
n_H	attention heads
K	number of partitions
k	partition
O_c	centroid
r_k	radii from centroid
p	probability

P	order of the norm
d	distance
A	informational complexity
D_M	fractal dimension
z	net output
a	activation output
$\mathcal{L}$	model loss
l	number of model layers
n	number of processes
T	number of events types
μ	mean

Chapter 1

Introduction

This section will briefly introduce the proposed development of a Complexity-based Graph Attention Network (XGAN), as well as the motivation behind representing Malware execution as graphs. Additionally, the reasoning behind focusing on malicious executables solely on the Windows Operating System (OS) will be discussed. The discussion will begin with several *Research Questions*, followed by the *Artifacts* made available as a part of this research work, encompassing the entirety of the datasets and source code created as a derivative of this work.

1.1 Research Questions

The limitation of state-of-the-art techniques for Malware analysis is that there is still a dependency on Static Analysis due to its simplicity and the wide availability of tools already available for quick feature extraction on binaries. It is also the case that in the research space many existing techniques operate on a small subset of Malware samples that are readily available in literature [1], which have since become deprecated due to their age and are no longer relevant in today's threat landscape and do not represent current threats. While virtualized sandbox environments, such as Cuckoo, do allow some flexibility in disassembling malicious process execution, it is limited in many ways on the telemetry that is available to the user. Establishing a baseline between malicious and benign behaviour is at the core of any classification problem. Not only must the features under consideration be relevant, but they must be salient such that a model can distinguish malicious behaviour from any other behaviour. The features must also be readily available, and preprocessing must be computable in linear time such that a model can be implemented in production at an endpoint. Additionally, the models we develop today must be highly accurate by having low False Positives (FP) and high True Positives (TP) rates through being highly generalizable to unforeseen threats. While this is a tall task, with new innovations in Deep Learning (DL) we can rapidly prototype models on a wide variety of input types spanning from malicious binaries [2] to usage patterns in Windows APIs [3].

Based on the limitations of current approaches, this work hopes to solve them by addressing them all at once. This work introduces a methodology that combines the sample collection, preprocessing, and model development stages to create a framework for Malware classification similar to Cuckoo; but which can be used to train a large GNN model on dynamic process behaviour. Representative samples of new and emerging threats are executed in a controlled sandbox environment, which

are used to mimic a threat exposed on a host operating system. The process behaviour is then investigated by examining the API usage - one of the most discriminative feature types available as it describes exactly what the process is doing. This information is then mapped as a feature vector as a node in a graphical framework based on the process execution hierarchy. This work also lays the groundwork for a GNN that can examine the dynamic behaviour of Malware as it spawns daughter processes and makes changes to the Registry, creates and deletes files in the File System, as well as opens Files and Threads. The temporal sequence of spawn behaviour is important to consider as it is not necessarily a single process that is indicative of malicious behaviour, but rather, a sequence of processes executed sequentially that provides an indication of malicious intent. GNNs can account for this by creating a topology of the spawn instances on the OS at any point in time. This proposal also looks at two innovations in the application of GNNs: the use of attention coefficients in graphs to better learn node neighbour embeddings (defined as Graph Attention Networks (GATs) in the literature [4]) as well as a novel Complexity approach which looks at the informational complexity of the topology at multiple scales simultaneously to share information about the network as a whole. Combining these innovations, the proposed architecture is coined a Complexity-based Graph Attention Network, or XGAN. The research questions covered in this work are summarized as follows:

1. What are the appropriate set of features to consider when analysing the spawn behaviour of malicious processes? How do we best construct our feature vector space when examining the API call usage from Registry, File System, and Process and Thread activity? Are all these event types relevant in detecting malicious behaviour?
   Summary – What do we tell the model to learn and how do we allow it to learn?

2. Does the GAN architecture facilitate the training of malicious topologies?
   Summary – Is the model an appropriate tool to learn?

3. Is it possible to integrate Complexity into the GAN architecture, and how well does it improve the performance of the model?
   Summary – Can we develop new things for the model to learn from?

4. How well does the model learn from the provided training and validation data, and can it maximize False Positives (FP) and minimize False Negatives (FN) simultaneously? How well does the F1-score, a combined measure of Recall and Precision, improve when being trained on malicious topologies?
   Summary – How well does the model learn?

5. Can the model train on enough topologies so that it can **generalize** to never-before-seen Malware threats of varying process topologies? How does a larger proportion of Windows OS behaviour or benign processes affect the discrimination of a malicious topology from a benign one?
   Summary – How well does the model perform on new examples?

1.2 Research Artifacts

Several research artifacts have been produced as a part of the research work towards completion of this thesis. An extended overview of these contributions can be found in Appendix A.1. These also

include code repositories that are incidental to the thesis but related to my research contributions.

- The series of works in Combinatorial Optimization and Hybrid methods published in [2, 5, 6] can be found in my Hybrid-PSO-SA Github repository.
- A series of published works which describes the methodology of the Complexity Lambda layer and the Complexity Convolutional Neural Network described in [7] and [8] are found in the Complexity repository.
- The source code which provides the backend functionality for the *Sandy* sandbox for tracking Malware behavior is found in the Malware-Sandbox Github repository with an overview published in [9].
- The code implementation for the first implementation of a Graph Attention Network for classifying malicious stack traces published in [3] is found in the Malware-Transformer Github repository.
- The Complexity Layer published in [10] can be found in the GAT-Malware Github repository. The work containing Graph Networks trained on different event types from [11] and the work found in this thesis is also found in the same repository.
- All model binaries, checkpoints and results are also saved in the GAT-Malware repository. See Section C.3 in Appendix C for the full description of how to retrieve these files and binaries and view them on your local web server.

1.3 Research Motivation

Digital resources and infrastructure have become some of the most crucial concerns in the field of Cyber Security. As we encourage a greater use of the internet to delegate the tasks of everyday life, we expose ourselves and our information through potential exploitation by malicious actors. The biggest culprit is Malware, a portmanteau for malicious software. Malware takes on many forms, but put simply, the ultimate goal of Malware is to carry out a series of actions for nefarious purposes. Whether the end goal is espionage, disrupting services or exploiting systems for financial gain, the costs associated with inaction is increasing every year as new Malware variants are deployed on unsuspecting enterprises and victims. Every year several Anti-Virus (AV) vendors publish their annual white papers regarding the current state of Malware worldwide. From a research standpoint, researchers are concerned with three aspects of Malware behavior: the ability for Malware to disguise its own structure to avoid detection; modification and/or utilization of the host Operating System (OS) resources; and the communication Malware aims to establish externally [12] to so-called Command and Control servers (CnC). These aspects of Malware behavior can be summarized as the following:

Obfuscation - Malware employs the use of various obfuscation techniques, such as packing and encryption, in order to avoid signature-based detection methods. Obfuscated Malware also makes it cumbersome to disassemble and produce accurate Control-Flow Graphs (CFG) when Reverse Engineering.

Resources - Malware will utilize various resources of the host operating system in order to carry out its predefined objectives. Malware will call several Windows application programming interfaces (APIs), make changes to the registry, read and write to the file system, as well as create and spawn new daughter processes and threads.

Network - Malware will attempt to communicate with an outside CnC server in order to relay information. Communication may be to serve a greater botnet network, relay personal confidential details obtained from surveillance of the target OS, or used in detecting the presence of a sandbox environment in anti-emulation and stealth Malware.

The scope of Malware worldwide is widespread and includes infections in both Macintosh and Windows OS - affecting businesses, governments and individuals alike. A total of 20% of individuals have experienced a Malware attack in one form or another, a 14% increase from 2018 [13]. Estimates obtained for 2019 identified 24 million Windows and 30 million Macintosh infections being recorded [14]; with Kaspersky noting over 24 million unique malicious objects being detected in 2019 alone [15]. While infections recorded span several different types of OS, approximately 94% of Malware developed is, in fact, Windows targeted [16, 17]. Malware takes on many shapes and sizes, and include archetypes such a Trojans, Adware, Spyware, Viruses, Worms, Ransomware, Rootkits, Exploits, Cryptojackers and Keyloggers. These all carry out some form of invasion, damage, or disabling of systems for the direct or indirect benefit of the malicious actor. More recently, the availability of free and open source software distributions has posed significant risks, as so-called "script kiddies" - which are users who have little to no experience in writing software themselves – have made use of these tools for nefarious purposes. The readily available access to distributions such as Remnux and Kali Linux (Offensive Security, New York City, NY) has made it even easier for users to deploy various forms of reconnaissance and penetration testing tools with out-of-the box software. As Natural Language Processing (NLP) tools become more sophisticated, chatbots, such as ChatGPT, can act as personal advisers in red-teaming and blue-teaming drills; which can also subsequently be used by black hats for their own pentesting campaigns.

Businesses are some of the most susceptible recipients to Malware attacks, as they are a potential victims to Ransomware attacks for monetary gain and experience service downtown due to Denial of Service (DOS) attacks. For example, in late 2019 the average downtime for a Ransomware attack was 16.2 days and the average ransom payment was 81,116 USD; almost doubling from 41,198 USD seen earlier in 2019 [18]. The average cost of a data breach to a business was estimated at 3.8 million USD [19] and the average cost of a DOS attack was placed at 2 million USD [20]. The prevalence of Malware in the business environment is evident, with 95% of organizations recording a malicious infection [21] and 81% having been affected by such an infection [22]. While total Malware detections have seen a small increases of 1% year-over-year, the business sector has seen a 13% increase in 2019 [14, 23]. The top 10 Malware variants which target business infrastructure saw triple digit increases in their number of infections between 2018 and 2019 [14]. Small businesses represent 43% of infected businesses reported, likely due to their inability to mitigate, flag and respond to infections appropriately [18]; and the fact that 37% of businesses spend less than 200,000 USD on Internet Technology (IT) security and 78% do not have a formal incident response plan in place [24, 25]. Security experts encourage IT security personnel to adopt the 1-10-60 rule: threats are to be detected within the first minute, threats are to be investigated in 10 minutes, and an appropriate

action must be taken within the first 60 minutes [26]. Businesses are prime targets for Malware due to the financial motivation, with 71% of all breaches being financially motivated and 25% being motivated by espionage [19, 18]. Furthermore, North America is one of the leading regions where corporate Ransomware is a pressing concern, with 68% of businesses having experienced attacks in the last year [27].

AV vendors are particularly interested in the emergence of new forms of Malware because these represent unique instances of Malware that have never been seen before and they pose a significant threat to security infrastructure. A report by FireEye noted over 100,000 unique Malware signatures are being reported each day by AV vendors [21]. Zero-Day attacks are of particular concern as they require AV vendors to develop signatures of these new Malware instances, requiring significant domain-level knowledge and constant revision of their signature database. New and emerging threats are evident, with 60% of Ransomware variants identified in the last 6 months of 2016 being developed in the last year [28]. Moreover, a small but mutable subset of Malware variants, totaling only 50 Malware families, were noted to make up 80% of all successful Malware infections [21]. This propensity for Malware infections to originate from a small family of Malware instances is due to the Polymorphism built into their development. Polymorphism allows for Malware to change their signature upon each iteration of its propagation, leading to previously unseen threats and new instances of Zero-Day attacks [29, 30, 31, 32]. As the stakes increase for both cyber criminals and businesses, so has the tools they develop to penetrate and mitigate threat vectors, respectively. The call for Cyber Security expertise has never been at its highest, with 62% of organizations planning on investing more in Cyber Security in 2020 [33]. The prevalence of Polymorphic Malware and its variants has expanded how we approach the field of Cyber Security for threat mitigation. Legacy methods, which classify new Malware based on previously known signatures, are no longer effective in identifying Polymorphic Malware [34]; lending credence to the development of a more adaptable, behavioral and cognitive-based approach to how we detect Malware [35]. The vast majority (93.6%) of Malware observed today is polymorphic [36], and the necessary steps must be taken to ensure our Instruction Detection Systems (IDS) and Security Information and Event Management (SIEM) systems are equipped to keep up with the ever-mutating nature of today's Malware landscape. This behaviour to avoid detection is, in of itself malicious, and it is suggested in this proposed work that we can develop models that can alert to this behaviour by examining usage patterns in Registry, File System, and Process and Thread Activity on a Windows OS.

Chapter 2

Literature Review

This review will provide a thorough review of the field of Malware Analysis; starting from the limitations of current signature-based methods to the various obfuscation techniques employed by Malware. Then, a section on feature analysis and application of Graph Neural Networks follows. The survey of published works covered in this Literature Review has already been published in a slightly modified form suitable for journal publication found in [11]. An additional section is written to cover the modelling of malicious threats, which is unique to this thesis document.

Section 2.1 will cover basic signature techniques used by AV vendors including the most common scanning techniques and considerations for scanners. Section 2.2 builds on the limitations of these techniques by introducing Malware obfuscation, which are the most commonly used routines used by Metamorphic engines in its obfuscation stage. In Section 2.3 the idea of obfuscation is put into perspective with a deepdive into a Metamorphic Engine: which involves the ability of Malware to unpack, obfuscate, compress and encrypt its on payload on the fly. Finally, Section 2.4 provides a overview of the most well-studied datasets used in Malware research, with Section 2.4.2 covering popular Metamorphic kits that can be used by researchers to create their own metamorphic binaries. A brief discussion on anti-armoring techniques follows which discusses the challenges researchers face in isolating Malware variants in a controlled environment.

The second part of this review focuses less on Threat Research and Reverse Engineering more more on the Prevention, Detection, and Modelling of malicious threats. First, a brief overview of how the Windows OS operates in Section 2.5 through the use dynamically linking, as well as how Microsoft provides native APIs for users to create applications to request system resources from the hardware. There will also be security considerations and potential exploits discussed, along with the ways in which malicious actors make use of Windows' legacy support and ease-of-use to gain control of the host OS (Section 2.5.2). This is followed by a brief introduction and overview of graph-frameworks using in Deep Learning (DL) as it relates to Malware classification and anomaly detection in Section 2.5.3. Finally, a section on Complexity is presented which sets the stage for computing Complexity with a DL framework, a first of its kind in Section 2.6.2. To begin, an introduction to the most primitive form of malware analysis: Static Signatures.

2.1 Signature Analysis and Creation

Signatures are used to help identify malicious code segments present, either existing as independent executables or attached to benign files known as Benignware. It is imperative that AV vendors constantly update their signature databases in order to cross-reference known malicious binaries with files suspected of being malicious. Acting as unique fingerprints for Malware, signatures are plagued with several fundamental issues. First, signatures are incapable of identifying emerging Malware variants. In an environment where approximately 60% of new Ransomware are never-before-seen variants according to the most recent estimates [14], this creates a significant shortfall in detection rates for new variants. Additionally, when the vast majority of Malware is polymorphic [36], signatures are sometimes not generalized to catch obfuscated instances of previously flagged Malware.

The art of file scanning is in of itself a laborious process, requiring trade-offs between speed and specificity. Incorporating longer signatures provides a more specific identification of Malware and Malware families, but are unable to catch the subtleties of minute changes [37]. Short signatures provide better coverage but results in more false positives [38, 39]. AV vendors therefore must come up with a series of rules to both generalize their signatures and improve their scanning efficiency. Some of the basic scanning strategies are shown in Table 2.1 and described in the following:

Table 2.1: Summary of byte (strings) scanning techniques. Legend: (?) Wildcard, (%) Mismatch.

	Description	Example
String Scanning	Searches for sequences of common strings characteristic of Malware	`AA AB AC AD AE AF`
Wildcards Method	Searches for sequences of common strings while introducing wildcard variables	`AA ??  AC %3 AE AF`
Mismatch Method	Searches for sequences of common strings, regardless of position	`AA AC AE AF`
Generic Detection	Searches for common strings combinations typical of Malware families	`AA ??  %2 AE AF`

String Scanning is the defacto standard for any string match scanning. The scanner is to look up the exact sequence of bytes in any offset.

Wildcards Method allows for the use of wildcard variables. In the example shown in Table 2.1 the use of `??` acts as a placeholder for 2 bytes of any string; while `%3` prompts the scanner to look for the subsequent byte sequence in any of the proceeding 3 byte positions. This is extremely effective for catching Register Swapping and Instruction Replacement obfuscations.

Mismatch Method incorporates the idea of partial match of any given byte sequence. In the example provided in Table 2.1, if the scanner allows for up to 1 mismatch, as long as 2 of the 3 byte sequences are found the scanner alerts to a match.

Generic Method allows for the detection of Malware families through the use of both Wildcards and Mismatch sequences. This method extracts the core Malware artifacts of a Malware family, thereby capturing any subtle alterations to the bytecode sequence that may arise in the future. For example, the Win95/Regswap virus uses similar opcodes between generations. Through a combination of wildcard string matching with mismatch, the entire Regswap generation can be flagged based on a few common signatures.

In addition to generating unique signatures as part of generating a greater signature database for Malware, scanning files requires a dedicated strategy – and in some cases – dedicated hardware. For

example, while a signature may be located in any one of the portable executable (PE) sections, such as *.idata*, it may also be located in the PE file header. Additionally, if you wish to cross-reference thousands of malicious signatures with an incoming data stream using Regex patterns, you would have to take advantage of intra-packet or inter-packet scanning to process them effectively [40]. AV vendors utilize cheaper operations, such as checking the file length, before committing to the use of a more arduous task such as a checksum [39]. In practice a signature can act as a representation for a series of bytes, a whole file, or certain sections. The ways in which AV vendors carry out simple scanning on a binary is described in the following sections.

2.1.1 Top-and-Tail Scanning

This mode of scanning used to extract signatures from the top and bottom of files. This is especially useful for viruses that append to the front or back of the targeted host program. Since the address of the main entry point of a program is in its header section, manipulation of this address to point to the appending malicious binary is possible [41]. As an example, the Polimer.512.A virus pre-appends itself at the front of the executable and shifts the original program content after itself. Alternatively, the Vienna virus is 1,881 bytes long and appends itself to the end of the host file.

2.1.2 Entry Point Scanning

This mode of scanning is used to extract signatures from the sequence at program entry points. Malware routinely alters program entry points as to avoid detection through re-routing of the execution flow to a decryptor stub which decrypts the original binary [42]. The Zmorph virus follows such behavior, whereby the decryptor aims to rebuild the instructions line by line by pushing the result into the stack memory. This can lead to "black hole" scenarios where useless operations are compiled early on in the process flow to burden the Reverse Engineering analysis.

Additionally, an assembly encoder or an altered `JUMP` statement can be configured to run encoded information in a "code cave", as to not increase the file size of the binaries. This would normally impact the binary file header values and any changes will alter relative/absolute offsets, so the pointers need to be changed accordingly. As previously mentioned, the Polimer.512.A virus appends itself to the infected program and in doing so is exactly 512 bytes long. This would raise flags and be easy to identify possible infected files due to the consistent file size differential.

Viruses such as the Win32/Simile is able to avoid changing the entry point of an infected file by altering call instructions which reference `ExitProcess()` to point to the virus code. This has the effect of not changing the entry point of the infected file. Other viruses such as W32/Bistro and W32/SMorph obfuscate their entry point [43]. SMorph is able to use existing API calls in the infected file to call to its own import address table containing references to API imports.

2.1.3 Integrity Checking

This mode of scanning can be an extremely powerful tool to detect manipulation of system files which should never change [38]. A checksum database can be used for reference when performing routine integrity checking of the system and files to detect any alterations [44, 45]. Common checksums include MD4, MD5 and CRC32. Checksums are routinely used on byte values suspected areas of a virus body, thereby reducing the number of total checksums required.

Alternatively, certain types of infections, such as companion infections, may attempt to mimic the name of an infected file and redirect the header section of an EXE which stores the address to the main entry point of a program to the start of the virus code [46]. The virus may also change the extension to COM as the Windows OS give a higher priority to COM over EXE extensions. In order to account for this, distributions such as McAfee's Network Security Platform can assign a magic number to file types and will flag files whose extensions have been tampered with [47].

2.2 Obfuscation

This chapter will provide an overview of the common obfuscation techniques employed by Malware. Examples of these techniques will be provided, along with some actual code snippets from popularized Malware variants. Finally, a brief overview of Encryption and Compression is given - two very important techniques to familiarize yourself with. This chapter will focus on obfuscations made specifically via changes to the opcodes and operands, which serve as the CPU instruction set which specifies the data that is processed and how it is done. Opcode examples will include both Intel and AT&T syntax, with the former being readily apparent as the source operand is always on the right side of the instruction and the destination on the left (e.g. `mov eax, 1`).

2.2.1 Dead-Code Insertion

Dead-code insertion – or sometimes referred to as garbage code insertion - is an obfuscation technique which inserts byte code sequences into a binary without affecting functionality [48, 49, 50, 51]. This obfuscation relies on the fact that instructions can be added to code which do not perform any meaningful function – or in other scenarios, can carry out an instruction and perform the operation in reverse [52, 53]. An example of this type of obfuscation is shown in Table 2.2 where a series of `nop` instructions are used to pad the instructions. Typically, Dead-Code insertion is used to carry out one of three functions:

1. Insertion of a pointless operation such as `nop`, `mov eax, eax`, `add eax, 0`, `and eax, -1` or `or eax, 0`. In practice these instructions do not change the content of CPU registers or memory as they are all semantically equivalent to nop; however, they may modify the status of the flag register in the CPU. These instructions also have different opcodes.

2. Insertion of operations with the purpose of burdening the Reverse Engineering process by altering values in registries and then reversing the instruction. An example would be incrementing a registry `add eax, 1` and then reversing the instruction by decrementing `sub eax, 1`. Other examples would be `push` and `pop`, and `inc` and `sub`. This form does change the values of the CPU registry, but simply undoes the operation sometime afterwards.

3. Insertion of dead code within branches of code that are never actually called – which may or may not make changes to variables in other branches of code which are never executed. An example would be a set of variables in `Function A` which are manipulated, but `Function A` is never executed because it is bypassed with a `jmp` statement.

Garbage code insertion is used successfully in the implementation of W95/Bistro, a later implementation of W32/Zperm, which utilizes a random block insertion engine which is placed directly

Table 2.2: An example of dead code insertion using `nop` before and after obfuscation by a mutation engine.

Before Obfuscation	After Obfuscation
	xor eax, eax
xor eax, eax	move eax, 0x2D
move eax, 0x2D	nop
mov ecx, 0xA	nop
	mov ecx, 0xA

after the virus entry point. Upon entering this block of code millions of instructions are ran, thereby over-burdening the emulator before the virus instructions are even executed. Other popular examples of viruses utilizing garbage code insertion are W32/Evol and W32/Zmist. Zmist is notable for its use of the Executable Trash Generator (ETG). W32/Evol in particular is able to utilize garbage code insertion to produce very different variants with different opcodes and string signatures; thereby evading signature scanning techniques as no sequence of bytes is similar between the two generations. An example of 3 variations of the same code is shown in Table 2.3.

Table 2.3: Three versions of the E32/Evol virus following obfuscation through garbage code insertion and encryption. Retrieved from [54].

	Opcode	After Obfuscation
Version 1	C7060F000055	mov dword ptr [esi], 5500000Fh
	C746048BEC5151	mov dword ptr [esi+0004], 5151EC8Bh
Version 2	BF0F000055	mov edi, 5500000Fh
	893E	mov [esi], edi
	5F	pop edi
	52	push edx
	B640	mov dh, 40
	BA8BEC5151	mov edx, 5151EC8Bh
	53	push ebx
	8BDA	mov ebx, edx
	895E04	mov [esi+0004], ebx
Version 3	BB0F000055	mov ebx, 5500000Fh
	891E	mov [esi], ebx
	5B	pop ebx
	51	push ecx
	B9CB00C05F	mov ecx, 5FC000CBh
	81C1C0EB91F1	add ecx, F191EBC0h; ecx=5151EC8Bh
	894E04	mov [esi+0004], ecx

The use of garbage code insertion techniques is useful in avoiding AV scanning for two reasons. First, the garbage code inserted is unique to each virus generation, thereby sidestepping previously seen AV signatures [55]. And secondly, garbage code from Benignware can be inserted into Malware to increase the false negative rate. In [56] the authors created binaries with approximately 30% of dead code along with 10% benign code and showed similar classification scores as Benignware. In the work of [57] ranges of garbage code between 5-35% were used to determine their effectiveness at evading detection - with 10% being sufficient. In an earlier work [58] combined various proportions of garbage code insertion with subroutine re-ordering to total 25 different combinations. Two different obfuscation engines, AVFUCKER and DSPLIT, also known as Crypters, were used in [59] to produce obfuscated code with dead code insertion. Since there is a wide variety of permutations – from single `nops` to inter-meshed garbage code blocks - upon which garbage code insertion can take form, string scanning is fairly ineffective against this form of obfuscation.

2.2.2 Registry Reassignment

Registry Reassignment, or sometimes referred to as Registry Renaming, is an obfuscation technique which swaps unused registers or memory variables with those currently used by the program [55]. In its simplest form, as demonstrated in Table 2.4, registry reassignment can replace the `eax` registry with `ebx`, with no change in functionality.

Table 2.4: An example of simple Registry Reassignment before and after obfuscation by a mutation engine. Retrieved from personal work in [60].

Before Obfuscation	After Obfuscation
mov eax, ecx	mov ebx, ecx
xor ebx, ebx	xor eax, eax
test eax, ebx	test ebx, eax

The downside to using registry reassignment is that string scanning techniques, such as Wildcard or Half-Byte techniques, can be used to detect any possible combination of registry used. This in effect will provide a constant string between generations of registry reassignment, rendering them easily flagged by scanners. The virus W95/Regswap (hence the name) effectively made use of registry reassignment as demonstrated in Table 2.4:

Table 2.5: An example of the Regswap virus. Adapted from [61].

	Opcode	After Obfuscation
Version 1	5A	pop edx
	BF04000000	mov edi, 0004h
	8BF5	mov esi, ebp
	B80C000000	move eax, 00Ch
	81C288000000	add edx, 0088h
	8B1A	mov ebx, [edx]
	899C8618110000	move [esi+eax*4+00001118], ebx
Version 2	58	pop eax
	BB04000000	move ebx, 0004h
	8BD5	mov edx, ebp
	BF0C000000	move edi, 000Ch
	81C088000000	add eax, 0088h
	8B30	mov esi, [eax]
	89B4BA1811000	move [edx+edi*4+00001118], esi

In Table 2.5 the string signature of Version 1 and Version 2 have a 60% similarity when it comes to their hexadecimal representation [61]. With the help of Regex expressions, the accuracy is greatly increased with variations of a similar instruction set [62]. Along with garbage code insertion, these primary obfuscation techniques make it considerably harder to flag new variants of Malware.

2.2.3 Instruction Substitution

The Instruction-Substitution technique introduces an additional layer of obfuscation on the existing techniques discussed. The power of Instruction-Substitution comes from the fact that there is a seemingly endless diversity to the substitutions you can introduce to an existing instruction framework. Table 2.6 demonstrates an example of a 2-4 instruction substitution (2 instructions are replaced with 4 to perform the same function) [63]. Another instruction substitution would be `push eax; mov eax, ebx` with `push eax; push ebx; pop eax`. Semantically these are equivalent, but `push, pop` is in fact slower as it is quicker to direct registry write with `mov`. This exact substitution is utilized by the W95/Zmist virus, along with interchanging `xor/sub` and `or/test` instructions.

Table 2.6: A simple example of Instruction Substitution before and after obfuscation by a mutation engine. Retrieved from personal work in [60].

Before Obfuscation	After Obfuscation
	add eax, 01H
add eax, 05H	add eax, 05H
mov ebx eax	push ebx
	pop eax

Instruction-Substitution is utilized very effectively in several high-profile viruses such as Evol, MetaPHOR, Zperm and Avron. Since instructions substitutions produce different opcode representations, this renders opcode frequency and accompanying n-gram techniques effectively useless. Researchers have attempted to draw from the basic set of fundamental operations in order to track the Malware's original intentions. In [64] a clue set was established for the Evol virus in which all rewritten instructions were based upon. This approach was found to be very effective at characterizing the metamorphic engine Evol uses. A similar approach was taken by [65] where the complex instructions the virus would create was transformed back into their simple representations using their similar semantics. In Table 2.7 two versions of the W95/Bistro virus are shown – using different instruction substitutions in each generation. Similar to registry reassignment the generations contain similar string signatures, making them susceptible to Wildcard and Half-Byte scanning techniques. While this manuscript is focused on obfuscators based on the Intel x86 instruction set, compile-time instruction set obfuscators can also create semantically similar rules sets for basic operations in other instruction sets [66, 67].

Table 2.7: Instruction replacement used by the Win95/Bistro virus. Adapted from [61].

	Opcode	After Obfuscation
Version 1	55	push ebp
	8BEC	mov ebp, esp
	8B7608	mov esi, dword ptr [ebp+08]
	85F6	test esi, esi
	743B	je 401045
	8B7E0C	mov edi, dword ptr [ebp+0c]
	09FF	or edi, edi
	7434	je 401045
	31D2	xor edx, edx
Version 2	55	push ebp
	54	push esp
	5D	pop ebp
	8B7608	mov esi, dword ptr [ebp+08]
	09F6	or esi, esi
	743B	je 401045
	8B7E0C	mov edi, dword ptr [ebp+0c]
	85FF	test edi, edi
	7434	je 401045
	28D2	sub edx, edx

2.2.4 Code Transposition

Code Transposition, or sometimes called Instruction Permutation, is an obfuscation technique which utilizes conditional or unconditional `jmp` statements to reorder single or blocks of instructions [29].

Since `jmp` instructions can theoretically be used for every line of instructions, the total number of permutations $m!$ is proportionally to the number of lines rearranged m [55]. Code Transposition carries out a very similar function as Subroutine Reordering with the exception that there is a change in the process flow; therefore, they will be discussed together. Subroutine Reordering, also known as Block Reordering, is an obfuscation technique that reorders the process flow by rearranging blocks of code that have independent subroutines [68]. If a program were to be categorized into n number of subroutines, then $n!$ permutations of subroutines are available for rearrangement [51, 69, 62, 70]. A simple program with 10 subroutines would therefore be able to produce over 3.6 million possible iterations. Subroutines require the instructions set are independent of one another, allowing them to be reordered without having an impact on functionality. In Table 2.8 an example of a set of instructions exhibiting multiple forms of obfuscation is shown. In the example Code Transposition, Subroutine Reordering, Garbage Code Insertion and Instruction Substitution are all used.

Table 2.8: An example of Code Reordering and Code Transposition in combination with other obfuscation techniques before and after obfuscation by a mutation engine. Retrieved from personal work in [60].

Before Obfuscation	**After Obfuscation**
	mov ebx, 10
	jmp F1
	jnk
mov eax, ecx	F2: push edx; jnk
mov ebx, 10	pop ecx
mul ebx	jmp F3
add eax, 5	F1: mul, ebx
mov ecx, eax	add ecx, 1; jnk
	add ecx, 5
	jmp F2
	F3: mul ebx

Several `jmp` statements are employed to permute blocks of instructions which are can be run independently from each other. Instruction Substitution is used to add more sophisticated instructions based on the simple instruction set `add eax 5; mov ecx, eax`. `jnk` insertions are used to add complexity to the existing code, as well as added following the `jmp F1` statement where it is never actually executed. This `jnk` could include code from Benignware that would normally fail to compile if it were embedded within the existing obfuscated framework – but may confuse scanning techniques nonetheless. Table 2.8 also displays another form of obfuscation called Subroutine Outlining [43]. This obfuscation explicitly turns instruction blocks into subroutines and uses the `call` instruction to perform an unconditional jump to the location indicated by the label operand. Subroutine Inlining would carry out the reverse: where subroutines would be unraveled and placed in order to preserve the process flow. Unlike simple `jmp` instructions, `call` preserves the locations to return to when the subroutine is completed.

This sophisticated form of obfuscation is used by the W95/Zperm and W32/Ghost viruses, with the former employing the use of the Real Permutation Engine to perform Subroutine Reordering. Zperm divides the code into frames which are independent subroutines, which are then repositioned randomly and connected using branch instructions to preserve process flow. When Zperm initializes it allocates a buffer sized at 64Kb filled with zeros, and then fills it with obfuscated code and randomly positioned `jmp` statements [54]. This means that a constant body is never generated between generations and is never present in memory. Similar to Table 2.8, garbage code is inserted

between frames to fool string detection similar to the Zmist virus. W95/Zmist also inserts `jmp` instructions after every instruction, making it the perfect shield to heuristic detection. In [50] 30% subroutine reordering was used to sidestep a developed similarity metric that compared Benignware to Malware based on the similarity of their transpositions. From a security analysis standpoint, it is extremely difficult to know when the virus begins when it is embedded within existing code and is encrypted. Partial emulation is one avenue whereby code can be reconstructed and then used to completely decrypt the virus. But when to decrypt during emulation is still a laborious process in of itself.

2.3 Encryption, Compression and Metamorphism

Metamorphism, and more generally obfuscation techniques, make up the backbone for most new and emerging malicious threats we see today. As the signature-based scanning techniques improved for AV vendors, so did the levels of obfuscation employed by malicious actors to thwart said techniques [61, 71]. Along with obfuscation came various forms of armoring, stealth-behavior and anti-emulation tactics, which made the job of a security researcher that much more burdensome.

To understand how mutation came to be it is worth mentioning the earliest forms of obfuscation and how they came into existence. Viruses make use of Entry Point Obscuration (EPO) in order to avoid any consistency in the execution order of the virus code in relation to the infected file. As shown in Fig. 2.1, the file header would point to an address that would execute virus code, which would then point back to the host file so that the virus execution would do so unknowingly.

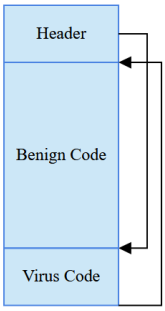


Figure 2.1: Illustration of an appending virus that latches onto the end of a benign file. Retrieved from personal work in [60].

The CASCADE virus in 1986 became one of the first known viruses to implement encryption, thereby requiring a separate Decryption Routine to carry out decryption and push the instructions into memory for execution. Since the form of encryption would become apparent as the virus propagated, the decryptor routine itself would have to be mutated – leading to the establishment of the first series of Oligomorphic viruses.

2.3.1 Oligomorphism

Oligomorphism began as a reaction to the signature-based scanning techniques widely utilized for flagging possible virus infections. With the help of scanning techniques such as Wildcard and Mismatch, a greater swath of possible infections could be characterized by a few unique signatures.

Furthermore, since virus code would either append or pre-append onto an existing file, Top-and-Tail Scanning was an effective tool for extracting signatures from certain select sections of a file. Emulators could also be utilized to uncover the Decryption Routine used in the encryption, meaning the Decryption Routine itself had to be altered in some form or another. Emulators wait as the virus is decrypted one instruction at a time and as it rebuilds itself by pushing the stack into memory. Once control is sent to the stack memory, the emulator monitors the stack and the code can be dumped. Oligomorphic Malware were the start of a new breed of Malware which would involve obfuscation of the routine itself, meaning viruses were unique among their generation. The first Oligomorphic virus was the Whale DOS virus first identified in 1990. In Fig. 2.2(a) an obfuscated, encrypted Decryption Routine is used to carry out decryption of the virus body and to avoid detection.

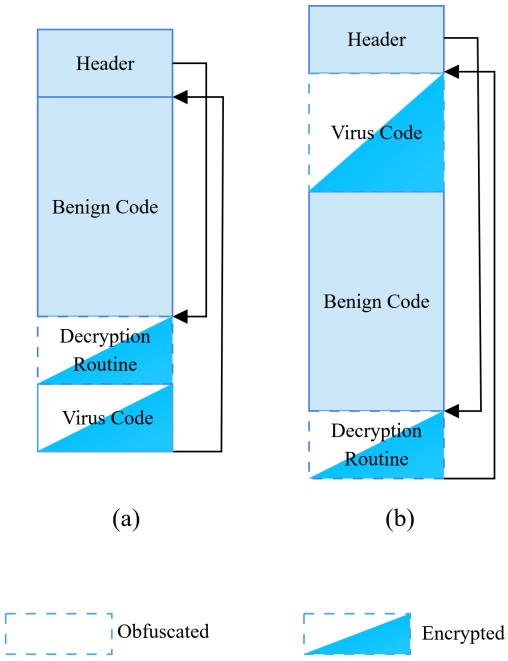


Figure 2.2: An illustration showing the variation in positioning and level of obfuscation found in (a) Oligomorphic and (b) Metamorphic Malware. Retrieved from personal work in [60].

However, a major limitation to Oligomorphism is that the loop of possible decryptors is finite. For example, the W95/ Memorial virus had exactly 96 different decryptors to choose from. Once an Oligomorphic generator is exhausted, the entirety of its possible generational variance is also exhausted and understood. The natural extension to this problem is to introduce obfuscation into the Decryptor Routine itself, leading to an infinite number of possible Decryption Routines [72]. This led to the first generation of Polymorphic viruses such as 1260, and popularized generators such as Phalcon/Skism Mass-Produced Code Generator (PS-MPC) and Virus Creation Lab (VCL) – which are still used to this day.

2.3.2 Polymorphism

Polymorphic Malware was seen as a complete package: complete with a compiler that could decrypt and obfuscate then recompile everything back together. The unencrypted virus body would create

a new mutated decryptor using a random encryption algorithm, and then allow the decryptor to encrypt itself before linking both sections back together. However, the core problem of emulation remains: the virus code section would be decrypted into memory and be able to be detected and flagged by security researchers. It was also the case that prior generations of obfuscators suffered from several limitations [73]:

1. Constant size of virus code between generations (Polimer.512.A or Vienna viruses).
2. Appending or pre-appending to the infected host file meant signature scanning could target these sections exclusively.
3. Similar virus code segments between generations means the virus is subject to entropy analysis.

In order to build on some of these deficiencies, the introduction of the Metamorphic engine came to be.

2.3.3 Metamorphism

The introduction of Metamorphic Viruses introduced the idea for the first time that no two generations of viruses can have similar signatures, as no constant body is present like with Polymorphic Malware [54]. In Fig. 2.2(b) an example of a Metamorphic virus is shown. Unlike Polymorphism the virus code is obfuscated, meaning the entirely of the virus is present in an obfuscated state. This introduces the fundamental issue: since "Metamorphics are body-Polymorphics" [74], and as a result have no constant body, they reinforce the notion that anomaly-based detection is NP-Complete [75, 76, 76]. The first Metamorphic viruses were W95/Regswap in 1998 [77] followed by W32/Ghost identified in 2000 [78]. W32/Ghost contained 10 submodules, so over 3.6 million possible variations were possible with Subroutine Reordering.

In light of the graphic shown in Fig. 2.2(b), the separation between decryptor and virus body is no longer possible and the level of obfuscation means encryption is no longer needed. Furthermore, as is typically the case, the Decryption Routine is scattered in the benign code. The executed code in the virus body mutates entirely along with the decryptor, and it does not need to unpack to create a new constant virus body like Polymorphics [62]. One of the most utilized and effective metamorphic generators is W32/NGVCK created in 2001. Metamorphic viruses have a sophisticated mutation engine that contains many subprocesses. These will be discussed in the following section.

2.3.4 Metamorphic Engine

A Metamorphic Engine is responsible for the obfuscation and reconstruction of the binary so that the file can remain operational. In Fig. 2.3 an illustration of a complete Metamorphic Engine is shown. Some of the key components of the metamorphic engine are described as the following [79, 77]:

Disassembler is responsible for turning the source code into assembly instructions. This creates an intermediate form that is independent of the CPU architecture for future adoption with different OS and CPU architectures [54]. Within the disassembler a code analyzer provides info for a code transformer module that gathers information related to control flow, subroutines, variables and registers.

Shrinker eliminates much of the garbage code produced from previous generations, and mainly eliminates garbage and other non-sequential code that is produced from obfuscation. This step also carries out Code Shrinking, a form of Code-Substitution that will turn previous 1 to 2 or 1 to 3 instruction substitutions back to their semantically similar primitive equivalents [80].

Permutor carries out much of the obfuscation using permutations of subroutines, many times in a randomized fashion. Insertion of `jmp` instructions is also common to divert control flow.

Expander performs Instruction-Substitution to convert instructions to another equivalent instruction set. Additionally, registries are reassigned and variables are re-selected according to fixed probabilities using substitution tables [77, 81]. Garbage and other do-nothing code is added and functions are inlined/outlined [82, 83] Both the Permutor and Exander steps are quite sophisticated in the metamorphic W32/Etap and W32/Zmist viruses [72].

Assembler restructures the control flow and converts the assembly code back into machine binary code where it can become operational again.

Virus Code contains the core instruction set that will execute on all new generations of the virus. Also contains the instructions that coordinates the mutation engine and other components.

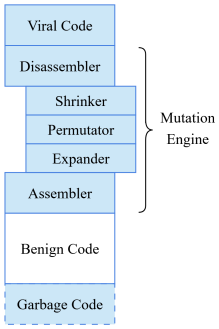


Figure 2.3: Overview of the major components of a Metamorphic Engine. Retrieved from personal work in [60].

The mutation engine does not have to operate at the assembly and or source code level but can also operate at an intermediate representation (IR) bytecode level [82]. In [84] and [85] morphing techniques are seen as deterministic automata, whereby transitions following a formal grammer are made to symbols and new mutations are produced. In [81] a template is used which illustrates how simple representations of formal grammar can produce several possible mutations. The depiction shown in Fig. 2.3 includes all the core components with the exception of a Decryption Routine. A metamorphic engine with the addition of a Decryption Routine is shown in Fig. 2.4, and follows a sequence of steps to decrypt, obfuscate and link everything back together. The steps are the following in order:

1. First the Decryption Routine decrypts the Virus Body and executes an instance of it.
2. The Decryption Routine then decrypts the Mutation Engine and executes it.

3. The Shrinker component of the Mutation Engine goes to work to de-obfuscate the virus body.

4. Obfuscation takes place by introducing a new and unique Decryption Routine using the various techniques discussed in Section 2.2.

5. The virus body is then obfuscated by the Mutation Engine to produce a unique generation using the various techniques discussed in Section 2.2. The Virus Body is then encrypted using a unique algorithm, a static key or a host specified temporary key. More on this in the following Section.

6. Finally, the Mutation Engine is encrypted.

Once all three components are re-obfuscated to seemingly new binaries, with the Mutation Engine and Virus Body decrypted, the virus re-links its components back up and can execute on a new host by decrypting its payload through it newly obfuscated decryption routine.

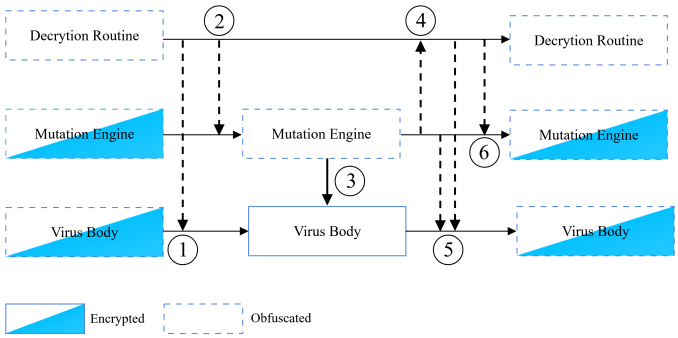


Figure 2.4: Graphical illustration for the Decryption, Obfuscation and Encryption carried out by a metamorphic mutation engine. Retrieved from personal work in [60].

The authors in [69] provide a detailed summary of the production and considerations for creating a Metamorphic generator; as well as in [86] for creating a Metamorphic Worm. One of the more sophisticated metamorphic viruses is W32/Simile, also known as MetaPHOR or Etap. The author, “Mental Driller”, referred to the expansion, contraction and permutation of instructions as the “Accordion Model” [73, 79] based on the changing form that garbage code takes when it becomes obfuscated. The Simile virus was also unique in that 90% of the virus code was dedicated to the metamorphic engine itself – with the decryptor being placed at the end of the code section and the virus body being partitioned elsewhere [54, 64].

2.3.5 Encryption

While Encryption was briefly touched upon at the beginning of Section 2.3, obfuscation engines make use of a variety of encryption techniques to avoid detection [61]. The earliest form of encryption was carried out by the CASCADE virus on DOS [51], and did so using a simple `xor` (see Fig. 2.5):

The Cascade virus, first identified in the early 1900’s, was shown to increase the file size of infected files by 1701 and 1704 bytes, and is mainly comprised of its encryption loop and main body. The virus uses a technique called ”cascading” to conceal its presence. When the infected files are executed, the virus code is executed first, causing the virus to infect more files and directories. This

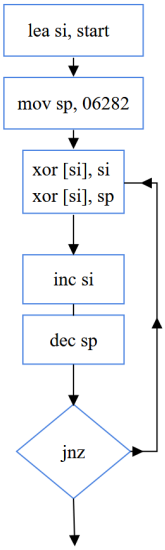


Figure 2.5: Simple `xor` decryptor which decrypts byte by byte using an increment counter and a jump not zero (`jnz`) loop. Retrieved from personal work in [60].

creates a cascading effect, making it difficult for anti-virus programs to detect and remove the virus [87]. The Decryption Routine in Fig. 2.5 is fairly simple: the stack pointer, `sp`, acts as the key and the `si` register is used to keep track of which position of the virus body to point to. As the decryption is carried out both the `si` and `sp` counters increment and decrement by one, respectively, until `sp` returns 0, otherwise it will jump using `jnz`. For example, applying a simple `xor` operation to each byte using an 8-bit value as the encryption key will produce the encrypted text. The string `2D03 002E` when `xor`'d with the key `0xFF` will produce `D2FC FFD1`. Doing so in reverse with the same key will produce the original text – thereby performing encryption and decryption with only one key.

Conventional decryption relies on the virus' own decryptor loop to decrypt the virus body. It did not take long for malicious actors to rely on multiple decryptors instead of one – such as the DOS/Whale virus in 1990 - which utilized dozens of different decryptors and chose one randomly each infection. It may be also the case that rather than the encryption being performed serially, the decryption can be performed in a random fashion, as is the case for W32/MetaPHOR which does so seemingly randomly, with each instruction only being decrypted once. In Malware deployments the use of a Crypter is typically used, which carries out encryption for anti-analysis and obfuscation purposes. A Crypter contains a stub which carries out the decryption and does so while generating a new payload and key with each new generation [88, 59]. All of this occurs in memory and nothing is written to disk. Decryption can take place in the stack, but then the key it is not writable; as opposed to allocating to memory which is easily flagged by emulations that are monitoring memory. On Intel x86 platforms 24 bytes or more of modified memory is characteristic of a Decryption Routine [39]. Once the stub passes control to the virus body after decryption, a new encryption key is created and all executables and `.text` sections are encrypted with the new key. Depending on the file type, a TEA cipher can be used for EXE and RC4 for DLLs as is the case for HackedTeam's core-packer [89]. The key is then stored in the decryptor stub or elsewhere.

Basic Encryption can be performed as mentioned previously with a single decryptor key (see Fig. 2.6), using 1-to-1 byte to byte mapping, with zero operand using `inc` or `neg`, or reversible instructions such as `add` or `xor`. Alternatively, Sliding Key Encryption makes use of starting key which updates as it progresses, and may even utilize the characters most recently encrypted (see Fig. 2.6(b)) or based on an algorithm, as shown in Fig. 2.6(c). Flow Encryption determines a key stream in advance equal to the size of the encrypted text, and then encrypts the body instruction by instruction. Key generation can also be varied amongst decryptor routines, where a key(s) can be located in the decryptor stub itself, hidden among the virus body, generated uniquely from the host system, or alternatively, randomly generated and not stored at all.

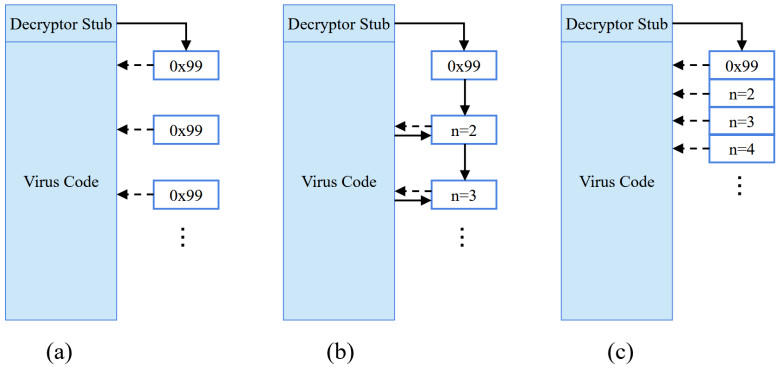


Figure 2.6: Illustration of different encryption archetypes, where (a) key is re-used for each encrypted block; (b) encrypted block is used as nonce for next encrypted block; and (c) stream cipher is used to encrypt each block. Retrieved from personal work in [60].

The sources for the encryption key can vary, but can either be hardcoded in one form or another, or obtained through the host. In the case of Variable key generation, the decryptor can develop the encryption key based on its own function calls. Alternatively, Environmental key generation does not involve any descriptors from the viral payload or stub itself, but rather, retrieves them from the infected host. One example of environmental key generation is the use of a Trusted Platform Module (TPM) chip, which is a hardware component built into many modern computers and devices [90]. The TPM can generate unique encryption keys that are tied to specific physical attributes of the device, such as the device's BIOS, firmware, or other hardware components. This makes it much more difficult for an attacker to access the key and decrypt the protected data, even if they are able to physically access the device. In the case of the RDA.Fighter virus family, the virus checks the BIOS address at `FFFF:000E0`, and if it returns advanced technology (AT), as in AT-class computer, the time stamp is retrieved from the CMOS buffer; otherwise it is retrieved from the system clock. The timestamp is then used to create a 16-bit number that is used to decrypt the next code section using a mirror table lookup as a mask. In addition to time, the current date, timer tick, host filename, and even the hard disk serial number can act as sources for developing the encryption key. As a form of armoring, the key can be stored on a distant web server; and outside of a typical host environment, such as in virtualization or emulation, the virus can disable itself and fail to run.

Decryptions and decryption loops are not limited to a single loop, or to a single key. For example, the RDA.Fighter virus family utilizes 16 layers of decryption and does so in a backwards fashion

– making it a laborious process to automate the disassembling process [39]. Multiple layers of encryption are also utilized by the W32/Harrier and Bradley viruses [91]. To avoid all form of local or external storage of the key, a Random Decryption Algorithm (RDA) can be used to brute force the key. The key can be any generated word value, and the decoding method will check the checksum following the decoding procedure to identify when it has successfully found the key. In the RDA.Fighter family, RDA is used as secondary form of encryption on top of Environmental key generation.

2.3.6 Compression

Compression represents an additional level of obfuscation on top of a possible Decryption Routine and other forms of obfuscation. A Packer is defined as a utility which enacts some form of compression to the executable, either to reduce files size to avoid entropy analysis or introduce a layer of obfuscation to the PE header. It has been estimated that 80% of all Malware uses some form of packer[92], as well as 90% of all worms [54]. Two of the most popular packers are Ultimate Packer for eXecutables (UPX[1]) and ASPACK[2]. In addition to significant compression ratios and great performance, these packers work for a variety of executable formats with no memory overhead due to in-place decompression.

Packers are ultimately tasked with compressing executables with decompressed code and a compressed payload. Packers compress the code to avoid reverse engineering and bypass firewalls. Malware makes use of packers by initially converting an *Image Section* (see Fig. 2.7(a)) into a *Packed Section* and *Unpacking Section* (see Fig. 2.7(b)). The *Unpacking Section* is then set to be the initial point of entry once the file is executed. Upon execution the packed section is decompressed to become the *Unpacked Section* (Fig. 2.7(c)) and is executed on virtual memory [93]. One of the more devious uses of packers in Malware analysis is that the original PE header is hidden as the visible import functions are those utilized by the packer itself. Since packers such as UPX, ASProtect, PECompact and Themida are widely used for non-nefarious purposes as well, there is no sure indication the file is malicious based on the import functions [94, 95, 96].

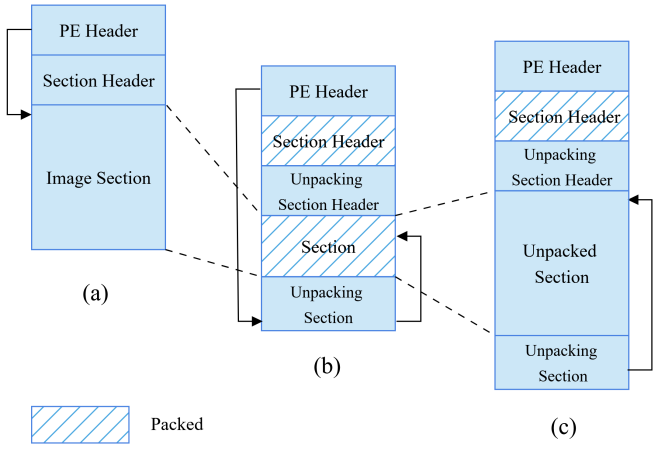


Figure 2.7: Overview of the main steps in a Packer. Adopted from [93].

[1]https://upx.github.io/
[2]http://www.aspack.com/

One of the more comprehensive tools for the detection of malicious packers is the use of entropy analysis [12]. In the work of [97] 28 different packers were used to classify a control flow graph to image representation through the use of a Convolutional Neural Network (CNN). When compression is coupled with encryption, as is the case with so-called *Protectors*, the resulting binary has high entropy levels, making it susceptible to classification. In [70] a file segmentation method that utilized entropy with wavelet analysis was used to classify Metamorphic Malware based on edit distance between file segments. This motivation was derived from the earlier work of [98] that established that the homogeneity of each a Malware's binary section is characteristic of the complexity of its data order. Along with this insight, Polymorphic Malware are able to be identified using these techniques, albeit with a high rate of false positives [98].

In Fig. 2.8 a historic summary is provided – complete with major milestones in obfuscation and new Malware deployments.

2.4 Metamophic Datasets, Generation Kits and Armoring

While Metamorphic Malware has grown in sophistication so has the tools we have available as researchers to thwart their actions. One of such tools and resources is the use of publicly available datasets, such as DARPA99 - a popularized dataset released to improve intrustion detection systems. Datasets encourage the development of classification tools by leaving the details for collecting representative samples in a controlled environment and at scale to others. Secondly, datasets also provide a baseline in which to compare competing algorithms, usually with the aim of increasing true positive rates and decreasing false positives. One of the downsides is that these datasets are typically outdated and are not representative of new and emerging threats. If researchers make raw malicious binaries available, as is the case with SOREL dataset [99], they cannot do the same for benign binaries due to issues with copyright. One workaround used in SOREL is to dump the entire metadata of the binary, and use that metadata dump to create features for a model to learn from. This section will touch on some of the more useful Malware datasets used historically, then transition into covering some aspects of Malware generation kits and anti-armoring behavior.

2.4.1 Malware Datasets

The DARPA dataset was created in 1998 and contains 7 weeks of raw TCP/IP dumps of a simulated attack scenario to an Air-Force base. The dataset contains both host and network files. The KDD99 was created based off the DARPA dataset [100], with a reduced size and a total of 24 attack types and an additional 14 existing solely in the test dataset [101, 102]. Based on the observations of [102], KDD99 was the most widely used dataset in IDS research between the year 2010 and 2015. Several issues arose with the use of KDD99, namely, the time-to-live (TTL) values for benign and malicious packets were different [103, 104], and the data rates were not characteristic of real-world networks [105]. Many of these issues were exemplified in the critique carried out by [104], leading to a need to provide much needed modifications to the existing dataset. Additionally, since the size of the KDD99 datasets was large for many trainable models, and the dataset contained duplicates of attacks such as DOS, the dataset was further reduced to become its most recent version, NSL-KDD [104]. Another dataset containing network traffic is the UNSW-NB 15. The dataset was created by the IXIA PerfectStorm tool at the Cyber Range Lab at the Australian Center for Cyber Security

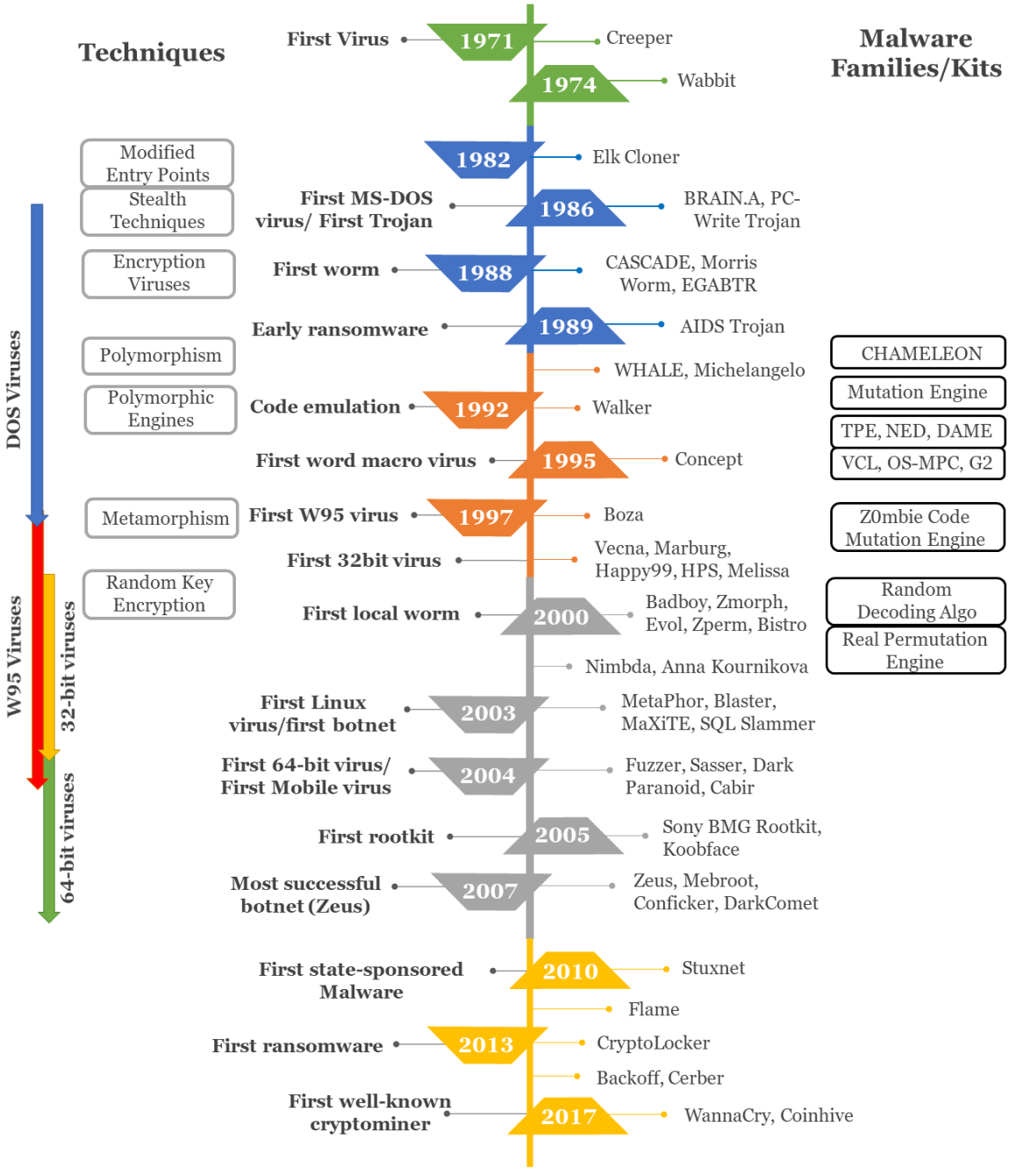


Figure 2.8: Timeline of major Malware variants, techniques, and Mutation Engines. Retrieved from personal work in [60].

[106]. A TCP Dump tool is used to capture 100 GB of raw traffic, with a total of 49 features generated using a set of tools and algorithms. Other lesser known network datasets include CAIDA [107], ISCX 2012 [108] for network intrusion detection and CICIDS2017 [109]. The CICIDS2017 dataset is unique in that the authors included behavior for Windows (XP, 7, 8 and 10), macOS, iOS as well as Linux operating systems – encompassing attacks from Botnets, DoS, DDos, Brute Force FTP, Brute Force SSH, Heartbleed, Web Attack and Infiltration [109]. For a thorough summary of network-based datasets, the authors refer to the review carried out by [110].

Several datasets have been used to represent the content of the Malware binary, versus relying on network activity. One of the more utilized datasets is the Microsoft Malware Classification Challenge dataset, which become popularized in a Kaggle competition back in 2015. The raw data of a virus' binary is represented in hexadecimal, with a compilation of metadata retrieved using the IDA disassembler tool. Binary representations of Malware binary has also become popularized as a dataset in image analysis, with the Malimg dataset [111] having the greatest impact in recent years [112, 113, 114, 115, 116, 117, 118, 119, 120]. Other alternatives include the Malicia dataset [121] which contains 11,668 malicious binaries from 54 families retrieved from 500 drive-by downloads over 11 months. However, the project was ultimately discontinued in 2016. The Malsign dataset [122] contains 142,000 signed Malware and potential unwanted products (PUP) binaries obtained from 2012-2015 for the Windows Platform [123].

Mobile and internet-of-things (IoT) security plays a unique but important role in Malware security, as these devices make up a larger proportion than ever in how we connect with others and exchange information. The Drebin dataset [124, 125] is one of the most used datasets in mobile security, with 5500+ Malware being included in the dataset belonging to 20 families, collected from 2010-2012. The Android Adware and General Malware Dataset (AAGM) [126, 127] includes network activity of 1900 Adware, general Malware, and Benignware running on Android smartphones. The IoTID20 [128] is a more recent dataset used to simulate network attack retrieved from two smart home devices. The dataset consists of 42 pcap files encompassing simulated attacks produced from Nmap and from the Mirai Botnet [129, 130].

Several datasets include features extracted directly from PE files, and this includes the ClaMP and EMBER dataset. ClaMP [131] includes features from the DOS Header, File Header and Optional Header of PE files. The integrated dataset includes 68 features, of which 28 features are from the raw dataset, 26 features are Boolean (File and Optional Header) and 14 are derived features. A second version of the dataset exists which consists of 56 features. Finally, the largest dataset by far is the Ember dataset [132] with a total of 1.1 million binary files. The authors in [133] include additional tools to extract features from the PE files to further encourage the use of the dataset to train benchmark problems. The Ember dataset was the larges of such datasets until the introduction of the SOREL dataset in 2020, which expanded from 1.1 million binaries to 20 million binaries - including 10 million disarmed Malware samples ready for feature extraction [99]. The Australian Defense Force Academy (ADFA) is the author of two datasets: the Linux Dataset (LD) [134, 135] and Windows Dataset (WD) [136]. Both datasets provide a comprehensive simulation of a HIDS based on the collection of system calls; however, a significant downside exists for the ADFA-WD as it was collected solely on Windows XP - which limits the applicability to future generations of Windows OS [136].

Insider threats are considered one of the more emerging sources of security vulnerabilities for

government and firms. CERT identified that 15-24% of firms experience an insider incident perpetrated by a business partner [137]. It has also been noted that a quarter of Cybersecurity risks are due to insider threats – meaning current or close business partners are considered as much of a threat as ransomware from a security standpoint [28]. That is why a dataset such as the CERT insider threat V.2 dataset is so important in our understanding and tracing of threats that exist in network topologies [138]. The dataset includes several synthetic threat scenarios, accompanied with information related to HTTP records, employee info, log on/off times, among other indicators. A summary of the datasets discussed along with some information on their makeup is shown in Table 2.9.

Virus repositories are also a source for millions of malicious binaries and source code for Malware research. theZoo[3] from [139] contains hundreds of malicious binaries that is updated on a regular basis as new threats emerge and as virus source code becomes available [140]. VirusTotal[4] contains one of the most comprehensive repositories used in the industry today. Malicious binaries can be uploaded or searched via MD5 hash to provide a detailed summary of the threat and other metadata. VirusTotal also comes equipped with a Public and Private API that allows threats to be uploaded while returning a detailed report, along with which AV vendors have already developed a signature for the given binary. Virushare[5] is a searchable sample database, boasting 34 million+ Malware samples for use for analysts, researchers and the security community [141]. Other, less popularized repositories for sharing Malware for research purposes include Malshare, VirusBay and dasmalwerk.

2.4.2 Metamorphic Generation Kits

Virus generation kits facilitate the creation of a bulk of the newly generated virus signatures we see every day. These kits perform some, if not all, types of obfuscation outlined in Section 2.2 to evade signature-based techniques; and are a significant problem for AV vendors and researchers alike. Additionally, some kits even provide functionality whereby users can customize the level of obfuscation and encryption to introduce variation into the Malware generation - and are even able to enact anti-emulation and armoring behavior. Some generation kits have been easily flagged by AV vendors since their generated code would contain similar code between generations; therefore, only a few signatures developed could flag the entire generation, rendering the generation kit obsolete. Depending on the generation kit, COM and EXE viruses can be produced directly while other kits generate virus assembly code. For example, Borland TurboAssembler TASM 5.0 can assemble an ASM file into an object file, then TLINK takes the object files and libraries and links them together to produce virus executables. As demonstrated in Fig. 2.9, disassemblers such as IDA Pro can be used to produce the ASM files [278]. The ASM files can then be used to extract opcodes and other features sets for use in Malware classification [279]. This section will discuss several popular generation kits used in research, with a brief description on some of the obfuscation techniques used by each generator.

The Phalcom-Skism Mass Produced Code Generator (PS-MPC) was developed in 1992 and includes over 25 options for different types of encryption and payload types – as well as having options to be memory resident. The generator employs its own decryption routine but lacks options

[3] https://github.com/ytisf/theZoo
[4] https://www.virustotal.com/gui/home/
[5] https://virusshare.com/

Table 2.9: Summary of the more prevalent Malware datasets publicly available for use by researchers. Retrieved from personal work in [60].

	Features	**Description**	**Ref.**
NSL-KDD	21 attacks from 4 families (DoS, Probe, Root 2 Local (R2L), User 2 Root (U2R)), 41 features	125,973 training examples (19.85% benign), 41 features	[142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 104, 167, 168, 169, 170, 171, 172, 173, 106, 174]
KDD99	21 attacks from 4 families (DoS, Probe, Root 2 Local (R2L), User 2 Root (U2R)), 41 features	489,431 training examples (20% benign)	[106, 175, 176, 177, 145, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 102, 104]
DARPA 99	Raw TCP/IP Dump files; 58 attacks from 4 families (DoS, Probe, Root 2 Local (R2L), User 2 Root (U2R)), 41 features	6,591,458 training examples	[200, 201, 202, 203, 204, 205, 206, 207, 100]
UNSW-NB15	Raw Traffic as Pcp files, 9 types of attacks (Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconaissance, Shellcode, Worms); 49 features	175,341 training and 82,332 testings examples, 49 features	[167, 106, 208, 209, 210, 211, 212, 213, 214, 215, 166, 216, 217, 218, 219, 191, 220, 221, 222, 223, 224, 225, 226, 227]
MalImg	Malware binary, converted to 8 bit vector then 8 bit grayscale image	9,339 training examples; 25 Malware families	[228, 229, 113, 230, 231, 114, 115, 116, 118, 119, 120, 111]
CERT Insider Threat V.2	HTTP Records, Emails, Employee Info; 5 unique scenarios of suspicious activity. 191 suspected users	33,771,224 training examples, 33 features	[232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 148, 248, 249]
Drebin	Features extracted from application manifest and dex code. 8 core feature sets. 179 Malware families	5,560 malicious and 123,453 benign applications	[124, 247, 250, 251, 252, 153, 253, 254, 148, 255, 172, 123]
Microsoft Malware Classification	Hexademical representation of binary content with metadata manifest; 9 classes of polymorphic Malware disassembled using IDA packet disassembler	20,000 Malware samples	[256, 257, 258, 258, 259, 260, 261, 262, 112, 263, 228, 264, 114, 115, 265, 266, 258, 267, 268, 269, 270, 271]
ClamMP	Header fields of PE headers; 54 raw features 15 derived	5210 examples (47.75% benign)	[131, 272, 273, 150, 274]
AAGM	Raw traffic from pcap files, 2 types of malicious applications (adware and general Malware)	1,900 malicious applications (80% benign)	[126, 127]
EMBER	Raw features extracted from PE files in JSON format	900,000 training and 200,00 testing examples	[132, 275, 276, 277]
IoTID20	Raw traffic from pcap files, 12 features, 8 attacks types (DoS, ACK Flooding, Brute Force, HTTP flooding, UDP flooding, ARP spoofing, Scan Host Port, Scan Port OS)	40,070 benign and 58710 malicious examples	[129, 130]
ADFA-LD	Linux System Calls; 6 attacks classes (FTP, SSH, poisoned executable, Adduser/Meterpreter, TikiWiki exploit,	2,430,0162 benign and 317,388 malicious examples	[135]

Figure 2.9: Assembly and compilation of a virus executable. Retrieved from personal work in [60].

for stealth techniques. PS-MPC generates files that reside in memory long enough to infect all COM and EXE files. The advantage of PS-MPC at the time of creation was the ability to carry out code generation in batches due to the generator operating as a code-morphing engine as it is script-driven [54]. While all PS-MPC generated code today is readily flagged by AV vendors, the generator is still used today for research on Metamorphic Malware [280, 63, 281, 42, 282, 283]. The Mass-Produced Code Generation Kit (MPCGEN) was first developed in 1993 and was used to create CFG files which were then passed to PS-MPC followed by TASM to produce 32-bit executables. The name "Mass-Produced" comes from the fact that the process of generating, compiling and assembling can be carried out for 500 files in as little as 25 minutes. Similarly, MPCGEN is used to produce a high quality and quantity of metamorphic variants for research purposes [63, 284, 68, 285, 286, 287].

The Second-Generation Virus Generator (G2) was developed in 1993 and produces COM and 16-bit EXE infectors. Employs several code substitution techniques, and as an extension to PS-MPC, introduces anti-debugging and anti-emulation features, as well as resident and nonresident viruses. G2 has easily modifiable source code to allow customization by an advanced programmer, and the routines is uses are semi-polymorphic. G2 to do this day is a go-to for generating polymorphic variants [280, 288, 63, 289, 285, 281, 290, 42, 71, 284, 68, 286, 85, 287, 282, 79, 62, 70, 283].

Virus creation lab for Windows 32 (VCL32) was created in 1992 but was revamped in 2003. Created by a virus writer named Nowhere Man, a member of a group called NuKE, this generator can produce the assembly source code of viruses. This means the assembly code needs to be compiled and linked afterwards before they are active. The versatility of VCL32 comes from being able to customize activation conditions based on, date, time of day, number of infected files, computer country code, version of DOS or the amount of RAM available. VLC32 supports COM file infections, generating companion viruses, as well various encryption and infection strategies. As a complete package with a GUI and drop-down menus, the most recent version VCL32 released in 2004 is commonly used in research [280, 291, 63, 292, 42, 284, 68, 286, 62].

The Next Generation Virus Generation Kit (NGVCK) is one of the more popular virus construction kits available. Developed in 2001 with the most recent version released in 2003, NGVCK has been widely adopted for use in developing 32-bit PE-EXE polymorphic Malware – especially in a research environment [293, 280, 294, 295, 291, 296, 291, 63, 295, 297, 289, 285, 298, 292, 281, 290, 42, 299, 284, 68, 286, 287, 282, 79, 62, 70, 58, 300, 50, 283]. Options for encryption include Rotate without Carry `ROR/ROL`, Twos complement negation `NEG`, Ones complement Negation `NOT`, logical exclusive or `XOR`, Addition/Subtraction `ADD/SUB`. NGVCK can carry out dead code insertion, subroutine reordering, code substitution and registry renaming – all very effective techniques for obfuscation. In [63] NGVCK was compared to other popular generation kits – including G2, MPCGEN and VCL32 – and was noted to produce the highest rates of obfuscation compared to other kits. A similarity metric was used to compare assembly programs and no similarity was found to G2 and MPCGEN, up to 2.4% was found with VCL32, and normal files had similarities between 0.98% and 1.2%. In [283] only a 10% similarity was found between NGVCK when run over multiple

iterations, meaning the kit produces a large amount of variability between uses. An example of two virus variations produced by the NGVCK generation kit is shown in Table 2.10. Obfuscation produces two semantically similar variants using garbage code insertion, instruction substitution and subroutine reordering as techniques.

Table 2.10: Variations in code obfuscation used by the Next Generation Virus Generation Kit. Adapted from [298].

Before Obfuscation	After Obfuscation (Version 1)	After Obfuscation (Version 2)

call Function A
Function A: pop ebp
sub ebp, OFFSET Function A
	call Function A
	Function A: sub dword ptr[esp], OFFSET Function A
	pop eax
	mov ebp, eax
		add ecx, 0021751B; junk
		call Function A
		Function A: sub dword ptr[esp], OFFSET Function A
		sub ebx, 00000909; junk
		mov edx, [esp]
		xchg ecx, eax; junk
		add esp, 00000004
		and ecx, 00005E44; junk
		xchg edx, ebp

A more recent polymorphic engine was introduced in [81] as the Virus and Metamorphic Worm (MWOR) generation kit. The effectiveness of the generation kit was exemplified in [282] for being able to fool common statistical analysis. The kit has also found more recent interest in research as it is able to control for the proportion of garbage code and subroutine reordering possible [294, 295, 285, 298, 282, 283]. This is extremely effective because inserting a certain amount of garbage code from Benign files has demonstrated an improved ability to thwart AV scanners [50]. This chapter does not provide an exhaustive list of generation kits, on the contrary, these kits represent a small subset of available kits widely distributed. Websites such as VxHeavens was one of such sources until the website was taken down in March 2012 by Ukrainian police. Repositories containing over 200+ generation kits once hosted on VxHeavens can be found circulating online to this day. Included in these kits as discussed is anti-armoring and anti-emulation capabilities. Some of these will be discussed in the next section.

2.4.3 Anti-Emulation, Stealth and Code Protection

Anti-Emulation is an all-encompassing term that includes all the various armoring, stealth and/or code protection techniques that are used to thwart or burden the process of reverse engineering of a Malware sample. According to Symantec approximately 28% of Malware are VM aware [23]. One of the shortcomings of virtual machines and other honeypot deployments is that the environment they are deployed in is static, with several configurations set to default. It is for this reason anti-emulation Malware can check the environment for indicators of virtualization and fail to execute or burden the reverse engineering analysis with cumbersome instructions. This section will cover some of actions taken by anti-emulation Malware to exploit their virtual environment and prevent security experts from understanding the full breadth of their behavior. Anti-emulation checks fall into four categories: Human Interaction, Configuration-specific, Environment-specific and VMware specific checks [301, 302].

Human Interaction

checks to see if actions routinely carried out by a user is being performed. This includes mouse-movements, use of the clipboard and opening and closing windows. The Cuckoo Sandbox for example has a setting which provides this sort of functionality for each Malware submission. Trojan Upclicker is a virus variant that monitors user input in the form of a left click in order to identify sandbox environments. It does this by using the `SetWindowsHookEx()` and `GetLastInputInfor()` API to determine the rate of user input over time. This would identify the presence of sandbox environments as automated analysis does not require the use of an auxiliary keyboard and mouse [303].

Configuration-specific

uses time periods or other configuration to execute at a later time and date only if certain conditions are met. The Duqu virus, which was first identified in 2011, included a series of anti-stealth techniques in the form of delays as a precautionary measure [304]. Code injection only occurs after approximately 10-15 minutes, and the lifespan of Duqu is set by an unknown communications module that removes its hooks, deletes its kernel driver and removes its registry key once the timer has elapsed [305, 304]. The Kelihos botnet and Nap Trojan both make use of the `SleepEx()` and `NtDelayExecution()` for extended sleep calls; with the Kelihos botnet having affected 41,000 users before being identified and taken down. Hastati has a hard-coded check where it executes only at 2pm on March 20, 2013. Otherwise, it doesn't execute if `GetLocalTime()` returns a time less than that - indicating the presence of a virtualized environment [306].

Environment-specific

looks at the settings and parameters of the host operating system and hardware, and decides whether to execute based on those findings [307]. Virtual machines incorporate virtual hardware which tend to have consistent configurations between VM deployments. Hardware such as Network adapters, USB controllers and Audio Adapters are all virtualized; meaning MAC addresses, USB controller types and SCSI device types are all telling signs of virtualization. The *Scoopy Doo* tool developed by Tobias Klein uses Windows Script Host to read registry keys located in `HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\` and `HKEY_LOCAL_MACHINE\SYSTEM \ControlSet001 \Control\Class` associated with SCSI; and can also lookup keys that are associated with IO and ports for strings containing "VMware". In another application Malware can utilize the internal processor tick counter via the ReaD Time Stamp Counter (RDTSC) instruction. Based on a random bit value that is returned, the decrytor contained within the Malware will decode and execute the virus body, otherwise it will bypass and exit.

VMware-specific

uses checks that add the ability for Malware to look for specific indicators of virtualization based on the VMware software used by the host. One of the best examples is in the use of VMWare workstation's WinXP Guest virtual hardware which includes a running *VMtools* service and 300 references to *VMtools* in the Registry. Another interesting adoption of VMware behavior is Pushdo. Pushdo uses `PspCreateProcessNotify()` to deregister sandbox routines [302, 16]. It also performs a check of the physical hard drive serial number, and checks if it is set to a default value of `00`

which is typically in virtual machines. In the work of [308] the authors looked at anti-emulator behavior in Android Malware and noted volume identifiers, network interfaces and invoking the GPU were all techniques used to obfuscate Dalvik Virtual Machines. Other evasion techniques, such exception process timing, IMEI checking and checking the variability in sensors have all been traced to emulation evasion in Android Malware [309, 310, 311, 312, 313, 314].

Alongside the specific checks mentioned above, general anti-debugging makes it difficult for researchers to extract signatures or strings to develop systems to protect against them. An example is the Bistro virus which inserts garbage code insertion and dummy loops before the decryptor stub. As a result, before the Malware is even unpacked millions of instructions burdens the emulator and Bistro fails to run. During analysis many Malware variants are memory resident, thereby requiring careful monitoring of viral payload to load itself into memory before it can be dumped and analyzed [73]. In the past Malware authors have been one step ahead in their efforts to thwart monitoring memory dumps or memory snapshotting. An example is the Zmorph virus which has its decryptor rebuild its instructions line by line by pushing the result into stack memory. One of the earlier adopters of this sort of technique was the DOS/DarkParanoid which contained 10 different encryption functions which it used to encrypt its previously run instructions while only allowing its current instruction to be decrypted at any point in time. Without a conventional decryption loop, it is a true polymorphic memory resident variant. The use of other so-called "Stealth Viruses" employed reconnaissance of the OS by waiting until AV products check-summed programs to check for changes. When a file was read, as opposed to executed as is the case with user input, it took that as indication of check-summing by the AV and removed itself from the target executable. Finally, once it waited until the file was closed, it then reinfects the file [315]. Using this process it can follow the AV and infect every file on disk. A thorough summary of anti-disassembly, anti-debugging and anti-emulation techniques can be found in [54]. For a summary of android application hardening used by malware authors and developers, we refer the readers to the work of [316].

2.5 Approaches to Feature Analysis

Malware features are typically categorized into two types: static and dynamic. Static features incorporate all the unique compositional information of the executable, irrespective of the contextual information of the target system [317, 318, 319, 320]. That is to say, the static features of an executable would be the same regardless of what machine the Malware is deployed on. Static features typically include portable executable (PE) structure, assembly code instructions [321], list of DLLs, n-grams, and byte sequences. PE structure features would include information related to PE sections, resources, application programming interface (API) calls, as well as which dynamic link libraries (DLL) are imported/exported. Most modern Anti-Virus (AV) products employ the use of a signature database which contains known signatures of the static features of Malware. Alternatively, dynamic features include API and DLL call graphs; information gathered from the file system, registry, as well as process and thread activity and the consumption of kernel resources. Dynamic Analysis can also include temporal snapshots of process execution, memory, network, and system call logs [322]. Dynamic analysis is OS specific because depending on the system resources, account privileges and other environmental variables, the Malware will behave different and have a different signature as a result.

The ability for Malware to mutate has also presented a problem for researchers, which render many of the legacy static approaches to Malware research obsolete. As a result, dynamic analysis has been presented as the de-factor standard in classification approaches as it is impervious to routine obfuscation and packing carried out my mutating Malware [10]. Nowadays dynamic analysis represents some 51% of the analysis methods in the body of literature examined [318], with a unique combination of feature sets and model architectures being used to perform classification. It has been noted that Malware classification is not a trivial problem, with some presenting it as an NP-complete problem [75] to identify a bounded-length mutating virus, or a polymorphic variant of one [323]. Characterizing Malware is the fundamental issue of concern, and researchers and practitioners are constantly refining their methods to stay ahead of the curve. Figure 2.10 provides an illustration of the feature pipeline used for most Malware classification approaches. Both static and dynamic features form the bedrock in the characterization of malicious behavior. Any number of these features can be combined to form a hybridized model for feature analysis, which is unofficially the third form of characterization.

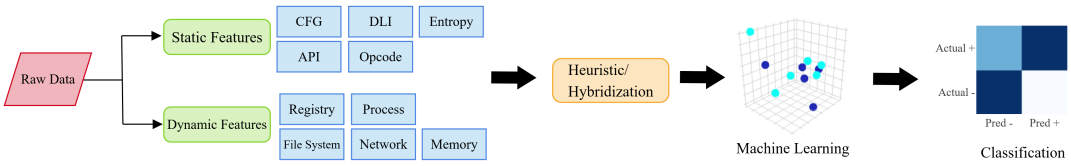


Figure 2.10: Summary of feature pipeline for the classification of Malware. Retrieved from personal work in [60].

Many of these methods are covered in the comprehensive review of [322] and [320], but this work will simply provide a narrow overview of Malware detection approaches as it concerns API calls. While API calls are just of one of many forms of static and dynamic behavior, it is one of the most consequential and information rich sources of discrimination. But first, an introduction to the source of APIs, files called known dynamically linked libraries, is required and will be the topic of the next section.

2.5.1 Dynamically Linked Libraries

Dynamically Linked Libraries, or DLLs, are libraries of code that are written by vendors such as Microsoft as well as third parties to coordinate and manage resources on the Windows OS. DLLs are fundamentally libraries of code that contain one or more functions, indicated in their Export Address Table (EAT), which identifies which functions are available for export to other processes. DLLs are structurally equivalent to executables, with the exception being that their main function is called `DllMain` and they cannot be executed without the use of a helper functions `RUNDLL.exe` or `RUNDLL32.exe`, for 64-bit and 32-bit, respectively. DLLs are useful because they allow multiple processes to share the same library of code loaded into memory, thereby reducing the time required to recompile each process and the amount of memory overhead if the same code segments had to be loaded in memory multiple times. Because each process does not need to include static code of its functions, its keeps file sizes smaller overall when it can connect to an already running copy of the library of functions. It also has the advantage of allowing the OS vendor to update a catalogue of core DLL libraries, which can work with subsequent versions of the OS.

When a DLL is requested to be loaded by an EXE, it does so through by checking some default directories first. There is a known registry key in `KnownDLLs` that tells Windows that the well-known DLLs should be found in the `System32` path, otherwise it searches in the .exe directory, the current working directory, the `%SystemRoot%` directory, the 16-bit `System` path, and then the directories in your environment `PATH`. DLL order hijacking is the process by which malicious actors inject their own malicious DLLs somewhere in this load order so that their payload is loaded instead of a legitimate DLL. For example, `ntshrui.dll` is loaded by `explorer.exe`, but it is not a know DLL and therefore can be susceptible to load-order highjacking. DLLs that are fully protected can recursively load other DLLs that are not protected, which forces the next executable to follow the default search order and be prone to highjacking. The tool Dependency Walker[6] can be used to see the dependency tree between loaded DLLs on the OS. Legacy Malware would change the Import Address Table (IAT) to point to a new address in memory for the DLL it needs. Changing pointers to new malicious address locations with malicious payloads has since been rectified on newer versions of Windows as it becomes apparent if all the address locations for functions are in higher memory space `0x7C86` and a single function is loaded into `0x3420` then most likely that IAT entry has been changed with a hook by a Rootkit. Alternatively, Malware can just modify the DLL in-line, requiring no changes in pointers just the code – leading to a vulnerability commonly known as DLL Proxying which is much harder to detect but can be alerted to using integrity checking.

Potentially vulnerable DLLs can be observed if using tools such as SysInternals' Process Monitor (Procmon[7]). In Procmon if a DLL is not found and it is not core to the functionality of the process, it will return an entry `NAME NOT FOUND`. Using an out-of-the-box option like Metasploit's[8] msfvenom will produce a DLL than can be put in place of the missing DLL, thereby running the malicious payload and executing a successful DLL highjacking. Other tools such as the SANS[9] tool can be used to search for DLLs that appear multiple times, are unsinged and in unusual folders. More common in research, the Dependency Walker tool[10] makes it easy to view the mapping of imported DLLs, and to even view a hierarchical view of all dependencies between modules by looking at the IAT. The authors in citewang2008 separated DLL usage according to Implicit Dependency, Delay-load Dependency and Forward Dependency, which are all responsibly for the static loading of DLLs in 3 tiers of hierarchy. Tier 1 starts from those used by the main program, followed by Tier 2 which are DLLs invoked by other DLLs that are not in the main executable, with Tier 3 being the entire statically loaded tree. The authors created a one-hot encoded vector if the particular DLL existed in the program and used that feature mapping for classification. In [324] a similar approach was taken which relied on the DLL dependency tree but incorporated encoding tree string dependencies. The authors looked at all the tiers of DLLs loaded and created a depth-first representation where the original executable is the root node and all nodes from root to leaf are assigned a unique integer value. They then used CMTreeMiner which extracts closed frequent subtrees that exist in a particular executable, and one-hot encoded a feature vector if a particular subtree exists in the executable. Looking at depths of subtrees from 3-6, accuracies as high as 98%+ were obtained following Random Forest and Naive Bayes classifiers. The work of [325] did not go in as depth as [324], but the authors looked at the number of API calls by a DLL in addition to the list of DLLs

[6] https://www.dependencywalker.com/
[7] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon
[8] https://www.metasploit.com/
[9] https://www.sans.org/blog/detecting-dll-hijacking-on-windows/
[10] https://www.dependencywalker.com/

used and the API calls made. In any case, while DLLs do provide a good proxy of malicious intent, it is in fact the API calls that are made that are the real discriminator. For this reason, researchers turn their focus towards API calls and their usage among Malware variants.

2.5.2 Windows Application Programming Interface

Windows API calls are interfaces provided by DLLs to access low-level resources [326]. API calls come in two flavors: User level and Kernel level APIs. User level APIs operate at Ring 3 and provide the average user just enough privileges to access system resources to perform typical workloads. The actual hardware on the other hand runs in kernel mode, which makes use of Kernel level APIs that are not directly available to users for the sake of security and stability of the OS. From the stability perspective, a user-level crash results in an error message, while a kernel-level crash results in the OS crashing. From the security side, Malware could reside in the kernel and operate at a layer that is indistinguishable to the user or any Ring 3 defenses. Nowadays it is much more unlikely to see Malware residing in the kernel, as the Windows OS has made it more difficult to run code in the kernel and make use of rootkits. Ultimately, to make use of the kernel all userland code uses `Kernel32.dll` as a gateway to communicate with `Ntdll.dll` which, in turn, communicates with the kernel.

The fascination with API calls comes down to the fact that API calls provides a higher resolution of analysis of the operation of any given process. It is the case that API functions and system calls are related to the services provided by the OS [322, 327]. As the API is responsible for all system resource management, it is a particularly discriminating feature for Malware classification as it provides the basic functionality for everything from networking to saving files to disk. The usage of APIs and patterns in usage can be very telling. Similar to the overarching view of Static and Dynamic analysis of behavior, APIs are approached from a Static and Dynamic perspective as well. In Dynamic Analysis the run-time behavior is monitored and, ideally, all code segments are traced to reveal the behavior of the Malware. This circumvents the obfuscation techniques of encryption, packing and polymorphism [328, 10]. Static analysis on the other hand can be fooled by adding fake API calls [329] or API calls typical of benign event activity [330]. It is also the case, as mentioned in Section 2.5.1, that the imported functions of a DLL may or may not ever be called - which can be used as a distraction from the real nefarious purpose of the Malware.

Features such as the API call function names, parameters, and the return values of an executable can be extracted from the APIs [331]. Monitoring the API calls is an approach to detecting the malicious behavior of software; however, there is no clear distinction between malicious APIs and benign APIs as all Native APIs are a helpful utility given the right context. The next section will outline some of the nefarious usages of APIs by Malware authors, and how they balance stealthiness with functionality.

Malicious Windows Application Programming Interface Usage

Broadly speaking API usage can be categorized into 7 categories based on the functionality they provide to a process [332, 327]. Researchers have also made use of similar categories to classify malicious intent [198]. Some of the malicious functionality APIs can provide to executables include the following:

File – create a file in sensitive folders; delete or hide files; file directory traversal

Process - inject DLL into a running system process; create mutex to prevent execution

Memory - free up or occupy memory; minimize memory usage

Registry – add or delete system service. Autorun, hide and protect.

Network – open and listen on a port, communicate over email service, communicate with CnC server

Windows Service – terminate windows update, firewall, setup Telnet or SSH

Other – hooking keyboard, hiding window, scan for existing vulnerabilities and configuration

Code injection usually begins with the usage of third-part DLLs, or injecting code into a Windows DLL. Malware makes use of `Ntdll.exe` indirectly to make use of kernel APIs, so checking the stack trace of event activity is important [2]. Malware authors have to balance gaining increased functionality at the cost of rising suspicion, so a careful deliberation of which APIs to use is always in mind [333]. Native Windows API calls that begin with `NTtQuery` are popular for Malware, as they include functions such as `NTtQuerySystemInformation` and `NTtQueryInformationProcess` which provide much more information about the host system. More invasively, early rootkits would make changes to the System Service Descriptor Table (SSDT) which contains addresses to the kernel functions, which would instead be changed to malicious driver functions. If, for example, a typical address of a kernel function is set to `804d7000` for `ntoskrnl.exe` then one can look at addresses which aren't familiar and contained within the address space typical for kernel drivers. With x64 bit versions of Windows starting with XP, *PathGuard* prevents modification of the kernel and the kernel code in the SSDT and the Interrupt Descriptor Table (IDT). The IDT takes care of exception handling, so re-routing the response to interrupts to malicious code would be highly disruptive. As a precaution to prevent making changes to native Microsoft DLLs and APIs, Windows Vista was the first Windows version to introduce digitally signed drivers. Some of the example use-cases and APIs used by Malware are the following:

File: If software wishes to make use of the file register, it can do so using `CreateFile`, `ReadFile` and `WriteFile`. Malware can make use of `CreateFileMapping` or `MapViewOfFile` which loads the file into RAM, avoiding writing to disk all-together. Some Malware types, like Ransomware, perform high volume file and encryption operations to carry out it's function [334].

Process: It is typical for Malware to use `OpenMutex` to check if a mutex exists for a running Malware executable. Malware can make use of DLL injection or Direct Injection. Code can be injected into a running process using `VirtualAlloxEx` and `WriteProcessMemory`. When code is injected into an executable such as `Explorer.exe`, the same privileges hold for the executable it is injected into. Asynchronous Procedure Call (APC) is a process by which malicious code is attached to the APC queue of a process' thread. `WaitForSingleObjectEx` is the most common call, with `QueueUserAPC` being used for queues running on a thread. Can be run from the kernel using `KeInitializeApc` and `KeInsertQueueApc`. APC remains a known vulnerability on the MITRE ATTCK knowledge base [335].

Registry: When it comes to making use of the Windows Registry, Malware can gain *persistence* so that it can load whenever Windows restarts [336, 328]. Most commonly the Run key located in `HKLM\Software\Microsoft\Windows\CurrentVersion\Run` can set executables to run automatically. The Sysinternals tool *Autoruns*[11] can be used to check dozens of registry locations, drivers loaded into the kernel, and any other DLLs. Other options for persistence include running *Services* which are typically more powerful than Administrator privileges. Other registry entries include *AppInit_DLLs*, which is a registry key that contains DLLs that are attached to processes that load `User32.dll`. This option has can be disabled in Windows 8 and later versions when Secure Boot is enabled. *WinLogon Notify* launches during log on, sleep, or when the lockscreen is open. Adding a malicious DLL to the ServiceDll parameter in the registry allows a malicious service to start its malicious service DLL into a loaded `svchost.exe` [337].

Networking: Certain network API usage can be indicative of malicious intent as networking APIs provide different levels of flexible. For example, the APIs in `Wininet.dll` will use higher level APIs for HTTP and HTTPS communications. Malware might use the raw Winsock libraries located in `ws2_32.dll` if there is a need to provide further flexibility to their malicious arsenal. The Metasploit framework can produce shellcode that acts as a listener on a port by creating a simple process using `CreateProcess`. The configuration for `STARTUPINFO` is set to a socket - thereby creating a remote shell. This setup allows for I/O and error handling for `cmd.exe` and does so with the command window suppressed to remain stealthy.

Other: Malware downloaders and launchers use `URLDownloadtoFileA` to download a file from a URL then executes the file by making a call to `WinExec`. Keyloggers use hooking or polling. Hooking uses an API such as `SetWindowsHookEx` to notify of a key press, while polling is done using `GetAsyncKeyState` and `GetForegroundWindow` to poll key states during any time period.

Researcher have looked beyond individual API calls and have investigated API call distribution [338]. A summary of some of these classes of API usage used by researchers is shown in Table 2.11. The issues arise in that it requires significant domain expertise to create and update a database of API calls for particular Malware variants or families. It is also the case that there is significant overlap between malicious and benign API usage, thereby making it difficult to alert to Malware without alerting to False Positives. The work of [339] developed a similarity metric to trace the similarity between Malware variants and Stuxnet based on groups of API calls. It comes to reason that groups of API calls in succession, or the distribution of API calls, can provide further insight into malicious behavior [340]. For this we investigate some of these research methods in the following section.

Table 2.11: Summary of Malicious API usage by behavior type. Retrieved from personal work in [60].

Behavior	APIs	Ref.

[11] https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns

General Behavior	`ShowWindow, GetWindow, WriteFile, WinExec, ShellExecuteA, OpenProcess, VirtualAlloc, *Hook, *Exception, *Shutdown, *Crypt, *Debugger, *Shellexecute, *Manager`	[341, 342, 339, 12]
Stealthiness	`NtDelayExecution, FindFirstFileA, FindNextFileA, GetProcAddress, LoadLibraryA, OpenProcess, Sleep`	[321, 339]
Kernel	`*Ldr*, *Section*, *DuplicateObject*, *Make*, *Object*, *Resource*, *UdiCreate*`	[12]
Memory	`*Memory*, *Volume*, *Space*, *Buffer*`	[12]
Registry	`CreateKey, OpenKey, CloseKey, RegOpenKey* RegSetValue, RegQueryValue, *EnumKey, *DeleteKey, *SetKey, *Enum*`	[12, 343, 328]
Reproduction	`*FindFirstFile, *CopyFile, GetFileType, SetFilePointer`	[343, 328]
DLL injection	`SetWindowsHookEx, CallNextHookEx, CreateRemoteThread, OpenProcess, LoadLibrary, GetProcAddress, VirtualAllocEx, WriteProcessMemory`	[321]
Search Files	`FindClose, FindFirstFile, FindFirstFileEx, FindFirstFileName, TransactedW, FindFirstFileNameW, FindFirstFileTransacted, FindFirstStream, TransactedW, FindFirstStreamW, FindNextFile, FindNextFileNameW, FindNextStreamW, SearchPath`	[338, 12]
Copy/Delete Files	`CloseHandle, CopyFile, CopyFileEx, CopyFileTransacted, CreateFile, CreateFileTransacted CreateHardLink, CreateHardLink, Transacted, CreateSymbolicLink, CreateSymbolic, LinkTransacted, DeleteFile, DeleteFileTransacted`	[338]
Get File Information	`GetBinaryType, GetCompressed, FileSize, GetCompressedFile GetFileInformation, ByHandleEx, GetFileSize, GetFileSizeEx GetFileType, GetFinalPathName, ByHandle, GetFullPathName, GetFullPathName Transacted, GetLongPathName, GetLongPathName, Transacted, GetShortPathName, GetTempFileName, GetTempPath SizeTransacted, GetFileAttributes, GetFileAttributesEx, GetFileAttributes, Transacted, GetFileBandwidth, Reservation, GetFileInformation, ByHandle`	[338]

Move Files	`MoveFile, MoveFileEx, MoveFileTransacted, MoveFileWithProgress`	[338]
Read/Write Files	`OpenFile, OpenFileById, ReOpenFile, ReplaceFile, WriteFile, CreateFile, CloseHandle`	[338]
Change File Attributes	`SetFileApisToANSI, SetFileApisToOEM, SetFileAttributes, SetFileAttributesTransacted, SetFileBandwidthReservation, SetFileInformationByHandle, SetFileShortName, SetFileValidData`	[338]
Metamorphic Engines	`HeapAlloc, LocalFree, HeapCreate, GetStartupInfoA, GetCommandLineA, GetEnvironmentStringsW, FreeEnvironmentStringsW, GetModuleFileNameA, GetCurrentProcess, CloseServiceHandle, GetCurrentProcessId, GetProcessHeap, HeapReAlloc, SetFilePointer, SetFileAttributesA, GetFileAttributesW, FindFirstFileA, FindClose, SetThreadPriority, GetCurrentThreadId, GetProcAddress, GetModuleHandleA, ResumeThread, GetEnvironmentVariableA, ExitThread`	[287, 339]
G2	`GetCurrentProcessId, GetConsoleMode, SetConsoleMode, FileTimeToDosDateTime, CreateFileW, GetFileSize, FileTimeToLocalFileTime, GetFileTime, LocalFileTimeToFileTime, SetFileTime, SetFilePointer, SetFileAttributesW, GetFileAttributesW, GetKeyState, ConsoleMenuControl, AppendMenuW, ReleaseMutex, FindFirstFileA, FindClose, SetThreadPriority, GetCurrentThreadId, GetProcAddress, GetModuleHandleA, ResumeThread, GetSystemTimeAsFileTime, GetTickCount, QueryPerformanceCounter, InitializeCriticalSection, LoadStringA, FormatMessageA`	[344]

MPCGEN	`HeapAlloc, LocalFree, GetVersionExA, HeapCreate, GetStartupInfoA, SetHandleCount, GetCommandLineA, GetEnvironmentStringsW, FreeEnvironmentStringsW, GetACP, GetCPInfo, GetStringTypeW, GetModuleFileNameA, LCMapStringW, MultiByteToWideChar, WideCharToMultiByte, GetEnvironmentStrings, LocalFileTimeToFileTime, SetFileTime, ReadProcessMemory, AppendMenuW, GetLastError, GetSystemTimeAsFileTime, GetTickCount, QueryPerformanceCounter, InitializeCriticalSection, FormatMessageA, GetCurrentProcess, DuplicateHandle, GetConsoleMode`	[344]
NGVCK	`GetClassLongW, CreateFontIndirectW, DeleteCriticalSection, TlsFree, UnmapViewOfFile, CloseHandle, GetCurrentProcessId, EnumDesktopsW, EnumDesktopWindows, CloseDesktop, GetProcessHeap, SetUnhandledExceptionFilter, OpenDesktopW, GetProcessWindowStation, GetUserDefaultLCID, CombineRgn, OffsetRgn, ExtCreateRegion, CreateRectRgnIndirect, SetWindowRgn, DefWindowProcW, PeekMessageW, SetCapture, SendMessageW, ReleaseCapture, MsgWaitForMultipleObjectsEx, PtInRect, GetRgnBox, HeapReAlloc, LCMapStringW`	[344]
Stuxnet	`LoadLibraryW, LoadLibraryA, GetModuleHandle, GetProcAddress, VirtualAlloc, VirtualFree`	[345, 339]

Classification of Windows Application Programming Interfaces

The investigation of API calls in the context of feature extraction is sometimes referred to as API call sequence or API call traces. In either definition we are concerned with the patterns that arise in the sequence of API calls used one-after-another. Early adopters of this form of investigation used Hofmeyr API call sequences, whereby behavior profiles were established between two sequences of API calls based on Hamming distance [346]. Originally, UNIX system calls were traced, and the investigators were motivated by the immune system in their attempt to draw an analogy between sequences of system calls and chains of amino acids in the human body. API call sequences have been leveraged in several applications involving Malware detection [347, 348, 198, 174, 349, 328, 350]; as well as in tracing the API call traces during event activity [328, 351, 352, 353, 354]. Overall, API call frequency and API sequences are effective techniques in identifying data-flow dependencies in a process [327]

Application Programming Interface Frequency

One of the more primitive approaches to API analysis is API frequency analysis. It stands to reason that if Malware and Benignware make use of similar API libraries, then Malware must make use of certain libraries or "malicious" APIs more frequently than others. In [331] considering API frequency alone was effective in achieving 97% accuracy in a multi-categorical classification problem involving metamorphic Malware variants. One takeaway was that incorporating sequential information did improve accuracy of the models, so frequency analysis is certainly a useful preliminary step in behavioral analysis. The work of [355] developed an end-to-end malware detector based on frequency of occurrence of opcode and API calls. Their detector coined OPEM, demonstrating increased area-under-the-cure (AUC) and lower FPs with static calls and a hybrid approach. Unfortunately, the authors didn't account for obfuscated Malware which tend to be packed and have polymorphic engines which obfuscates the opcode. Their hybrid approach, which included API execution trace, did outperform all other features sets used in their work [355]. Certain works, like that of [259], decided to use a frequency of a subset of 794 APIs calls extracted from 500 thousand Malware samples. The authors then fused this feature set with other static techniques such as Entropy and features extracted from the PE file such as the total number of assembly instructions in the `.data` and `.rsrc` section. The drawback to these approaches is that taking the most frequent API calls leaves out information of potential edges cases; and it is also a fact that frequented API calls by Malware are still routine events carried out by Benignware, such as reserving memory, creating a file, etc. The work of [356] approached the problem in a similar fashion, where they eliminated API calls with low frequency. Again, doing so removes important edge-cases, and is used typically to reduce the size of the feature vector space to improve training times. These aforementioned works all made use of ML techniques to classify their Malicious behavior. Other works make use of statistical similarity metrics to differentiate malicious versus benign by using one or more metrics of comparison. For example, in [316] the authors made use of information gain to select the features based on the sequence of opcodes from Android applications. Based on some key obfuscation techniques discussed thus far, including Control Flow obfuscation, string encryption, in addition to advanced techniques such as Class Encryption ad Reflection, the authors found several ML approaches were effective in detecting obfuscated samples.

In [357] *Cosine-Similarity* was proposed to compare API call frequency between two vectors to represent the similarity in vector space of a known signature to a new Malware sample. The expression for Cosine-Similarity is shown in Eq. 2.1. The motivation for using Cosine-Similarity is that the measure computes the similarity between two vectors while excluding their magnitude. This has the effect of ignoring the impact of magnitude if one vector were to use an API much more frequently than the other, as the θ angle in Eq. 2.1 is indifferent to their magnitude.

$$\cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \tag{2.1}$$

The *Extended Jaccard Measure* is another similarity metric than is useful in measuring the degree of overlap in two sets [357]. As an extension to the Jaccard for use in continuous or count attributes, it is effective in demonstrating the similarity, or the ratio of *set intersection*, between two sets in the context of set theory. The equation for this relationship is shown in Eq. 2.2. The numerator can be seen as expressing the set intersection, while the denominator as the union which acts as a form of

normalization.

$$J(x, y) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|^2 \|\mathbf{y}\|^2 - \mathbf{x} \cdot \mathbf{y}} \tag{2.2}$$

Cosine similarity was used effectively to create a similarity matrix between the rarest 20-30% raw security events and events of the training set [174]. This approach was used to significantly reduce their dimensionality of their set by focusing their efforts on the similarities between a baseline set of unusual events and their dataset more broadly. In [358] similarity metrics were computed for API sequences that appear frequently, and both assembly instructions and API calls were considered in their work. API calls were noted to be faster have a smaller signature; however, the authors noted the API approach is bad for network applications like PuTTY and encrypted files which show few or do not show any API calls. Their work did rely on unpacked executables as it was limited only to static analysis. In [357] an API call frequency similarity measure was used followed by a Chi-square test to test the representation based on a distribution from a known signature. Families of APIs of known metamorphic mutation engines were categorized and compared to one another and to the same mutation engine using both Cosine-Similarity and extended Jaccard Measure. An interesting finding was that comparing a similarity metric between variants from the same mutation engine provided a measure of the degree of *obfuscation* – which was shown to be the largest for the Next Generation Virus Creation Kit (NGVCK) a well-known mutation engine [10]. The work of [287] completed similar work, whereby a proximity index table was setup to compare the similarities between mutation engine families. Due to the sheer number of possible API calls, feature dimensionality reduction was carried out on the original 1000 or so APIs according to frequency. The authors noted common APIs were used between Mass Code Generator (MPCGEN) and NGVCK generated viruses. An approach that included data mining was taken in [332], whereby the calling frequencies of the raw features are calculated to select a subset of features, and then Principal Component Analysis (PCA) is used for dimensionality reduction of the selected features. In total 24,662 API function calls, 792 DLL features, along with PE header info were considered in their feature set while considering only the top 30 DLLs according to frequency [332]. To address the issue with high-dimensional data, [347] developed a string-based malware detection system that focused on the top 3,000 interpretable strings that included API names using a Max-Relevance Algorithm. Their feature parser extracted strings from 9,838 executables and classified them as Backdoors, Spyware, Trojans and Worms, in addition to Benignware. While these techniques have proven useful in many controlled scenarios, frequency-based analysis is still prone to Malware which can obfuscate themselves to avoid heuristic detection. For this reason, sequence analysis is used.

Application Programming Interface Sequences

The investigation of API sequences has become the de-facto standard for many behavioral approaches as the information contained within sequences is too powerful to rely on API frequency alone. It has also led to the adoption of natural language approaches which will be discussed in Section 2.5.4. The work of [328] provided an example of the flow of information surrounding a process can act as a template for how to carry out sequence analysis of APIs. The three flow paths are as follows:

1. the API call `GetModuleFileName` takes a `NULL` character as its first argument which returns the Malware file path

(a) the path can be passed to `CopyFile` to open the executable and run its processes

(b) or, if desired, a process can call `CopyFile` on itself with the share permission shared to `NULL`, thereby preventing applications from opening and scanning the file

This example serves to demonstrate that two very different uses of `CopyFile` can indicate malicious behavior, and only once the whole context is understood can a detection system alert to it. An application that performed this successfully was in [348] where 2,727 unique APIs were categorized into 26 groups based on functionality such as hooking, file and directories, registry modification, etc. Based on the sequence of the APIs, critical patterns were uncovered which were essential for core functionality such as screen capturing and DLL injection. Results demonstrated F1 scores as high as 0.999 with a focus on the longest common subsequence between existing malicious signatures and those of unknown variants. A similar approach was taken in [12] where 11 hand-crafted signatures of dynamic and static behaviors were created based on malicious operations spanning registry operations to device operation to kernel operations. These signatures were Converted to semantic blocks based on the largest common subsequences between dynamic and static APIs. The work of [359] created a formulation that includes API sequences as a part of a temporal domain, and pointers passed to API calls as spatial information. The motivation being similar to [328] in that an API call such as `LocalAlloc` takes in `uBytes` as an argument that is statistically lower for malicious files than benign files during allocation of the heap. Capturing this information in the spatial domain, while modeling the sequences of APIs in the temporal domain were effective in classifying 516 executables with accuracies as high as 0.966. Rather than focus on API sequences as it pertains to general malicious behavior, researcher have explored common API sequence usage among Malware variants and types. In [342] five classes of malware including Worm, Trojan-Downloader, Trojan-Spy, Trojan-Dropper and Backdoor were associated based on the presence of 26 API categories and sequences. 534 Malware variants were hooked and then categorized based on the presence of these API sequences, which were characteristically different for different Malware types that aim to pursue different objectives through their API usage. In [360] the authors considered 9 behaviors based on sequences of 2-4 APIs in succession, while [327] looked at combinations of 3 APIs (such as `CreateFile`, `WriteFile` and `CloseHandle`). The work of [361] obtained a 99.7% detection rate using several API calls sets, which included sequences of different lengths.

When it comes to determining appropriate sets of API calls for classification, researchers have pursued approaches in the data mining space to optimize for a set of association patterns towards a particular objective [362] – in this case, optimizing an objective that a sample belongs to a malicious or benign sample. Several papers have been published in this area, in particular those published out of the Xiamen University [363, 364, 365] focused on Malware classification. Ultimately, regardless of the particular mining algorithm used the idea is to find a set of API calls that support the objective of classifying Malware from Benignware. In [364] this was done using a Frequency Pattern Growth algorithm [366]. The goal is to create a Frequency Pattern tree which encodes sequence in a tree-like structure similar to a Huffman Coding where parents of a node are encoded as longer extensions of the child sequences. So, for a given API call `API_i` it would exist as a leaf node, while its parent nodes would contain sequences that contain `API_i` such as (`API_i`, `API_j`) or (`API_i`, `API_k`). This is done recursive up the tree, and frequencies are stored as satellite information at each node and this is how rules are generated. A new sample is then matched against the rules according to the descending order of the rules' confidence and support [367]. The motivation is to maximize the

likelihood that rules exist which can discriminate one objective from the other. This procedure was further described in [363] and used successfully to generate rules which parse 29,850 Windows PE files – half of which were malicious. In the approach of [367] the authors compared frequency mining approaches to ML approaches including SVM, Decision Trees and Naïve Bayes, and noted a 2-9% improvement in classification accuracy. Because these approaches did extract the APIs from the PE files, this static approach is not effective for packed Malware or APIs which are imported by the executable but never used. In a later paper by Ye, et al. [157] rule pruning was used for duplicate rules, and only elected to use the top 100 API calls as no further improvement was shown beyond 100. While using a Linear SVM, Associate classifier and novel Hierarchical associative classifier, 26 thousand malicious samples were parsed and a Precision as high as 96% was achieved, but with a low Recall of 34%. A thorough examination of the state of Data Mining approaches as it pertains to Cyber Security are covered in [368]. While handcrafting sequence signatures can be time-consuming and require knowledge of specific patterns in API usage, the alternative is to consider all possible subsequences of a given length and consider the usage patterns of all sequences simultaneously. While data mining does provide a compact representation to do this, more innovative works allows models to discern these rules on their own when coupled to ML approaches. For this purpose, n-gram representation is used.

Application Programming Interface n-grams

One of the earliest forms of sequence analysis in the Malware domain was carried out in [369]. It was also the first successful application of n-grams, which involves translating a sequence of L APIs into subsequences n long and doing so for every possible subsequence that exists in the original API sequence. This has the effect of incorporating information about the sequences of APIs with little preprocessing required. For any given API sequence, a sequence of length L would have $L - n + 1$ n-grams; where n is the length of the subsequences and assuming a stride length of one. So, for an API sequence 10 APIs long, we would have (10 - 5) – 1, subsequences for $n = 5$. The number of possible n-gram combinations would be $|C|^5$ - which represents all the unique combinations of five APIs in sequence that are possible in the set of APIs C. The authors in [369] looked at short byte string n-grams of the PC boot sector which was 512 bytes long. They utilized a ML approach that removed the sigmoid activation and stored the weights as 5/6-bit integers. The technique became part of the IBM AV package and was successfully deployed to millions of machines.

The versatility of n-grams means that one can look at smaller n to generate shorter signatures which are noisy but more generalizable; or use larger n to create more specific signatures which lead to lower False Positives (FP) but at a cost of lower True Positives (TP). The application of n-grams is known to have low FP rates with increasing sequence length L, however, the space complexity of n-gram sequencies is exponential in the length of the sequences $O(|C|^5)$ [83]. The work of [370] focused their attention of the PE header and body and carried out static analysis using the top 500 most common 4-grams [371] representing DLL names. Results demonstrated that the header-only features are as relevant as body information, and that separately they both have a use-case [370]. Similarly, in [372] a 4-gram representation was used to model API sequences. The authors developed average confidence values of benign and malicious activity and used the average confidence of Malware as a threshold. This simple thresholding obtained 90% accuracy however, the work provided no indication of FP rates to support their findings. The work of [353] went one

step further and carried out n-gram modeling of API call sequences based on File System, Network and Registry activity. This work was unique in that it separated API events based on File System, Network and Registry, to provide a further analysis of how these event categories fare in acting as discriminators. In all, the authors looked at over 17,900 malicious executables and obtained 92.5% test accuracy. Finally, [356] resorted to 3- and 4-gram representations but focused on the dynamic API usage after process execution. This resulted in 94% accuracy, but when coupled with static features sets based on frequency, improved the accuracy beyond 97%. The shortfall of n-grams is that sequences exceeding that of 4 or 5 are impractical to model due to the number of permutations of API calls, which significant hinders the ability for models to attend to different behaviors. For this reason, we can pursue graph-based approaches in an attempt to consider different behaviors simultaneously.

2.5.3 Graph-based Approaches

Graph-based approaches to Malware detection have a long history. The earliest application of graph-based include the use of Control Flow Graphs (CFG) to evaluate unique control flow sequences of a program. A CFG is created as a directed graph where the nodes represent individual or blocks of program instructions, and the edges represent the control flow between statements [322]. Within each CFG we have a sub-graph that is isomorphic to the whole graph. Trying to map a sub-graph from one sample to another is part of the set of problems which includes the Subgraph isomorphism problem which is NP-complete [373]. In Fig. 2.11 we can see an illustration for the control flow from the Trojan.Emotet virus. This instruction segment belongs to the set of instructions that are responsible for spawning a child process which depends on the initial call to `CreateEvent` at the top of Fig. 2.11. When examining such a control flow, the question becomes which segment(s) of instructions are responsible for malicious behavior. While this segment was carefully selected to show the behavior of Emotet, extracting similar segments from the entire malicious execution is cumbersome – especially when they include diversions and dead-ends. Extracting such segments as signatures and generalizing these signatures to flag future Malware samples is the goal of CFG-based Malware classification.

Most application of CFGs look at extracting some subset of the flow of sequences to compare to other samples to establish a baseline for malicious control flow. One approach used by [374] looked at `jmp, jcc, call, ret, inst` and `ret` opcode instructions, and built the CFG based on only these instructions; thereby creating a reduced graph and leaving placeholders for the rest. Based on these, the authors created unique signatures for malware detection. In [375] the authors looked at the system call functions, which included call, jump and conditional jump expressions in the x86 Intel instruction set. In [376] the authors looked at the most frequent subgraphs and simply excluded the rest. The sample set used by [377] included 25,145 functions which were 5 nodes (simple instructions) large, and 15,439 unique functions which were 5 nodes long. Setting the threshold at 5 ensures that only atypical calls and procedures are included. One of the issues associated with CFGs is that the control flow is either (a) similar among all executables, regardless of malicious activity (also known as boilerplate code) or (b) is sometimes appended with benign code segments that aren't ever executed but can confuse string-based scanning techniques [377]. This was considered by [378] in their CFG reconstruction based on system call logs extracted using *Procmon*. Their approach did not look at functions that were not loaded by the dynamic linker in order to remove boilerplate

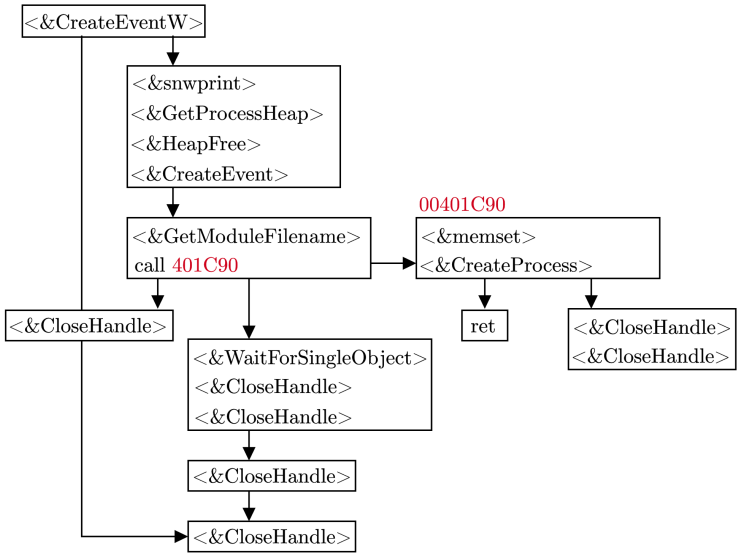


Figure 2.11: A CFG representation of the disassembled instructions for Trojan.Emotet produced in Ghidra. Retrieved from personal work in [60].

code. However, this is a double-edged sword as Malware doesn't only rely on its Import Address Table (IAT) to fetch the APIs it needs, it can load those statically as well. An alternative approach used in [379] looked at contrast subgraphing [380], which is the opposite of graph isomorphism since it looks for the smallest subgraph of G_1 that does not belong in G_2. This approach lends itself well to looking for characteristically significant differences between Malware and Benignware; rather than developing signatures that look for similarities among classes. Alternatively, one can consider creating signatures as co-opcode graphs that belong to Malware families, and therefore, create high-level signatures that can be used to classify Malware families based on co-opcode graph similarity [331]. While opcodes have been investigated extensively, Windows API usage has been shown to perform well at detecting polymorphic variants, [157, 375, 174] but the large size of potential sub-graphs remains a limitation to graph-based approaches. Going more in depth, [381] examined not just the API functions used, but also their function input arguments between File System, Registry, Socket and Process Operations. This provides additional insight into the calling process, such as through bytes written to when using `WriteFile` or destination key when setting a registry value using `RegSetValue`. The work of [300] looked at opcode similarity to detect polymorphic variants. The authors developed a weighted directed graph where the edges were probabilities that one opcode followed the next. They then computed scores between metamorphic viruses, between viruses and benign files, and developed a threshold score for maliciousness. This approach performed well since metamorphic viruses are created with a select few metamorphic engines; therefore, the signatures developed are in fact tracing obfuscation used by a given mutation engine [375, 382].

Another factor to consider when using CFGs is how to establish a comparison between CFGs from malicious and non-malicious control-flows. The authors in [373] examined the detection of metamorphic code based on a cross-comparison of the control flow graphs of known Malware. The authors normalized the code to remove dead or unreachable code, removed common subexpressions, removing dead paths, and analyzed indirect control flow transitions to remove longer chains of control

flow and avoid misdirections. The authors recorded a 96.5% True Positive rate while producing almost no False Positives. The Jaccard similarity matrix, was used in [378] between system call subsequences. Cosine similarity is another approach used [383]; but all similarity metrics suffer from drawbacks because they are all subject to the selection of sub-graph as discussed earlier. Even with reliable sub-graphs that perform well on a particular set of Malware, the work of [384] demonstrated that 23 algorithmic graph features including betweenness centrality, closeness, degree centrality, density, number of edges and nodes, can be used in adversarial analysis and result in a 100% misclassification rate. Their approached target IoT Malware, but Android Malware is also an ongoing field of study [385, 386, 387]. With all the shortcomings that come with the graph-isomorphism problem, newer advances in this field remove the need for graphs all-together and convert the entire graph into feature vectors [388, 384]. Once features are vectorized, this opens up the door for other machine learning models to act as discriminators for the classification step.

2.5.4 Natural Language Processing Approaches

The use of Natural Language Processing (NLP) approaches applied to API call sequences was a natural extension to developing models that can predict malicious behavior. Malicious behavior is not simply a product of individual API usage or frequency of APIs, but rather, a consideration of the pattern in the API usage over time. Similar to how word usage and context can provide an indication of whether or not an email is spam or not, the context of API called in succession can tell you something about malicious intent. This has the effect of being able to attend to different behaviors simultaneously and allows the model to learn what malicious behaviors exist on its own.

Many popularized vectorization techniques used in NLP applications have also been migrated for the purpose of Malware research. Two of these techniques were displayed in the work of [389] which used a Bag-of-Words (BoW) model and term frequency-inverse document frequency (tf-idf). The background specifics of these techniques will be discussed in the next section. Their work created fixed lengthed vectors from behavioral reports produced in virtual machines and automated the feature extraction step. Finally, an ensemble of ML techniques, such as Random Forest, k-Nearest Neighbors (k-NN), Support Vector machine (SVM) and XGBoost were used, with majority voting summarizing the end predictions over the models. An application that did involve APIs was carried in [12] who looked at both dynamic and static behaviors and hand-crafted groups of signatures based on operation. The authors created 11 different types of malicious operations, spanning from registry operations to device I/O to kernel operations. APIs were converted to semantic blocks which look at the largest common subsequences between dynamic and static behavior. Following the sequencing, tf-idf was used to vectorize the contribution of each API, with a focus on rarely used APIs that drive malicious behavior. In [174] tf-idf was used to convert the sequence of a unique event name to a representation for a machine learning mode to learn which included both 1-dimensional Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) architectures. A similar line of work was used in [390] where a LSTM was used to model sequential API usage of 20 thousand Malware samples run on a Windows 7 machine using the Cuckoo sandbox. The authors only considered 342 API calls but limited their investigation to those that were used at least 10 times among all samples in the training set. When coupled with tf-idf, this has the effect of focusing more on rarely used APIs, and by limiting the minimum threshold to 10, there are enough training examples for the model to learn the importance of those features.

In addition to the form of vectorization, modern NLP models allow the model itself to learn the importance of each word (or API) relative to the context of the surrounding words. For this purpose, word embeddings were developed which can learn the semantic relationship between words and map that relationship to vector space [391]. This has the effect of allowing models that are closely related to have similar cosine-similarity scores. A modest application by [392] used 300-dimensional word embeddings followed by a similarity matrix to cluster Malware and Benignware using k-means. This way, the cluster index was a dense representation of Malware and Benignware. A more end-to-end approach was used in [3] whereby API stack traces were modeled as an NLP problem. Embedding dimensions of size 50 to 200 were used to map the API stack trace that included APIs that communicated all the way to the kernel. With the use of a Transformer architecture which learns latent representation of the sequences, F1 scores as high as 96.2% were obtained when considering Registry APIs. The authors in [393] looked at developing a semantic transition matrix to segregate API calls which have similar contexts into clusters. This was done by capturing the relationship between API calls that represent Malware and Benignware using Word2Vec [391], a word embedding technique which has more powerful encoding ability than vanilla word embedding approaches. More powerful encoders translate to better ability to learn context, which was evident in their FP rate of only 1%. A similar use of Word2Vec was followed by a LSTM in [394] to analyze opcodes and API function names. In total 1369 API function names and opcodes were used, of which 958 were API calls.

Several works have made use of Windows PE Malware API sequence dataset [390] a dataset of API call sequences extracted from 7017 malicious binaries from 8 Malware classes including Adware, Backdoor, Downloader, Dropper, Spyware, Trojan, Virus and Worm. For this dataset, [395] achieved poor results with an 0.38 F1 score when using a 32-dimensional embedding to represent the API sequences followed by a 2-layer LSTM. Their approach used 342 API calls and discarded those that were used less than 10 times. Similar poor results were obtained in [396] which reported F1 scores ranging from 0.33-0.72 for the 8 Malware types based on a similar LSTM approach. The work of [397] went one step further and compared a LSTM approach to that of a Tranformer, and finally to a bi-directional encoder representation from transformers (BERT). BERT relies on learning latent representations from both directional contexts from before and after sequences, meaning it does a better job encoding context of the API sequence. In [397] they also used the Windows PE Malware dataset, and found similar issues classifying the 8 classes with a weighted F1 score of 0.51 on their best performing BERT model. One approach that did find success using BERT was that of [398] who implemented *fastText* [399], a text vectorizing technique based on n-gram. While removing redundant API calls, such as `NtDelayExecution`, accuracies as high as 96.76% using BERT were obtained.

Language models are an attractive alternative because they model API calls as if they were natural language, where the context of one sentence (or API call sequence) matters relative to the surrounding sentences. However, process behavior does not necessarily operate in such a sequential manner, where one API call sequence necessarily follows from the next. In Section 2.5.3 it was noted that behavior can sometimes branch out as the control flow is passed on from instruction pointer to the next. From a process perspective, parent processes can spawn daughter processes, which in-turn, each have their own behavioural profiles. This graph-like interpretation of process behaviour can be further examined using the tools available to us in the form of graph models. The next section will

briefly describe graph-neural networks, as well as their application towards security.

2.6 Introduction to Graph Neural Networks

Graph Neural Networks (GNN) represents one of the newest advancements in the DL space, with applications spanning protein design [400], botnet detection [401], traffic flow forecasting [402] and text classification tasks [294]. The use of GNNs has spanned many applications where learning the attributes of entities (formulated as nodes) and their relations (formulated as edges) can be used to make a prediction about another node, edge or the entire graph itself. Illustrated in Fig. 2.12 we observe a graph whereby nodes u and v are encoded into a d-dimensional embedding space, where ideally these vectors would be situated closer together if they have similar properties, and farther apart if they do not.

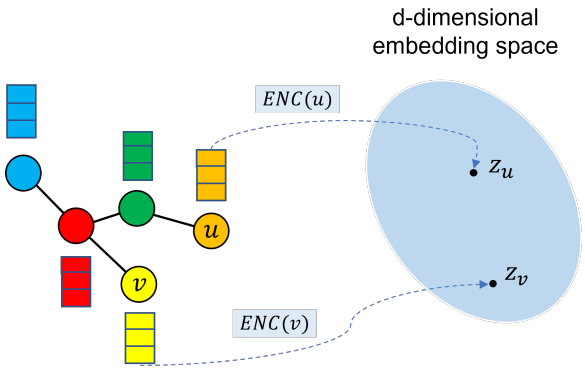


Figure 2.12: Graph nodes being encoded into embedding space via encoding function $ENC(\cdot)$.

Mathematically this can be expressed using the dot product between two vectors $\mathbf{z_u^T z_v}$. The question is how can we share the knowledge of nearby nodes, and pass the information along the network so that all nodes incorporate the nearby attributes of its neighbours? Both message passing and network aggregation are two of the foundational components which allow graph models to aggregate and pass on information to other nodes in the network. Message passing involves creating and compartmentalizing the message h_u for node u which can be carried out by computing the dot product $\mathbf{W}h_u$; where $\mathbf{W}$ is the learnable weight matrix, $W \in \mathbb{R}^d$ for a d dimensional embedding space, and h_u is the node feature vector. This weight matrix is shared among all nodes in the network, therefore, the message passing is treated equally across the network. The network aggregation step is also straightforward and involves one of many forms of concatenation of the messages of nearby nodes, with Eq. 2.3 serving as one example using a weighted mean.

$$h_v^l = \sigma\left(W^l \sum_{u \in \mathcal{N}(v)} \frac{h_u^{l-1}}{|\mathcal{N}(v)|}\right) \tag{2.3}$$

The summation will iterate over the neighbourhood nodes of node v, where $N(v)$ is the set of neighborhood nodes. The term $\frac{1}{|\mathcal{N}(v))|}$ in the summation is a normalization to prevent the contribution of high node degree nodes from exploding, and l is added to denote the layer of the GNN; where the numbers of layers is increased to incorporate node messages from multiple hops away from the target node. Finally, $\sigma(\cdot)$ is used to denote any form of non-linearity to inject expressiveness

into the node embeddings. The normalization $\frac{1}{|N(v))|}$ is treated equally across the model, and the model does not distinguish node connections which are perhaps more valuable than others. For this reason, graph attention is added (Eq. 2.4) which allows the model to learn the importance of nearby nodes. The normalization constant in Eq. 2.3 is removed and replaced with α_{vu} which provides a factor for computing the importance of neighboring nodes $\mathcal{N}(v)$ to the node of interest v. Due to the additional *attention* that is placed on the network passing, these networks are coined Graph Attention Networks (GAT) [4]. More on the application of GATs and inner workings is described in Section 3.4.2. For a thorough examination of GNN best practices and techniques, we refer readers to the work of [401].

$$h_v^l = \sigma\left(W^l \sum_{u \in N(v)} \alpha_{vu} h_u^{l-1}\right) \tag{2.4}$$

The first two papers in the field of GNNs [403, 404] endorsed their application for directed, undirected, labelled, and cyclic graphs. The Graph Convolutional Network (GCN) is one of the earliest forms of GNN layers, that includes both a message passing layer to pass messages to neighboring nodes, and an aggregation function that aggregates the messages to create a new node embedding. In the context of security, GNNs have been used in networking applications to optimize wireless routing xu2021 and in network traffic prediction he2022 with great success. An issue arises when the size of the neighborhood $N(v)$ varies for each node, it becomes difficult to maintain the local invariance of GNNs [401, 405]. This can be easily resolved using the normalization factor shown in Eq. 2.3, or by applying ℓ_2 normalization to ensure the embedding vectors have the same scale. This can be done using $h_v \leftarrow \frac{h_v}{\|h_v\|_2} \forall v \in V$ where $\|u_2\| = \sqrt{\sum_i u_i^2}$. At this point all vectors have the same ℓ_2 norm. As an example application of GNNs, the authors in [406] classified packed Malware among 6 Malware families. The function call graph was extracted, and the subgraph belong to the unpacking function was removed to get the local function calls. The graph contained the CFG based on the disassembly instructions; and based on this approach the authors recorded a 96.4% categorical accuracy. Similarly, [407] extracted the CFG on 20 thousand Malware samples using a deep convolutional GNN, and recorded F1 scores in excess of 0.9926 for 12 Malware families. Their proposed end-to-end defense system architecture, coined MAGIC, obtained attributes from the code sequence such as the number of compare instructions, `mov` instructions and numerical constants, as well as topology features such as degree of neighbors and the instructions themselves. Overall, the MAGIC system performed excellently, but scored less than an XGBoost [259] and ensemble of Random forest classifiers [261]. Both of which have significantly less parameters to be tuned and is a well understood classical ML technique.

2.6.1 Application of Graph Neural Networks in Anomaly Detection

GNNs have made significant advancements over the years, to the point where the application of GNNs and the techniques involved are in of itself a subspecialty of Deep Learning (DL) almost as much as computer vision or NLP. As a primer for anomaly detection in GNNs, many techniques make use of an *autoencoder* to learn a latent representation of the dataset. This autoencoder comprises an encoder module which transforms the input into a dense representation, and then a decoder which reconstructs the input back into it's original dimensionality. Transformers serve as one example of

this architecture [3] whereby the loss is calculated by how well the model guesses the next word in the sequence. Autoencoders are useful because we can learn the context of a given input sequence $\mathbf{x}$ through the reconstruction based on the output $\mathbf{x}'$. Autoencoders are trained based on how well they reconstruct the input, meaning their loss $\mathcal{L}(\mathbf{x}, \mathbf{x}')$ can be computed as differences in the input and output $\|\mathbf{x} - \mathbf{x}'\|_2^2$ or the root mean squared error (RMSE). In the application of graphs, this reconstruction can be reconstruction of the adjacency list through the use of a structural decoder, or an attributed encoder which simply reconstructs $\mathbf{x}'$ as mentioned previously. It is also common to combine both a structural and attributal autoencoder for modeling both anomalous structures of edge segments and the anomalies in the attributes themselves. One core advantage of autoencoders is that labels are not required, making them an attractive alternative to clustering in unsupervised or semi-supervised approaches. Identifying outliers in graphs presents it's own unique challenges. The complexity of graph relations require more powerful and expressive neural networks to converge to optimal solutions, which then correlate with increased run-time and memory requirements [408]. The introduction of Outlier Aware Networking Embedding resolved issues with the lack of homophily found in networks where outliers are prevalent and node structure and attributes are not consistent among the graph structure. This technique is more resilient to real life networks where outliers and noise is commonly found.

The advantage of autoencoders in the context of GNNs is that when reconstructing from the dense representation (i.e. the decoder) any deviation for a new instance of a problem from the expected output can be attributed as an *anomaly*. Therefore, this an advantageous property to have in the security domain to detect anomalies in graphs which can be correlated with malicious insiders or malicious intra-network communications. The work of [409] was an early adoption of the use of autoencoders to detects subtle anomalies in input sequences. In [410] an unsupervised model coined the ONE algorithm was created and was optimized by detected deviations in the structure and attributes of the network. This would later be used to detect outliers in node embeddings in downstream tasks. Anomaly detection can expand beyond the detection of single nodes and can be used to detect nodes which are outliers among communities of nodes, which are sometimes referred to as community-aware frameworks [411]. In [412] their DONE implementation used an autoencoder coupled with a distortion of the Node2vec embedding space to learn noise-aware outliers in attributed networks. This allows the model to generalize better to potential outliers by allowing for less strict conditions, such as nodes from one community having connections to another community of nodes. A shallow GNN using a deep autoencoder was used to spot reconstruction errors in [413]. A similar approach was taken in [414] who used dual autoencoders: one to spot structural changes and one to spot attribute changes. Coupled with an attention mechanism, they also developed a measure of the reconstruction errors to spot anomalies in three real-life benchmark datasets. The work of [415] implemented the exact same dual encoder approach with attention coined GUIDE, and noted improved scores on five datasets including Cora and Pubmed; two datasets which describe citation networks among research papers and authors. An interesting use of adversarial models was developed in [416] in their anomaly-aware model, whereby their encoder learns anomaly aware node representations. The generative adversarial network takes in generated noise anomalies and then plays a min-max game to train the network. The discriminator model becomes better at discriminating between real and perturbed node instances of the network and can therefore generalize well with a high degree of effectiveness. Another use-case in anomaly detection is towards the detection

of edges of a graph, which have an important application in detecting communications which may belong to malicious entities. The structural GNN model implemented in [417] investigated subgraph generation along with a temporal network to detect normal versus malicious edges. Their approach went one-step further and collected data from 109 hosts and executed 82 attacks from 9 different categories. Their approach garnered a 9-28% improvement in area-under-the-curve (AUC) results, displaying the effectiveness in solving real-world anomaly detection tasks in enterprise systems [417]. A recently published survey covering graph anomaly detection for both supervised and unsupervised applications can be found in the work of [418], with an a additional work covering benchmarking for outlier detection algorithms found in [408]. As an example application in a networking context, Fig. 2.13 serves to demonstrate how an autoencoder can be used to identify anomalous backbone network traffic. The workflow illustrated in Fig. 2.13 follows four main steps:

1. A graph autocoder is used to identify anomalous nodes in a graph. This is done by passing in unlabelled node topologies, and the model learns what is anomalous by subtracting the reconstruction error $\mathbf{x}'$ with the original topology $\mathbf{x}$. The weights are adjusted according to how well the input topology maps to the reconstruction. When nodes and edges scores highly in the reconstruction, they are more likely to be anomalous since they don't coincide with what the model has learned is baseline normal.

2. A compact graph substructure is generated which simply refers to a subset subgraph that includes highly scored anomalous nodes and the neighbors which helped to contribute to that score. Techniques such as GNNexplainer [419] are used for this purpose.

3. Given these highly anomalous substructures, extract taxonomy rules which refer to feature thresholds that can be used to identify these anomalous nodes in the original feature space. Techniques such as LIME [420, 421, 422] and SHAP values [423] are used. In this case, these rules can refer to firewall rules or any other sort of rules that are interoperable with a Security and Information Management System (SIEM).

4. Provided these taxonomy rules are established in the previous step, anomalous hosts and connections can be flagged earlier on, and a Network Security Operator - or someone with similar domain expertise - can establish safeguards and guardrails against anomalous network traffic.

2.6.2 Application of Complexity in Cyber Security

Self-similarity implies the object under consideration is self-replicating at different scales, and therefore, is scale invariant. This property of an object is important because it provides long-range dependency between successive measurement at different scales and resolutions. This degree of dependency can be extracted from the polyscalar relationship, and is known as the Fractal Dimension (FD). FD exists in many flavors related to polyscalar relationships based on variance, length, correlation, entropy, spectral, among others. For example, self-similarity can be used in data compression, where a large data set is represented by a smaller set of self-similar patterns. This can be used to reduce the storage and transmission of data. While in network science, self-similarity or fractality can be used to understand the organizational structure of a network quickly. This section

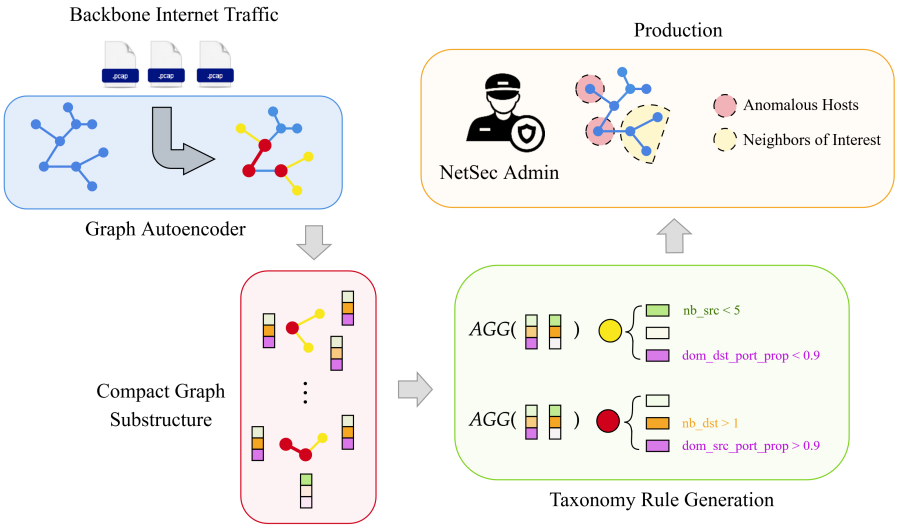


Figure 2.13: An autoencoder pipeline for the identification of anomalous backbone network traffic. Retrieved from an unpublished work by the author.

will only cover FD measures that are commonly used in applications in Cyber Security. A thorough explanation of the mathematical foundations of Complexity will be described in Section 3.5.

One of the more common applications of self-similarity in Cyber Security is in the detection of network intrusions. Researchers have used self-similarity measures, such as Hurst exponent, to identify abnormal behavior in network traffic, which can indicate the presence of an intrusion. The Hurst exponent, also known as the Hurst coefficient or the Hurst index, is a statistical measure used to quantify the degree of self-similarity or long-term memory in a time series [424]. The Hurst exponent is typically denoted by the symbol H. One way to calculate the Hurst exponent is through the use of Rescaled Range (R/S) analysis [425]. The R/S analysis involves dividing a time series into sub-series of increasing length and calculating the range (R) and the standard deviation (S) for each sub-series. The Hurst exponent is then calculated using the following equation:

$$H = \frac{\frac{1}{n}\sum_{i=1}^{n}(log(R_i) - log(S_i))}{log(n)} \tag{2.5}$$

Where n is the number of sub-series and R_i and S_i are the range and standard deviation of the ith sub-series, respectively. What we find in the literature is that the calculation for H is more readily carried out through detrended fluctuation analysis (DFA) which is done by calculating a least squares linear line to the cumulative sum of the data, and then calculating the slope of this line *alpha* and dividing by 2. DFA was successfully performed in the work of [426] via calculation of the Hurst exponent H. The authors developed a proportion metric based on SYN and ACK packets as their base signal, and then computed local fluctuations using a root-mean-squared (RMS) calculation to approximate H. This technique proved to be robust on the four DDoS related datasets used based on a custom adaptive threshold developed in their work. In [427] the authors implemented the Hurst exponent as a form of alert metric in an intrusion detection system via recording the types of events, *EventIDs*, and the users, *SID*. The motivation for their technique was that the event log would have a similar distribution of *EventIDs* for each *SID* in normal baseline behaviour, and that deviations

in these event records over snapshot time interval δ would be indicative of anomalous behaviour. It was noted that following three different types of attacks produced by the vulnerability scanner Tenable Nessus, self-similarity values decreased via monitoring of a vector of *EventIDs* and *SIDs* which estimates the self-similarity of the system as a whole.

The self similarity of network traffic is most likely the most well studied application of self-similarity in network traffic, and it has been known since the seminal paper of [428] that network traffic was fractal in nature [429]. Researchers aim to investigate whether the distribution of packets that is common over certain ports and IP addresses are found to change to any discernible degree in the event of a network intrusion or probing. The authors in [430] looked into this exact problem when they noted scanning probes carried out by worms led to a rapid change in network latency by sending a large number of TCP, RST or ICMP host-unreachable packets. While looking at port distribution of the traffic distribution of port, packet size and protocol they computed the self-similarity of their traffic attribute space and were able to detect early burstiness stage of worm propagation. In the works of [431, 432] wavelet analysis was performed initially before computing the Hurst exponent to detect DoS attacks. The authors noted the sensitivity of their techniques based on their analysis of a baseline threshold of 0.75, where values less than 0.5 corresponded with unusual network activity. In [432] they used an Isomap feature dimensionality reduction technique which carries out non-linear transformation that they claim is better able to enlarge the different between the Hurst parameters of normal versus DoS traffic. Seeing as how non-linear transformations would presumably destroy the property of self-similarity, it raises some questions about their approach.In [433] the R/S techniques was used to estimate the Hurst exponent as a part of their ATMSim package which leveraged Hadoop YARN to investigate a DoS attack scenario and an Address Resolution Protocol (ARP) spoofing scenario. Their technique showed a burstiness phenomenon when anomalous traffic occurred with correspondingly high self-similarity values. This differed significantly from the results obtained when normal traffic, such as LAN traffic. In network analysis researchers typically look at both packet bytes and packet count as an indication of the nature and behaviour of traffic. Packet count is useful in identifying patterns and trends in network traffic, while packet bytes can be useful in identifying patterns and trends in the amount of data being transmitted over a network. While looking at both simultaneously, [434] noted the self-similarity of certain types of anomalies are driven by markable differences in $H_{byt} \approxeq H_{pkt}$. They looked at byte and packet aggregation count over time, and noted those trends during different forms of DoS attacks. A more novel potential future application of self-similarity was used in [3] whereby the measurement of self-similarity, measured via the variance fractal dimension, was directly integrated into a Recurrent Neural Network (RNN) in order to improve predictive performance. The motivation for this work was that the long-range dependency can be used as a feature towards the future behaviour of a time series. This was performed on Brownian noise which has a known self-similarity, but future adoption may see the application extended for non-synthetic dataset scenarios. In Fig. 2.14 we see an illustration of Brownian noise (left y-axis) superimposed on its variance FD (right y-axis). Observe that the variance FD remains (relatively) constant even as the noise monotonically increases due to the invariance to scale of long-range dependency.

Another application of self-similarity in Cyber Security is in the detection of Malware. Researchers have used self-similarity measures, such as the Longest Common Subsequence (LCS), to identify similarities between different samples of Malware and detect new variants of known Mal-

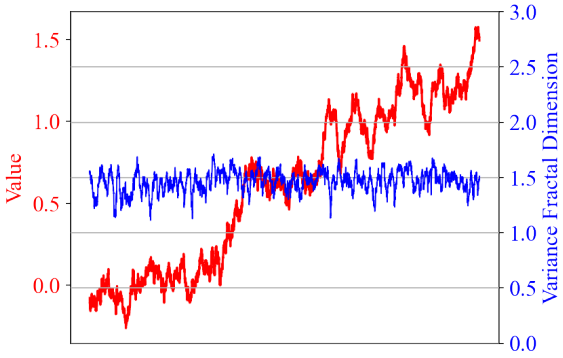


Figure 2.14: Brownian motion (left-axis) overlapped with it's variance fractal dimension (right-axis). Retrieved from personal work in [3].

ware. Similar sub-sequences of opcode or strings would be indicative of a common Malware family or functionality, while at a certain point differences would indicate a new Malware variant. In the series of works by Mirai et al. [435, 436] the LCS was determined for a series of Malware, and it was demonstrated to outperform analysis techniques which used API call sequences. Their approach used the familiar n-gram technique and compared matching similarities between binaries based on proportion of several common LCSs. The concept of LCS is fairly important when one is considering Metamorphic Malware as it provides an indicator for commonality between Malware families. While Metamorphic Malware uses techniques such as garbage code insertion and instructions re-ordering to obfuscate, ultimately, the fingerprint of the Malware remains in the binary; and can be readily identified via Entropy or self-similarity approaches. This intuitively makes sense as Metamorphic Malware carries the same payload and decoder stub each time is is propagated and obfuscated [10]. In terms of LCS, the Kolmogorov complexity measure of an LCS can be thought of as the amount of information required to describe the LCS in the context of the two or more sequences that it is being compared to. For example, if the LCS of two sequences is very short, it can be described with very little information and thus has a low Kolmogorov complexity. In terms of signature creation, this would have larger coverage of Malware variants but lead to more False Positives. On the contrary, if the LCS is very long and complex, it would require more information to describe, and therefore has a higher Kolmogorov complexity [437] and therefore low coverage but low False Positive rate. In one of the first applications of Complexity in a DL architecture, Brezinski and Ferens., [8] integrated Kolmogorov complexity via box counting to classify Malware samples into their respective families. This was done using the max-pooling layer and other innovative approaches to compute the FD directly within the CNN architecture, with an illustration of this technique shown in Fig. 2.15. While this was a proof-of-concept tested in 8500+ Malware binaries belonging to 25 families, the opportunities for Complexity in the field of DL is still to be seen and explored. The next section will set the stage for a new innovation of Complexity with the introduction of the Radius of Gyration and Mass Radius FD measures.

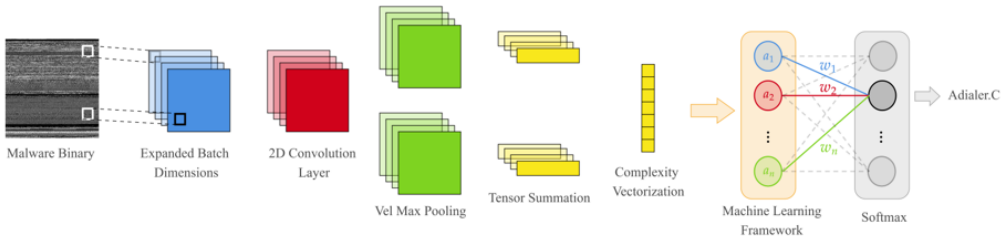


Figure 2.15: Convolutional Neural Network architecture which incorporates informational fractal dimension in its feedforward and backward pass for the classification of malicious binaries. Retrieved from personal work in [438].

Chapter 3

Methodology

This chapter sets the stage by for the core contribution of this work by presenting the methodology and techniques developed. First, in Section 3.1 an overview of the virtual environment and sandbox used in the collection of malicious event activity is presented, followed by a case study on a real malware variant known as *Emotet*. This section also discusses the malware dataset and some methodology as it relates to run-time configuration of the sandbox virtual machine snapshot. Next, an introduction to Graph Networks will follow in Section 3.4 which sets the foundation for presenting malware execution as a graphical topology. Finally, the theory behind the topological complexity in Section 3.5 will form the basis for the Complexity-based Graph Attention Network, or XGAN. Finally, graph benchmark datasets that were used to evaluate and compare the performance of these new methods against other techniques are discussed in Section 3.6.

3.1 Introduction to the Sandy Toolbox

3.1.1 Sandbox Infrastructure

The Sandy Toolbox is a set of tools developed to collect, prepare, organize and present malware process execution behaviour. This toolbox must be able to efficiency process large quantities of event activity such that the information can be reconstructed later on and presented in a way that is suitable for model training. This work includes a collection of packaged scripts which can be found on the official Github page for this package. This work has been published in [7] with a video explanation found here. First we begin with the preparation and initialization of the virtual machine on available hardware. The system architecture is built on the following hardware:

1. Dell PowerEdge R730 rackmount server machine
2. Intel Xeon E5, 10 Cores/20 Threads, 25MB Cache, 2.3GHz
3. 32 GB DDR4 Random Access Memory (RAM)
4. 36 TB SATA 5400rpm, RAID 0
5. Windows 2016 R2 Server

On the server hardware, Virtual Box[1] is installed which provides a hypervisor in which the virtual machines will host the various OS with the deployed malware instances. A graphical overview of the virtualization is shown in Fig. 3.1.

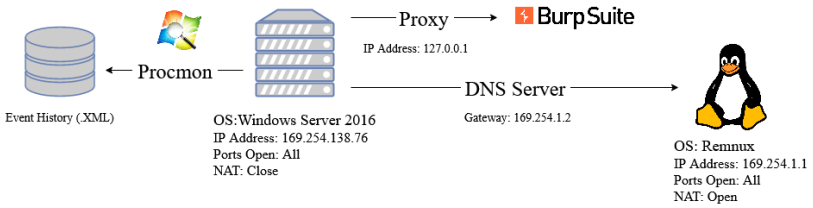


Figure 3.1: Sandbox configuration for use for malware execution.

The host OS is tasked with recording all activities, including executing malware while recording all Registry, File System, Network and Process and Thread activity. Several programs are run in order to track all ongoing behavior, as well as prevent the obfuscation of the malware's digital trail. These software including Procmon[2], Process Hacker[3], BurpeSuite[4] and Flypaper. BurpeSuite is installed on the host OS in order to act as an HTTP Proxy to intercept any raw activity passing to and from the host OS. This includes both HTTP and HTTPS protocols. A proxy is configured on the host OS to point to the localhost address `127.0.0.1` on port `8080`. Remnux[5] is setup as a DNS server on the same subnet as the malware host machine. In this work the subnet `169.254.0.0` is used for the LAN connection in Virtual Box. Two important services are run on the Remnux distro: `inetsim` and `dnsspoof`. These will be discussed in Section 3.1.1. Finally, the virtualization is hosted on a Windows Server 2016 SP1 machine. A shared folder is created between the server and malware host machine, and will be the means of sharing Procmon log files and other pertinent information. Each virtual machine is assigned 4GB of RAM, 32GB of fixed size storage, as well as a Network Address Translation (NAT) and Loopback Adapter. NAT is disabled on the host OS to prevent malware from establishing connections with outside CnC servers.

Windows Configuration

Several Windows OS were used as malware deployments. Among them include Windows 8.1, Windows Server 2016 and Windows 10. Windows 8.1 serves as an example of a deprecated OS which stopped receiving regular security updates from Microsoft in January 2023. This makes Windows 8.1 particularly vulnerable to zero-day exploits, as some end-users have not made the upgrade, either knowingly or unknowingly, and are therefore susceptible to unpatched security exploits. Additionally, Windows Server and Windows 10 were used as examples of more modern day OS.

Procmon (short for Process Monitor) is a windows diagnostic tool developed by Sysinternals. Mainly used as a diagnostic tool for system administration and application debugging, the software

[1] https://www.virtualbox.org/Virtual Box 6.0.2
[2] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon
[3] https://processhacker.sourceforge.io/
[4] https://portswigger.net/
[5] https://remnux.org/

doubles as a tool for carrying out forensic investigations. *Procmon* works by hooking functions and using event tracing. This is carried out via Standard Query Language Server Data Tools to monitor a registry and filter a driver into a file system. Procmon is able to capture information related to registry, file system, network and process and thread activity.

BurpSuite is used to capture HTTP and HTTPS requests. Each request logged is complete with a Host Name, Method (GET, POST, etc.), URL as well as the raw header in plain text and hexadecimal.

Process Hacker is a forensic tool that provides a live view of the processes currently running on the host operating system. This tool does not carry out any logging, but is useful in understanding which processes are spawned from other parent processes.

Flypaper is a forensic incident response tool that is able to monitor the behavior of malware by logging all the temporary files that are created and deleted when malware is deployed. This tool provides researchers the ability to track child processes and temporary files which are removed as a result of the malware attempting to cover their digital tracks.

Linux Configuration

The Linux environment is prepared to act as a simulated NAT router for which the traffic from the OS sandbox is captured and logged. The Linux environment is deployed with root privileges, and accepting all incoming IP addresses using the command `iptables -P INPUT ACCEPT`, and spoofing DNS requests using the command `sudo inetsim`, while producing simulated services over the *eth1* adapter using the command `sudo dnsspoof -i eth1`. These utilities carry out the following actions:

inetsim is a Linux tool for simulating common internet services. Internet services simulated include: HTTP/HTTPS, SMTP/SMTPS, POP3, DNS, FTP/FTPS, TFTP, IRC, among others. The most useful utility in *inetsim* is the ability to produce fake DNS responses in response to connection redirects, and redirection of TCP, UDP and ICMP connections.

dnspoof is a Linux tool which forges fake replies to DNS address/pointer queries. This will allow for the forging of responses from the true DNS with spoofed entries. This will produce a log of all DNS queries being made, and when they were made with an assigned timestamp.

3.2 Sandy Overview

3.2.1 Sandy Pipeline

The Sandy pipeline described the logical flow of information retrieval from process behavior, from initial data collection to the final step of malware classification. Along the way, various file formats and containers are used to modify and process the information into more easily readable and computationally efficient formats in order to interface with popular machine learning libraries. This introduces another core motivation with this work, which is that following each step the information

can be extracted and saved in various file formats for customized use. A brief description of each module is described as the following:

Data Collection includes the series of programs and scripts which facilitate the collection of raw activity. *Procmon* exports all stack traces and logged entries in `.XML` file format. Each **Tag** needs to be identified with its corresponding end **Tag**, and each **Element** needs to be extracted as unique column entries.

Preprocessing includes all forms of data organization, cleanup, and feature extraction. The *Sandy Toolbox* includes a separate sub-module `Sandy.pp` for parsing through entries, and converting the attributes into the proper data format for model development. Data can also be exported as *Pandas* dataframes for viewing and plotting.

Model Development allows for the development of machine learning and deep learning models based on the extracted features following preprocessing. The *Sandy Toolbox* makes use a popular deep learning learning framework called *Pytorch*. Models are created through inheritance from *torch.nn* and is optimized for use with torch dataloaders.

In Figure 3.2 a graphical representation is shown to indicate how each submodule is incorporated within the Sandy environment. The submodules provide extreme flexibility when it comes to working with a variety of datasets and preprocessing types. For example, in the *sandy.pp* module feature extraction is used to extract salient features from the datasets. In *sandy.graph* the entire history of the process tree can be viewed, and the extent of the `.DLLs` imported can be viewed as a time sequence graphically. Additionally, we can track a target executable and trace the spawn behavior of the child processes. Finally, *sandy.model* provides all the functionality to automatically encode features for model development. This includes one-hot encoding, embedding, normalization, etc. depending on the feature type.

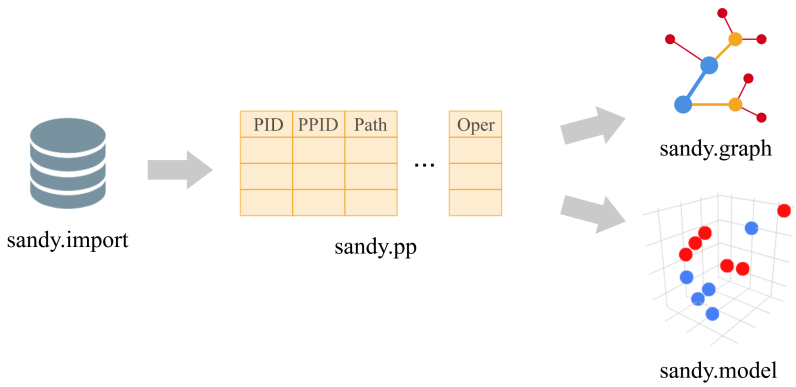


Figure 3.2: A graphical illustration of the core submodules contained in the *Sandy* environment.

3.2.2 Data Collection

Data collection begins when the windows snapshot is loaded in the VirtualBox platform. A batch file is run automatically using a event created in Windows Task Scheduler. The `.bat` file uses several

command line arguments to open, capture, wait, save and terminate the *Procmon* session when done. Once the `.XML` is exported, a series of import functions created in the *Sandy Toolbox* will import the data into the *Sandy* environment.

A series of helper functions facilitate the import from the `.XML` file via `XMLtoDict.py` and `XMLtoStr.py` for the *Python* environment. Two additional import methods handle the importing of the data to the *Sandy* environment, they are the `import_mod` and `import_event` methods from the *Sandy.import* module. The `import_mod` will handle the higher level process information such as process name, timestamp, command line arguments, etc. - while the `import_event` will import the event instances and the features associated with them. These features will be discussed in the following section.

3.2.3 Data Preprocessing

The data processing functionality of the *Sandy* toolbox is used to extract useful features from the *Procmon* logs, which can then be parsed using Regular Expression (Regex) and used for classification. Once the data is imported into the *Sandy* framework, the data is stored as a *.net* object. The three important categories of information traced are Processes, Modules and Events:

Processes : are simply the processes running on computer. Within the context of logging behavior, this includes process threads which are the basic units in which the OS allocates processor time. For example, a Process would be *Windows Explorer*, with a Process Identifier (PID) of 5604, Parent Process Identifier (PPID) of 13528 and Timestamp of 6/23/2020 5:56:17 AM.

Modules : include all the `.DLL` and `.EXE` files that are loaded by a process. Using the same example as above, *Windows Explorer* loads a total of 269 modules, one of which loads `misosh64.DLL`. This module has an address (`0x400000`), size (`0x10000`), and path (`E:\ProgramFiles(x86)\MagicISO\misosh64.DLL`).

Events : include the utilization of system resources and includes registry/network/file system and profiling activities. A registry event for the *Windows Explorer* process has a timestamp (6/23/2020 11:12:23.0910525 PM), thread (1200), class (registry), operation (RegSetValue), duration (0.0000395), path (`HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\FeatureUsage\TrayButtonClicked\ClockButton`), length (4) and data (37).

Further expanding on the event category, the *Sandy* sandbox can organize based on Registry, File System, Network, Process and Profiling events:

Registry : entries include creating, enumerating, querying and deleting keys and values; includes path to the hive name

File System : includes operations on local storage and remote file systems; includes path to drive name

Network : UDP and TCP activity; includes Source and Destination IP addresses

Process : includes process and thread events involved in the loading of executable images and data files in the memory

Profiling : for every process and thread event captures the kernel and user time changes, memory usage and context switches; captures CPU time charged to the process.

3.2.4 Malware Samples and Run-time Configuration

The malware samples used for execution were drawn from a repository of recent malware samples obtained from VirusTotal[6]. Through an Academic License, VirusTotal provides researchers a repository of 10's of thousands of malware samples identified and bundled in the last quarter year to represent new and unique infections submitted to VirusTotal. In this work samples were retrieved from Q2 2023. This dataset of malware samples was filtered to remove non-Windows malware, and includes both 32-bit and 64-bit executables. A full list of malicious executables used in this work can be found in Appendix D complete with file sizes and MD5 hashes.

In the sandbox environment the samples were automatically run through the use of a batrch script which helps to automate the process and provide consistent executions between malware samples in terms of time window. In Code Snippet 3.1 we see a short script that is used to run *Procmon* and load relevant configuration files and filters to begin collection. In this example, a process named "Malware.exe" is executed. The filter files (`.pmc` files) are used to exclude some Procmon specific events from appearing in the list of captured events, but all other events, including windows operating system behaviour, is captured. This ensures there is no bias introduced in the collected event activity by the author. A list of the rules used in the *Procmon* filter can be found in Table D.1 in Appendix D.

Code Listing 3.1: Batch script for automating the collection of malware process event activity.

```
@echo off

cd %USERPROFILE%\Desktop\Benign

for /f "delims=" %%i in ('dir /b') do set "processes=%processes% %%i"

set /a num_programs=%RANDOM% %% 3 + 3

for /f "delims=" %%i in ('echo %programs% ^| sort /R') do (
  set "selected_programs=!selected_programs! %%i"
  set /a count+=1
  if !count! == %num_programs% goto done
)

:done

for %%i in (%selected_processes%) do (
  start "" "%%i"
)

cd %USERPROFILE%\Desktop\Malware
```

[6] www.virustotal.com/

```
start Procmon64.exe /RunTime 3 /BackingFile test.pml /LoadConfig All.pmc /Quiet
timeout /t 8 /nobreak > NUL
start Malware.exe

for %%x in (File, Registry, Thread) do (
  start Procmon64.exe /LoadConfig %%x.pmc /OpenLog test.pml /SaveAs %%.xml
)
```

Alongside malware, benignware is executed in tandem as to simulate a real host environment. In Fig. 3.1 this is done initially before *Procmon* is ran in silent mode. During malware execution, 3 - 5 processes are randomly selected from the benignware list and run sequentially. This is to populate the execution graph with noise and negative training samples, and ensures the dataset mirrors the OS environment of a real host. In total, 300 benignware samples are collected from the `cnet.com` Apps for Windows category representing popular windows applications. All benignware was confirmed to have an AV score of 0 according to VirusTotal, and are tabulated in Appendix D with their file sizes and associated MD5 hashes.

First, consider the set of malware samples M and benignware samples C. If each malware executable were to be executed once, then M total trials or executable graphs are being populated with process activity. Let c_m represent the subset of benignware executables $c_m \subset C$ for execution m. At each trial m, the chance of drawing a particular benignware sample is $(1/|C|)^{E[p_c]}$ with replacement; where $E[p_c]$ is the expectation of drawing samples with probability p_c. Therefore for M trials the chances of not drawing a particular benign sample is governed by Eq. 3.1; where the $|C| - i$ term takes into account the fact that replacement can not occur as no executable can be executed twice on a clean snapshot. This exercise is simply to demonstrate the case that some benignware samples are not used to train the model based on the sampling technique used. This provides a good generalization of potential host environments, without introducing bias into the dataset if the processes were chosen manually from a small subset.

$$M \prod_{i=1}^{E[p_c]} \left(1 - \frac{1}{|C| - i}\right) \tag{3.1}$$

3.2.5 Malware Tracking

It is important that we are able to track the process that is our malware executable, trace the spawn behavior and events that result from its execution, and then be able to readily distinguish malicious from benign activity. The process for tracking the PID and PPID of the malware executable is carried out using the `create_target` method. A flow diagram in Fig. 3.3 is used to demonstrate how this is carried out. The process begins by assigning one target as a target label - which ideally would be the malware executable of interest. The PID is then extracted from the entry associated with the chosen target and stored in a buffer of PIDs. The *.net* object is then scanned for PPIDs that match entries in the target PID, which would correspond to processes which are spawned directly or indirectly from the PID of the malware executable. Once this is done the process repeats itself by adding the new PIDs of the spawned processes, then scanning the dataset for any further branched processes. Once no more PIDs are found and added to the buffer, the method returns the PIDs and process names of all the entries found.

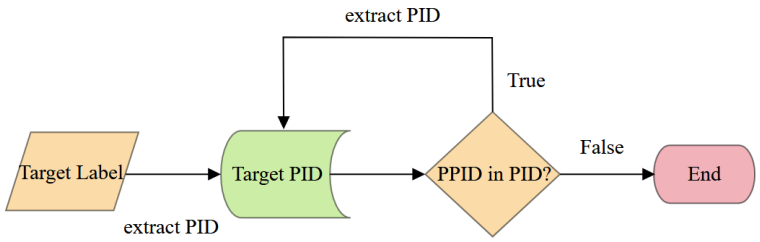


Figure 3.3: Flow diagram illustrating the process of extracting the Process identifier (PID) of the tracked process and all child processes using an iterative approach.

3.2.6 Data Visualization

In the *Sandy* framework the relationship between spawned processes can be interpreted as a directional flow graph. A directional flow graph $G(V, E)$ where the vertex set V is a collection of vertices that represent nodes on the directional graph. Successor nodes $\mathcal{N}^{+}(v)$ and predecessor nodes $\mathcal{N}^{-}(v)$ - where $v = \{0, 1, \ldots, V\}$ - represent spawned instances of either processes and sub-processes, respectively, according to a timestamp. The edge set E, where $E = \{(i, j) : i, j \in V, i \neq j\}$ for vertices i, j. Each vertex and edge contains a component set $E(v)$ stored as a dictionary of keys K and values V where $E(v) = \{Dict : K \rightarrow V\}$. In Fig. 3.4, vertices V_i and V_j and edge $E_{i,j}$ each have dictionaries stored in the *.net Sandy* object. This facilitates the retrieval of edge values using the notation: `net.V[i]['Timestamp']` for the i^{th} node with the dictionary key of interest K, which in this example is 'Timestamp'.

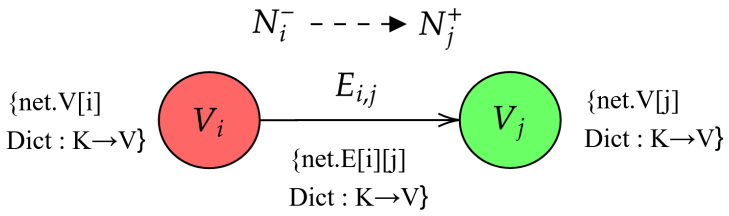


Figure 3.4: Directional flow graph relationship between neighboring vertices and the connecting edge used in the *Sandy* environment

This graph framework can be plotted using the `plot_network` method in *Sandy.graph.buildGraph*. For illustrative purposes we can assign one of the values retrieved from the *.net* object using the `assign_attribute` method as a feature to display on the graph. Fig. 3.5 an overview of the imported `.DLLs` are shown based on the process they were originally spawned from. The blue shading indicates the timestamp that was assigned to the loading into memory of the particular `.DLL`. From this vantage point we can appreciate the extent to which a process imports many `.DLLs` and the order in which they are imported during run-time. In the next section we will investigate the execution of malware and how we can investigate certain event logs to track malicious behavior.

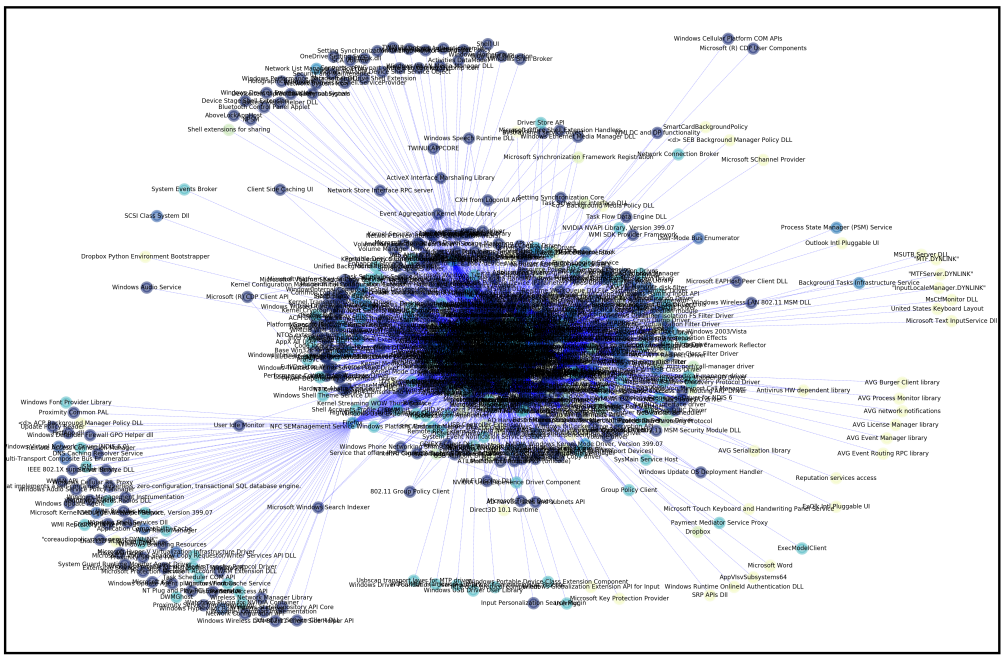


Figure 3.5: Tracking `DLL` imports over 30 seconds in the Windows 8 operating system. The color shade of nodes correspond to the relative timesteps in which `DLLs` are imported. Connections between nodes indicate the parent process which imported the associated `DLLs`.

Table 3.1: Summary of the proportion of Registry, File System, Network, Process and Profile events that contribute to the overall data collection for the **Baseline** operating system; after several **New Processes** are executed; and once both processes are running in the background along with the Windows operating system (**Baseline + Proc**)

	Baseline	**New Processes**	**Baseline + Proc**
Registry	91.99%	68.62%	90.02%
File System	3.63%	30.64%	5.51%
Network	0.04%	0.03%	0.01%
Process	0.11%	0.26%	0.12%
Profile	4.24%	0.46%	4.34%
Total Events	**114,350**	**1,265,133**	**119,659**

3.3 Case Study

With these descriptions we can look at an example of the captured events in a 30 second time window in the sandbox environment. In Table 3.1 three headings correspond to events captured during **Baseline** activity, during the execution of several **New Processes**, and the new baseline once the processes are running (**Baseline + Proc**). The main takeaway is that Registry events take up a large majority of the events during idling; however, once new processes are executed, loading of images make up 30.64% of the 1.2+ million events. Following execution, file system only makes up 5.51% of the remaining event activity. This indicates features could be present in the File System activity, making it a potential suitable feature class for classification.

Table 3.2: Summary of the proportion of Registry, File System, Network, Process and Profile events that contribute to the overall data collection for **Benign** activity (operating system processes and benignware) and **Malicious Activity** (Emotet).

	Benign	**Malicious**	**Proportion**
Registry	69,906	216	0.31%
File System	71,832	217	0.30%
Network	1614	0	0.00%
Process	1716	45	2.62%
Profile	1335	2	0.15%
Total Events	**146,403**	**480**	**0.33%**

Since our focus is on the immediate change in the OS environment following malware execution, we want to draw our focus towards the potential activity of a single malware executable. For this we look at event activity for execution of the Emotet malware. In Table 3.2 the numbers of unique event activities is compared for both benign and malicious activity. We can see that malicious event activity makes up a small proportion of total activity at any point in time, however, there is an appreciable surge in Process and Thread activity, accounting for 2.62% of all Process and Thread activity during malware execution. This may provide further insight into which events should be considered as general malicious behavior. The results in Table 3.2 also demonstrates another problem in anomaly detection: class-imbalance. With only a 0.33% presentation for all event activity, this represents a fundamental problem when attempting to distinguish small changes among millions of event activities that are occurring.

3.3.1 Emotet Malware

The Trojan.Emotet malware is a longstanding malicious piece of software that was first identified in 2016 as a banking Trojan. To-date Emotet is responsible for over 1 Billion USD in damages, with hundreds of CnC servers having been used for the basis of operation at any point in time. In 2019 Emotet was recorded as being the second most prevalent family of malware targeting businesses worldwide, with increased growth of 6% year over year. The primary attack vector for Emotet was through phishing attempts via email attachments. Users were enticed with email attachments claiming to have Edward Snowden's new book, or support for popular public figures such as Greta Thunberg. Once malicious `.doc` files are opened, macros launch a powershell script which installed the malware from parked Wordpress domains. Emotet was typically coupled with the installation of a secondary loader which would install Trickbot - which made use of the Eternal Blue exploit - or the Ryuk ransomware. Trickbot would operate by harvest network credential and brute forces networks, while Ryuk would ask for ransomware payments in return for decryption of the file system. In late 2020 Qbot was also used as a credential stealer installed as a secondary loader. While infection of any one of these malware variants is bad enough, Emotet marketed themselves as a Botnet-as-a-Service which enabled other malicious actors to purchase infected computers for hire [439]. In January 2021 joint investigators from several countries including the United Kingdom, Germany,

France, the United States, among others, took down the central CnC for Emotet located in Ukraine.

3.3.2 Emotet Event Characteristics

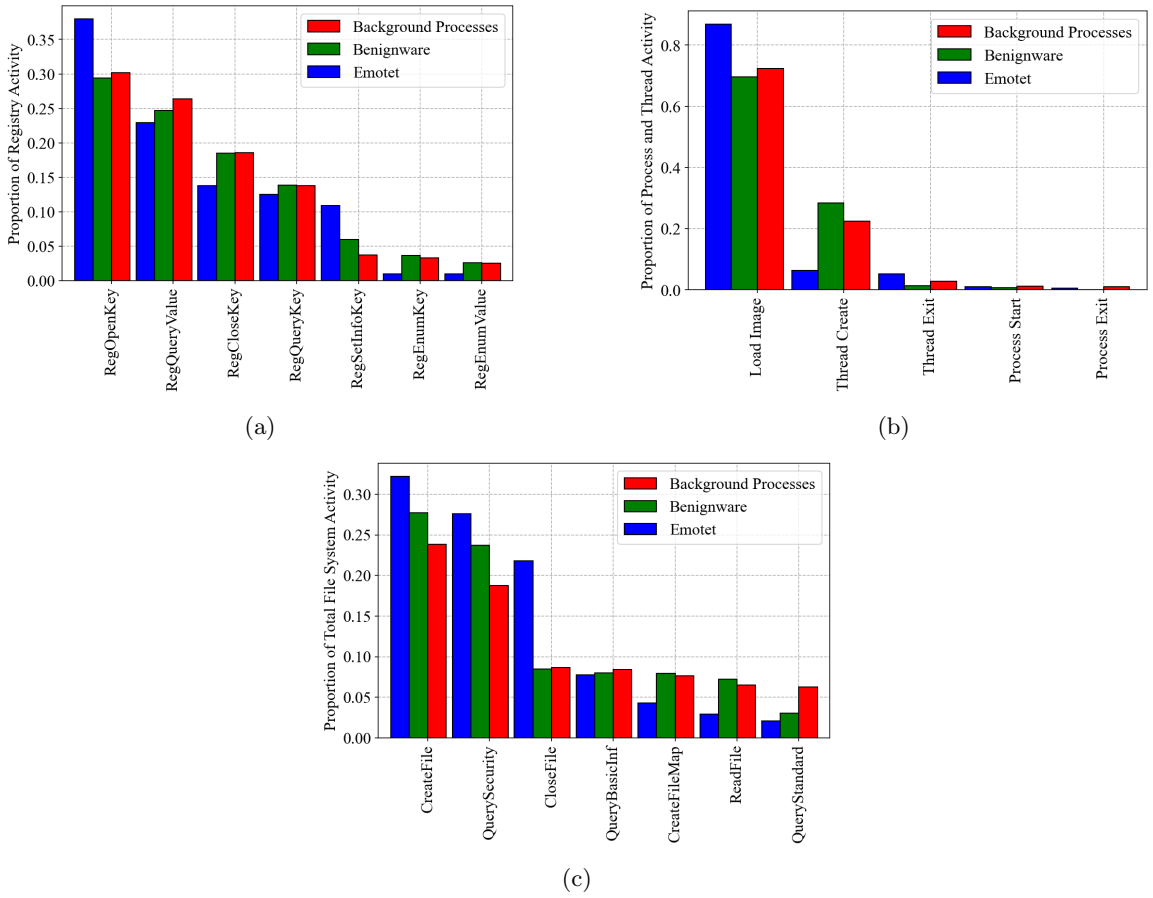


Figure 3.6: Proportion of total event activity belonging to the most prevalent Win32 API calls for (a) Registry; (b) Process and Thread; and (c) File System activity. Values are normalized based on the proportion of total event activity for that particular class.

In virtual environments many malware variants become Virutal Machine (VM)-aware and fail to execute their viral payloads. Therefore, the signatures of the malware which probe the current environment for signs of virtualization are as part of the malicious payload as the payload itself. Additionally, we wish to capture and detect anomalous behavior before secondary payloads are fetched. To examine this we executed the Emotet malware alongside other processes including Mozilla Firefox, Adobe Acrobat Pro and Remote Desktop (`mstsc.exe`). In Figure 3.6 the relative proportions of the most common API calls for Background Processes (i.e., OS) benignware (the aforementioned processes) and Emotet are shown. Network is not shown as the malware did not proceed to resolve and reach out to any domains. In Fig. 3.6a the Emotet malware made more calls to `RegOpenKey` and `RegSetInfoKey` than both benignware and background processes. `RegSetInfoKey` acts as a pseudonym for `NtSetInformationKey` that makes changes to the specified Registry key. Additionally `RegOpenKey` (or `RegOpenKeyExa`) opens the specific registry key. These both show evidence of probing behavior that is not found in benignware.

In Fig. 3.6b we can see the results for Process and Thread API calls. From the results Emotet is more likely to load itself into memory, and is much less likely to create a multi-threaded application (`ThreadCreate`). Additionally, for every thread that is created by Emotet the thread is exited (`ThreadExit`). For background and benignware which are continuing processes until closed, there are proportionally more `ThreadCreate` than `ThreadExit`. Since malware does not want to make use of system resources for extended periods of time, these function calls demonstrate this behavior is present and can be observed in the event logs. Finally, if we draw our attention to Fig. 3.6c we observe File System activity. Emotet is proportionally more likely to use `CloseFile` - at almost twice the proportion of its own total File and System activity. Again, these are indicators of concealing behavior that is not present in benignware.

3.4 Introduction to Graph Networks

Building off of some of definitions described in Section 3.2.6, we will start by preparing a node feature vector from a directional flow graph $\mathcal{G}$. We have defined $\mathcal{V}$ as the set of vertices on the graph which represent spawned processes on a host OS. Each vertex has a set of node features, $\mathbf{h} = \{\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_N\}, \bar{h}_i \in \mathbb{R}^m$ where N is the number of nodes in the graph and m is the number of features for each node. In Fig. 3.7 we observe four nodes that are neighbors. For example, if $\bar{h}_2$ is the node under consideration it shares a neighbour with $\bar{h}_1$ and $\bar{h}_3$ but not with $\bar{h}_4$. Therefore, for any ith node we want to consider the influence of its neighboring nodes $j \in \mathcal{N}_i$.

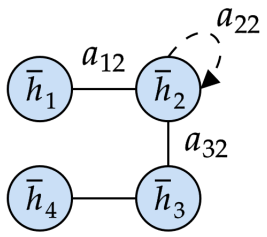


Figure 3.7: Graph structure of a 4 node cluster, complete with attention coefficients α between each node.

In Fig. 3.7 we also observe coefficient a_{ij}, where i, j correspond to the indices of neighboring vertices, which acts as an attention mechanism, known as the *attention coefficient*, which dictates the amount of influence to be paid between nodes. In Section 3.4.2 we will discuss the role these coefficients play and how they are computed. For now we will explore how we can make use of our feature vectors through the the use of linear projections.

3.4.1 Creating Higher-Order Features

To create a higher-order representation of our features a simple linear transformation is applied. In Fig. 3.8 the feature matrix, $\mathbf{h}$, is created using a simple *Bag-of-Words* feature representation (more on this in Section 3.4.3) by numerating whether or not a node uses a particular Windows API or not. If a process uses the particular API call it is incremented by 1, otherwise it is given the default

value of 0 meaning it was not used. Using a *weight matrix*, $\mathbf{W} \in \mathbb{R}^{m \times E}$ where E is the embedding dimension and m is the number of features, we can compute the embeddings for node j using the dot product between $\mathbf{W}$ and $\bar{h}_j$ - producing $\mathbf{W}\bar{h}_j$. These embeddings are learned and provide the model the ability to learn which features - in this case Windows APIs - are important in considering the maliciousness of nodes.

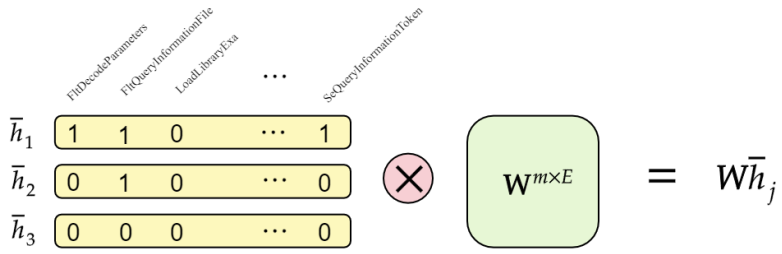


Figure 3.8: Creation of the node embeddings via the dot product between the BoW API matrix and the weight matrix $\mathbf{W}$.

3.4.2 Attention Coefficient

The *attention coefficient* combines the influence of nearby nodes and allows every node to attend to every other node on the graph framework. The expression for calculating a_{ij} between nodes i and j is shown in Eq. 3.2; where $\bar{w}^T$ is a trainable weight vector that is transposed, $||$ signifies the double pipe operator which stacks the transformed feature vectors $\mathbf{W}\bar{h}$ along the second axis, and $LeakyReLU(\bullet)$ applies the leaky $ReLU$ activation function.

$$a_{ij} = \frac{\exp\left(LeakyReLU\left(\bar{w}^T\left[\mathbf{W}\bar{h}_i || \mathbf{W}\bar{h}_j\right]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(LeakyReLU\left(\bar{w}^T\left[\mathbf{W}\bar{h}_i || \mathbf{W}\bar{h}_k\right]\right)\right)} \tag{3.2}$$

The use of $leakyReLU$ has the effect of only considering the positive influences of the node feature vectors, while allowing the weight to possibly recover it were driven to less than 0 where the derivative evaluates to 0 in a normal $ReLU$. Additionally, the use of $\exp(\bullet)$ in the numerator and denominator implements a $softmax$ which ensures the *attention* coefficients for node $a_{ij} \in [0, 1]$ are normalized; as the denominator will always evaluate to 1 when considering all possible neighboring nodes $\mathcal{N}_i$ for node i. When we compute a_{ij} for all neighboring nodes, we can then proceed to calculate the transformed feature representation according to Eq. 3.3. Therefore all neighbors to i, j are considered through their respective a_{ij} values; with a final sigmoid $\sigma(\cdot)$ being used to introduce non-linearity. A less verbose explanation and original derivation of GATs can be found in [4].

$$\bar{h}_i^{'} = \sigma\left(\sum_{j \in \mathcal{N}_i} a_{ij}\mathbf{W}\bar{h}_j\right) \tag{3.3}$$

3.4.3 Vectorizing Stack Traces

This section will describe several methods that are used to transform Windows API usage by process into feature vectors for the GNN to learn from. Several techniques are employed, including *Bag-Of-Words*, *n-grams* and *Term Frequency–Inverse Document Frequency* (tf-idf).

a) Bag-Of-Words (BoW): considers the frequency of usage of a Windows API by a process by numerating how many times it is used. Put another way, it is the raw count of the API usage. In Fig. 3.8 each API has a count for whether or not it is used by the node process $\bar{h}_i$. BoW suffers from two major drawbacks: the inability to account for word positioning and the weighting of the vectors of more prevalent works overwhelming the influence of less frequent words - even with vector normalization.

b) N-grams: looks at all the unique n sequence of APIs, and creates a vector with the numeration of the sequence. N-grams improves on BoW by accounting for pairs - in the case of bigrams - or trigrams of API sequences. These sequences are taken from the stack traces, where they appear in order from low-memory space to high-memory space (`0xffffff`) in the stack. In Fig. 3.9 we see an illustration for a stack trace that has been resolved using the Windows Symbol Table to produce the names of the API calls. In this example `LoadLibraryExa` and `LoadLibraryA` appear after one-another, therefore the tuple `(LoadLibraryExa, LoadLibraryA)` is added to the corpus. Then the next element in the sequence would include `BaseThreadInitThunk` which would be added to the corpus with the previous element as `(LoadLibraryA, BaseThreadInitThunk)`. This captures all unique combinations of Windows APIs, and maintains some of the order in the sequences. When considering trigrams we would incorporate more information of the sequence by considering all combination of three APIs in succession.

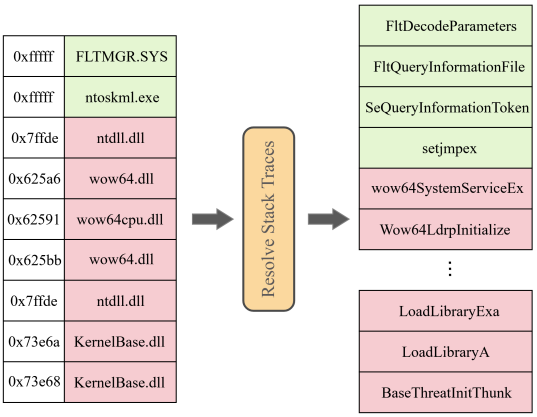


Figure 3.9: Translation from memory locations of imported DLLs to Windows API function calls. Retrieved from [3].

The decision to select larger n comes down to data sparsity. In a previous work the number of Windows APIs used by event processes were noted to be in the range of 1500-1700 [3]; which is a relatively low number compared to typical Natural Language Processing (NLP) problems. Larger values of n mean larger and larger sequences which will occur less frequently, meaning the model may underfit on the training data as it will generalize poorly. Conversely, a smaller n will lose the ability to trace longer sequences and will be unable to learn the importance of classifying maliciousness, thereby causing the model to underfit.

c) Term Frequency–Inverse Document Frequency: or tf-idf, is a technique which attempts to learn the importance of words by considering their inverse-frequency of occurrence. The motivation for this technique is that more frequent APIs are less important in distinguishing maliciousness,

while greater attention should be paid towards less frequent APIs. Theoretically this technique would be more powerful in attending to rarely used APIs that malware will use but benignware does not, while at the same time, disregarding commonly used APIs that are routine to the functionality of any running process. Calculating tf-idf (idf) is a straightforward process and can be done using Eq. 3.4.

$$idf_{td} = tf_{td} \times \log\left(\frac{N}{df_t + 1}\right) \tag{3.4}$$

Where tf_{td} accounts for the proportion of the API, t, used by the process d by dividing the count of t by the total number of APIs used by d. df_t keeps track of the number of occurrences of t by all N processes. The logarithm has the effect of dampening the explosion of the numerator for high values for N. When df_t gets larger for more frequent APIs, the term in the logarithm approaches 1 and the idf_{td} is small. When df_t is small for rarely used APIs the idf_{td} becomes larger and is given a larger weight. Through this preprocessing step APIs can be vectorized in such a way where malicious API usage can have a greater impact on model discrimination.

One question that may arise is in the use of n-gram for feature representation versus more sophisticated word embedding models such as word2vec [164] or GloVe [440]. The difference being is that word embedding models learn vector representations in d dimensional feature space, while n-gram simply encodes the sequence as a one-hot representation. The downside to n-gram being that all API calls and their sequences are orthogonal to one-another ($\mathbf{x^T x} = 0$), so while API_i and API_j may both be used in malicious contexts by malware, they are treated no differently in an embedding space when they are simply one-hot encoded. While for word embeddings API_i and API_j would ideally be very close to another if it were the case they were found in similar contexts in the stack trace. This was how previous work by Brezinski and Ferens [3] learned the embeddings of the stack trace to feed into a Transformer architecture. This did however require training an embedding layer from scratch, as publicly available word embeddings are almost always trained on the human language not API calls. However, there is a stark trade-off in practicality when considering either approach. Tf-idf n-gram is significantly better at capturing local contextual information effectively in certain tasks, and since API calls are localized to the stack trace and the windows event they are scoping, then it doesn't make intuitive sense to learn richer representations of sequences exceeding a few dozen tokens. Compare this to modern NLP models such as Generative Pre-trained Transformer (GPT) which can learn from sequences exceeding 1000-4000 tokens [441, 442]. Finally, since the number of API calls that commonly exist (see Table 4.3 for number of API calls in this work) belong to a limited set, then the drawbacks of n-gram being that they lack the ability to generalize to out-of-vocabulary words doesn't apply. This makes a simpler approach using n-gram suitable and appropriate for this work.

3.4.4 Batching Node Frameworks

In a GNN architecture the model needs to learn from various topologies in order to learn the contexts of malicious and benign node frameworks. For this reason the model will be trained in a *deductive setting*, where nodes are batched into training and validation sets. For a given topology the model

will be fed with a list of edges, $\mathcal{E}^{2\times|\mathcal{E}|}$, derived from an adjacency list connecting source nodes to destination nodes using PIDs and PPIDs, respectively.

In Fig. 3.10 an example of the topology for the execution of the Trojan.Kryptik virus is shown. The malicious `kryptik.exe` spawns from `Explorer.exe` because the executable was ran from the desktop in this example - similar to what a user might do if this executable was accidentally opened. Many daughter processes are also spawned, indicated by the red nodes which originate from Trojan.Kryptik. Ideally a model should be able to learn the structure of the topologies, such as in Fig. 3.10, while also being able to learn topologies where the virus believes its in a virtualized environment and fails to spawn any daughter process at all. That is why in this work we are considering the Windows API usage, which would include all the behavioural information prepared and vectorized as described in Section 3.4.3.

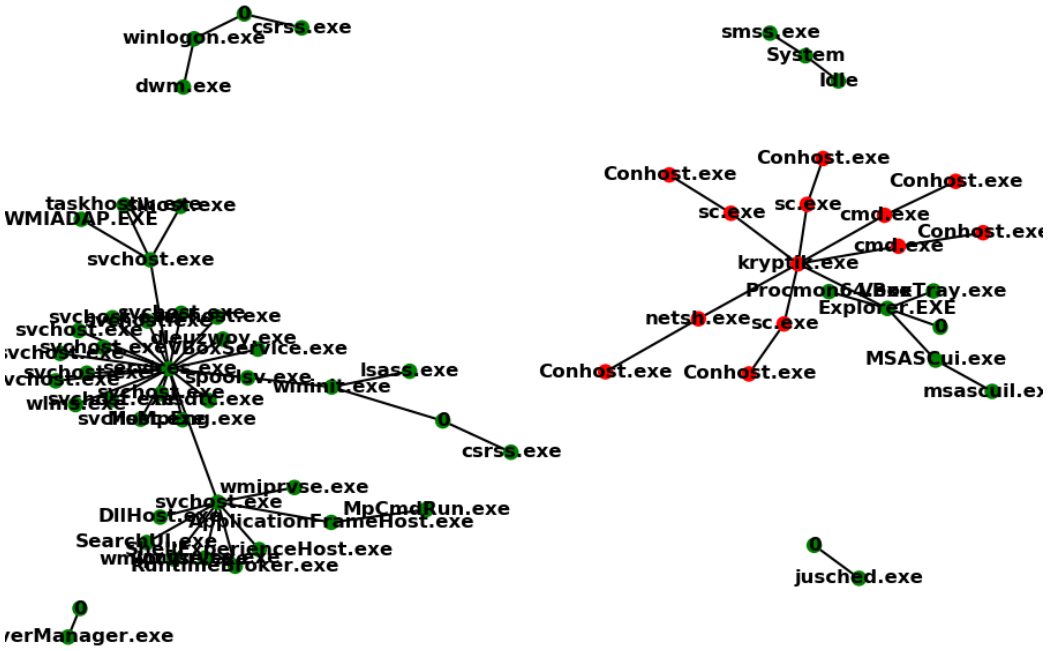


Figure 3.10: Execution of Trojan.Kryptik on a host operating system running Windows 8. Red nodes indicate those spawned by the parent process Trojan.Kryptik, while green nodes are Benign processes from the background operating system.

3.4.5 Graph Model Architecture

The model architecture for the proposed GAT is a vanilla one, with inspiration being drawn from the original GAT paper [4] as well as others [443]. A visual overview of the GAT layer is shown in Fig. 3.15. The first step in Fig. 3.15 is the preparation of the input examples based on the discussions on feature vectorization and batching node frameworks as discussed in Section 3.4.3 and Section 3.4.4, respectively.

Beginning with the linear projection discussed in Section 3.4.1, $\mathbf{W}$ is initializing as a weight vector of dimension $(n \times E)$. $\sum$ and σ are defined as the summation and sigmoid activation operators. *Softmax* was defined in the previous explanation for Eq. 3.2. The use of the *Dropout* layer is more

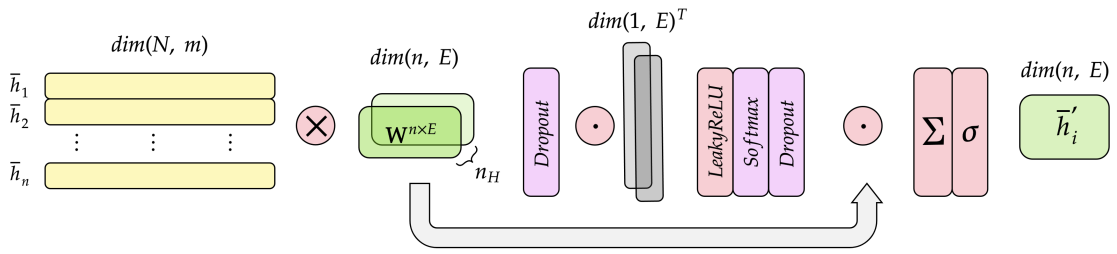


Figure 3.11: Summary of the inner workings of the GAT architecture.

nuanced and is left to its own explanation in Appendix C. The use of $\odot$ denotes the *Haddamard* operator or the element-wise product. The layer shown in Fig. 3.15 can be expanded to any arbitrary number of layers to create a deep GAT network. Due to the relative shallowness of the topology of the malicious process topologies, only 2 layers were used. This translates to the model learning about its two hop neighbourhood, i.e. the daughter of a daughter process. A full description of the model architecture, hyperparameter tuning, and training and validation considerations can be found in Appendix C.

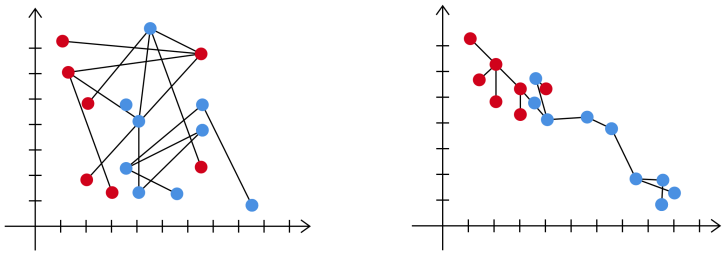
3.5 Introduction to Topological Complexity

The proposed work aims to combine the aforementioned GAT with that of a Complexity mechanism that incorporates the topological information of the embeddings themselves as the model is trained. This section will describe the theoretical underpinnings for this mechanism, as well as introduce two modes of implementation.

3.5.1 Theory of Topological Complexity

In Fig. 3.12 we see two illustrations for an example embedding, one being randomly initialized before training (Fig. 3.12a) and one following training (Fig. 3.12b) for a given number of training epochs. These illustrations can be seen as a theoretical visualization of the embeddings of the linear projection followed by projection onto a 2D embedding using t-distributed stochastic neighbor encoding or Principled Component Analysis (PCA). The red nodes are used to represent positive training examples, and the blue nodes represent negative training examples in a binary classification task. Typically when an embedding, like Fig. 3.12b, is produced a decision boundary (linear in this case for simplicity) can be placed to separate both classes of training examples. As the model learns the nodes are pushed farther apart as the model learns to distinguish the training examples without knowledge of which ones are malicious (the nodes are colored for illustrative purposes). Only after applying an activation function and a loss metric does the model discern how well it separates the samples. This work aims to investigate the application of complexity to distinguish the topology of these node embeddings as they are learned, with the hypothesis that the model will attempt to stretch the embeddings farther apart in an attempt to classify the malicious examples. The complexity measure will track this process, and the model will include this complexity measure in its learning process.

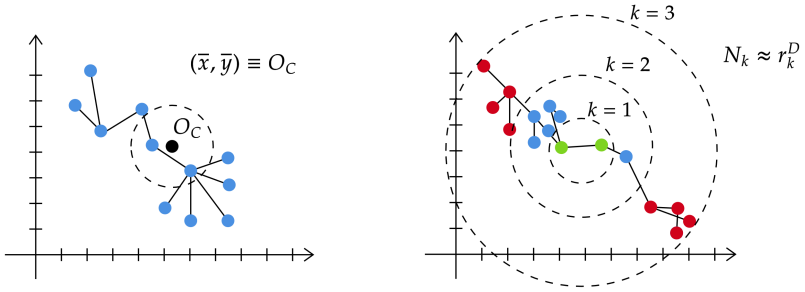
For example, Fig. 3.13a demonstrates a training batch with no positive training examples.



(a) (b)

Figure 3.12: Node embeddings for a model initialized and (a) untrained; (b) trained.

Contrast that with Fig. 3.12b with both positive and negative training examples, and you will see that the topologies are more spread apart due to the learned weights being updated. In Fig 3.13a we observe a marker for O_C which represents the center of mass of the topology. How far apart these nodes are from the centroid will provide a gauge of the *spread* of the topology. Additionally, in Fig. 3.13b we observe that when considering different subsets of nodes farther and farther apart from the centroid, in this case for different values of k colored differently, we can learn more and more about the topology. There is a relationship between the information contained within the nodes N_k considered at any given k, for a range r_k considered. This relationship, which follows a power-law distribution, plays a key part in determining the complexity of the spread.



(a) (b)

Figure 3.13: Visualization for the node embeddings, demonstrating (a) the centroid of a cluster, and (2) partitioning of the nodes in K partitions, where each color represents a different subset k.

In Fig. 3.13b we can capture the topology of the embeddings by incorporating a measure of the proximity of the nodes from the centroid at any given k. One approach is to use some form of distance-based metric, like Euclidean Distance, to measure the distance between nodes. We can use the more generalized form, the Minkowski Distance in Eq. 3.5, to measure the proximity of the nodes for any value of P where $P > 0$. When $P = 2$ Eq. 3.5 is equivalent to the Euclidean distance, and when $P = 1$ it is equivalent to the Manhattan distance.

$$d_{ij} = \left(\sum_{i=1}^{n} |x_i - y_i|^P \right)^{\frac{1}{P}} \tag{3.5}$$

More clustering equates to smaller distances between nodes, while less clustering leads to larger

distances between nodes. Now that we can produce information related to the *spread* of the topology, we need to incorporate this information in such a way so that it is invariant to the scale or size of the topology. There are several approaches to do so through the determination of the Fractal Dimension, which will be discussed in the following sections.

3.5.2 Fractal Dimension Application

This approach aims to measure the Complexity of these topologies by using a Mass-Radius, or Radius of Gyration [444] fractal dimension. At different radii r_k, the information, lets call it A_k, of the nodes N_k that fall within the radius under consideration follows a power law relationship that can be extracted from the exponent D. This relationship assumes that $A_k \sim r_k^{D_M}$ for $r_k \to 0$. Taking successive measurements at different values for k can give an estimate of the Mass-Radius by observing the limit in Eq. 3.6:

$$D_m = \lim_{R_k \to \infty} \frac{\log(A_k)}{\log(r_k)} \tag{3.6}$$

Using this approach we can measure the mass of the fractal topology by measuring the spread of the nodes from the center of mass of the topology. This way the standard deviation, σ, of the spread can be used to approximate the mass of the topology; which can then be related back to Eq. 3.6 as the value for A_k. This is done in practice my measuring the standard deviation of the distance to the nodes from the centroid, as defined in Eq. 3.7:

$$\begin{aligned} D_{Mk}(N_k) &= \sqrt{\sigma_x^2 + \sigma_y^2} \\ &= \sqrt{\frac{1}{N_k} \sum_{i=1}^{N_k} r_j^2} \end{aligned} \tag{3.7}$$

Where r_j is the distance between the j^{th} node to the centroid O_C. One thing to note is that Eq. 3.7 can be extrapolated to work for any d-dimensional data, and is shown in Eq. 3.8 for two (2) dimensions.

$$\begin{aligned} D_{Mk}(N_k) &= \sqrt{\frac{1}{N_k} \sum_{i=1}^{N_k} r_j^2} \\ &= \sqrt{\frac{1}{N_k} \sum_{i=1}^{N_k} (x_i - \bar{x})^2 + \frac{1}{N_k} \sum_{i=1}^{N_k} (y_i - \bar{y})^2} \end{aligned} \tag{3.8}$$

Mass Radius Fractal Dimension

In Alg. 1 we see an implementation of the Mass-Radius fractal dimension. The main function MASS-RADIUS-2D is initialized with a set of nodes, $\mathcal{D}$, where $\mathcal{D} \in \mathbb{R}^2$, and scales K ,where $K \in \{1, 2, \ldots, n_k\}$ where n_k is the largest scale used, which is equal to $|K|$. The function DETERMINE-CENTROID-2D on Line 9 is used to determine the center of mass, O_C, where $O_C \equiv (\bar{x}, \bar{y})$. Function SORT on Line 12 is used to sort the nodes based on distance d from centroid O_C. The SPLIT function has the

effect of producing a set N_K such that $N_K \subset \mathcal{D}$ of K partitions of length $\lfloor L/K \rfloor$ along with $L \pmod K$ sub-arrays of length $\lfloor L/K \rfloor + 1$ where $L = |\mathcal{D}|$. For each subset $\mathcal{N}_k$ more and more points are considered farther away from the centroid, thereby giving an indication of the spread as more and more points are considered (shown on Line 15). One important thing to note is that Lines 11 and 17 have computation that is repeated in Alg. 1. For the final iteration of k in Line 14 it is the same as Line 11 as it sums up all the standard deviations, with the addition of the $1/N_k$ term. In practice Line 17 simply computes the summation of the standard deviation until N_k and does not re-compute the distances. In the final implementation they are not repeated, and are simply shown in their entirety for clarity.

As for time complexity, the most heavily weighted functions in terms of time invested would be DETERMINE-CENTROID-2D, SORT and Lines 11 and 17. DETERMINE-CENTROID-2D would have time complexity $\Theta(n \times d)$, where n is the number of nodes and d is the dimensionality of the embedding. SORT would have complexity $O(n \log n)$ based on an efficient comparison-based sorting algorithm of the node distances, and Line 11 would involve $n \times d$ summations of constant time work $\Theta(1)$. Line 17 is a little less straightforward, as the bulk of the computation is already done in Line 11, with the addition of the summations. For a value of K equal to 3, the loop will perform $n/3$ summations in the first iteration, then $2n/3$ in the second, followed by the full summation of n terms which is repeated from Line 11. Similarly, for $K = 4$ we would perform $n/4$, $2n/4$, $3n/4$ then n number of summations. Knowing that much of the computation is repeated at each kth sum, we can compute a running sum of the distances so that we only need to perform n summations and retrieve the sum we need at index $\lfloor n/k \rfloor$ as our answer. Since n is already calculated in Line 11, we only need the running sum up until $n(K-1)/K$, which would be $3n/4$ for a K of 4, or $2n/3$ for a K of 3; leading us to our running time of $\Theta(n(K-1)/K)$. Combining all the costs into one produces the following run-time execution for MASS-RADIUS of d dimensions:

$$\text{MASS-RADIUS}(n, d, K) \in \Theta(n \times d) + O(n \log_2 n) + \Theta(n(K-1)/K) + \Theta(1)$$
$$where \{n, d, K \in \mathbb{Z}^+ | K \neq 1\}$$

Radius of Gyration Fractal Dimension

The Radius of Gyration Fractal dimension is similar to the Mass Radius dimension with a few caveats. Namely, rather than the centroid being computed for the whole node framework, it is computed for each subset $\mathcal{N}_k$. This has the effect of determining the *spread* of the node framework as it expands, providing a better approximation of the structure when the node framework is asymmetrical - which is preferably and likely is for node embeddings. The changes to the computation in Alg. 1 is trivial to produce the Radius of Gyration FD and is shown in Alg. 2 for completeness.

We observe DETERMINE-CENTROID-2D is contained within the loop for all $k \in K$ of Line 13. Similar to Alg. 1 there is computation that is repeated in Line 15. While Line 15 is repeated for each k, only a running sum is required as Lines 3 and 4 of DETERMINE-CENTROID-2D can be calculated easily on the fly with a small requirement for auxiliary storage. Given that fact, only $n \times d$ summations are required, meaning the running-time cost is the same for MASS-RADIUS. There

Algorithm 1 Mass-Radius Fractal Dimension

function DETERMINE-CENTROID-2D($\mathcal{D}$)
  $\langle x, y \rangle := \mathcal{D}$
  $\bar{x} := \frac{1}{N} \sum_{i=1}^{N} x_i$
  $\bar{y} := \frac{1}{N} \sum_{i=1}^{N} y_i$
  return $\bar{x}, \bar{y}$
end function

function MASS-RADIUS-2D($\mathcal{D}, K$)
  $\bar{x}, \bar{y}$ = DETERMINE-CENTROID-2D($\mathcal{D}$))
  $\langle x, y \rangle := \mathcal{D}$
  $d := \sqrt{\sum_{i=1}^{N} [(x_i - \bar{x})^2 + (y_i - \bar{y})^2]}$
  $\mathcal{D}_{sorted}$:= SORT($\mathcal{D}, d$) ▷ sort elements by distance from centroid
  $\mathcal{N}_K$:= SPLIT($\mathcal{D}_{sorted}, K$) ▷ split dataset into K splits
  for all $k \in K$ **do**
    $\mathcal{N}_k := \mathcal{N}_K[1:k]$ ▷ fetch splits up until k
    $\langle x^k, y^k \rangle := \mathcal{N}_k$
    $R_{Mk} := \sqrt{\frac{1}{N_k} \sum_{i=1}^{N_k} [(x_i^k - \bar{x})^2 + (y_i^k - \bar{y})^2]}$
  end for
  $D_M := \lim_{k \to \infty} \frac{\log N_k}{\log A_k}$ ▷ extract exponent from log-log plot
end function

is another form that Eq. 3.8 can take as to avoid the batch computation of the mean value for $\bar{x}$ and $\bar{y}$. For just the standard deviation in 1-dimension, Eq. 3.9 can be used to calculate σ_x^2 (proof found in Appendix B) requiring only a running sum of $\sum_{i=1}^{N_k} x_i$ and $\sum_{i=1}^{N_k} x_i^2$. The proposed work will only use batched node frameworks, but an extension to an on-line mode is possible in a future application. Both Radius of Gyration and Mass-Radius Fractal Dimension runtimes are recorded in Appendix C.

$$\sigma_x^2 = \frac{1}{N_k} \left[\sum_{i=1}^{N_k} x_i^2 - \frac{1}{N_k} \left(\sum_{i=1}^{N_k} x_i \right)^2 \right] \tag{3.9}$$

Provided the expressions are now defined in Alg. 1 and 2, the next step is to generate a dataset in which to make use of these expressions. The representation $\langle x, y, \dots z \rangle$ can be extrapolated to represent any d-dimensional solution space, thereby making the FD measure a powerful tool for any application. Following from the discussion in , this work aims to look at the node embeddings in a graph to determine their complexity, and this can be done on the original feature vector h or a higher-order representation $h\prime$ generated from a deep learning layer. More on the generation of these node embeddings follows this section. For a preliminary application of both FD measures on a simple multi-layer perceptron network, the author refers the readers to previous published work in [10]. The work in [10] provides a solid theoretical and practical foundation for the methodology of work that follows.

3.5.3 Proposed Complexity-based Graph Attention Network

This section will outline various approaches to introducing the complexity measure into the graph model architecture. Further reference to this architecture will refer to it as a Comple**X**ity-based

Algorithm 2 Radius of Gyration Fractal Dimension

function DETERMINE-CENTROID-2D($\mathcal{D}$)
  $\langle x, y \rangle := \mathcal{D}$
  $\bar{x} := \frac{1}{N} \sum_{i=1}^{N} x_i$
  $\bar{y} := \frac{1}{N} \sum_{i=1}^{N} y_i$
  return $\bar{x}, \bar{y}$
end function

function RADIUS-GYRATION-2D($\mathcal{D}, K$)
  $\langle x, y \rangle := \mathcal{D}$
  $d := \sqrt{\sum_{i=1}^{N} [(x_i - \bar{x})^2 + (y_i - \bar{y})^2]}$
  $\mathcal{D}_{sorted} :=$ SORT($\mathcal{D}, d$) ▷ sort elements by distance from centroid
  $\mathcal{N}_K :=$ SPLIT($\mathcal{D}_{sorted}, K$) ▷ split dataset into K splits
  for all $k \in K$ **do**
    $\mathcal{N}_k := \mathcal{N}_K[1:k]$ ▷ fetch splits up until k
    $\bar{x}_k, \bar{y}_k =$ DETERMINE-CENTROID-2D($\mathcal{N}_k$)
    $\langle x^k, y^k \rangle := \mathcal{N}_k$
    $R_{Gk} := \sqrt{\frac{1}{N_k} \sum_{i=1}^{N_k} [(x_i^k - \bar{x}_k)^2 + (y_i^k - \bar{y}_k)^2]}$
  end for
  $D_G := \lim_{k \to \infty} \frac{\log N_k}{\log A_k}$ ▷ extract exponent from log-log plot
end function

Graph **A**ttention **N**etwork, or **XGAN**. How the complexity measure will be incorporated into the GAT, how it will be configured, and how the measure can improve the model performance is the topic of discussion for this section. The considerations for the integration are the following:

How are the complexity layers positioned within the network? Ideally we would want the complexity measures to be incorporated into different stages of the network, as to assess performance and efficacy of computing the complexity of the intermediate embeddings. If additional layers are added which don't improve training or are redundant, the model should be able to drive the weights to zero in an attempt to diminish their effect. Additionally, one must consider the effect of computing the complexity sooner than later in the network. Fig. 3.14 demonstrates how complexity can be added between layers in a simple Feedforward Multilayer Perceptron. After each linear projection in d dimensional space, the complexity of the embedding space is captured.

What are consideration for placement of the complexity layer prior- or post-activation? The main difference is that in post-activation the samples are projected into highly non-linear space, therefore Euclidean-based distance metrics may be hard to capitalize on in the complexity layer. For example, the fractal dimension can be computed according to Eq. 3.10 which is pre-activation; where z represents the net-input following linear transformation $z = \mathbf{w}^T \mathbf{x}$; $ReLU(\cdot)$ is the activation function and D_M is the complexity measure. Or alternatively, post-activation (Eq. 3.11) which computes D_M after the activation function which computes D_M based on the result of the activation $ReLU(z)$ in this case.

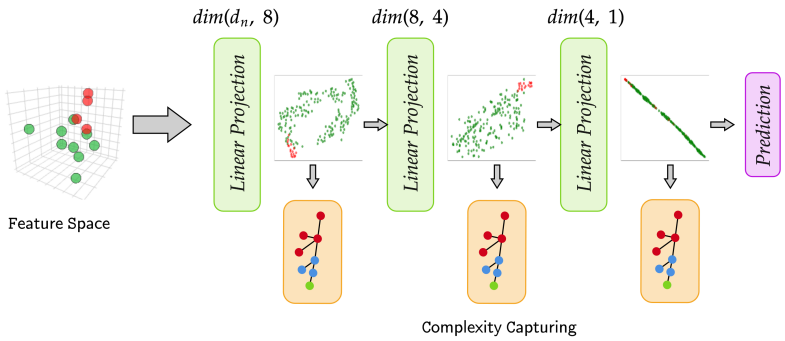


Figure 3.14: Addition of the complexity layers between the linear projection layers of a simple feedforward MLP architecture. Retrieved from previous work in [10].

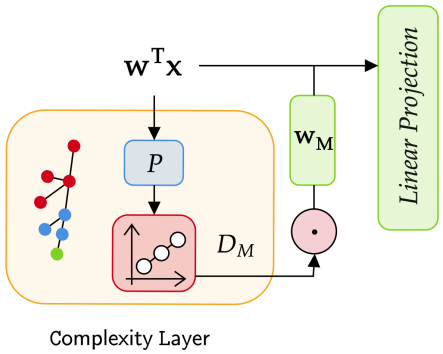


Figure 3.15: Summary of the inner workings of the GAT architecture.

$$a = ReLU(D_M(z) + z) \tag{3.10}$$

$$a = D_M(ReLU(z)) + ReLU(z) \tag{3.11}$$

What are the optimal hyperparameters for the complexity layer? What should be the optimal order of the norm P, and in what scenarios would Mass Radius versus Radius of Gyration be suitable? Part of this analysis was carried out in previous work in Brezinski and Ferens [10]. Is it necessary to include a trainable weight w_D to the model output? This will provide the model the ability to determine what layers it finds helpful for model performance, and which ones are better removed. Figure 3.15 provides an illustration on the inner workings of the proposed XGAN layer. The order of the norm P determines the distance metric used, while the trainable weight w_D is used to assess how much importance should be placed on the complexity measure D_M. Ideally, if the measure hinders performance, D_M is driven to 0. In cases where D_M is informative and improves model performance, the model should adjust the weight w_D accordingly until the model converges.

3.6 Benchmark Datasets

For the purposes of providing a proof-of-concept of the complexity implementation, a benchmark graph dataset was used. Initial testing on a synthetic dataset was important as a preliminary step in showing the sensitivity to different parameters of the complexity approach for an application with a GNN. Additionally, a graph benchmark dataset can be used to test the complexity implementation on a vanilla graph architecture that is much smaller and simpler than process execution on malware behavior on a host OS.

This work tested three graph benchmark datasets that are widely used in literature for assessing the performance of GNNs for a variety of applications. These datasets were chosen because they include multi-categorical classification, include highly interconnected networks with a large number of edges, and are single, large static graphs. These qualities are not covered by the malware execution dataset developed in this work, so they demonstrate alternative applications for the Complexity layer. A brief descriptive summary of each dataset follows, with a summary of the dataset attributes shown in Fig. 3.3.

The Cora dataset[7] consists of 2708 scientific publications classified as one of 7 classes, each corresponding to a category of scientific literature. Node features are binary (presence of a word or not) BoW vectors of size 1433, used to indicate if a particular word is used in that publication. Each citation between two papers is encoded as one of 5429 directional edges between the nodes. This dataset is a single topology as opposed to several graph topologies found in the sandbox execution graphs, but includes multi-category classification to the mix.

The CiteSeer dataset[445] consists of 3312 scientific publications classified into one of six classes, each corresponding to a category of scientific literature. This collection included 4732 directional edges. Each node includes a 0/1-valued word vector indicating the absence/presence of the corresponding word from a dictionary of size 3703.

The FaceBook Page-Page dataset [446] contains 22,470 nodes each representing verified pages on Facebook. The 342,004 connected edges each identify a mutual like between two pages. Node features are extracted from the site descriptions that the page owners created to summarize the purpose of the site. In total there are 128 node features, and each page is classified into 4 classes as defined by Facebook. These categories include: politicians, governmental organizations, television shows, and companies.

Table 3.3: Summary of the benchmark datasets used in this work, along with training and validation splits used for training.

	# of Nodes	**# of Features**	**# of Edges**	**# of Classes**	**Training/Validation Split**
Cora	2,708	1,433	5,429	7	2,208/500[162]
CiteSeer	3,312	3,703	4.732	6	19,217/500[162]
Facebook	22,470	128	342,004	4	0.9:0.1 (this work)

[7] https://relational.fit.cvut.cz/dataset/CORA

3.6.1 Model Performance Metrics

To evaluate the model performance for synthetic, benchmark and original datasets, several performance metrics are defined and used. During model training we are concerned with assessing the training performance of the model, and it's ability to generalize to new never-before-seen instances. For this we use a validation set which is a subset of a dataset $\mathcal{D}_{val} \in \mathcal{D}$ that is used to evaluate the model's performance during the training process. It is typically used to tune the hyperparameters of the model and to select the best model among a set of candidate models. A description of the activities to train, evalulate and record the model performances are listed in Section C.3 in Appendix C. During training the validation loss, accuracy and F1 score are recorded as to evaluate the efficacy of model training.

Loss - is defined in this work as the Cross-Entropy loss which is used to measure the difference between the predicted probability distribution and the true probability distribution of the target variables. The equation to calculate Cross-Entropy is shown in Eq. 3.12 for C_s classes for the true and predicted probability y_i and $p(y_i)$, respectively, for class i. In this case y_i is the one hot encoded label vector. Crossentropy and Eq. 3.12 is a generalized form of the Binary Cross-Entropy, which only applies to predictions with 2 classes.

$$\mathcal{L}_{CE} = \sum_{i=1}^{C_s} c_i \cdot y_i \log(p(y_j)) \tag{3.12}$$

For the sandbox execution datasets, a modification was made to Eq. 3.12 to account for the heavy class imbalance of the dataset. The term c_i where $c_i \in \mathbb{R}^{C_s}$ accounts for the class imbalance by penalizing the loss function by c_i when class i is batched. For example, if $c = [1., 3.]$ then for each sample belonging to class 2, the sample y_2 is penalized 3 times more than if a sample belonging to class 1 y_1 is batched. This allows the model to modulate the loss accordingly so that it doesn't learn to minimize the loss by guessing the sample is from the negative training example all the time.

k-fold Cross Validation was used to reduce the variance of the model performance by rotating the samples used in the validation set. This allows each sample node to be used for $k-1$ training steps so the results of each individual trial can be averaged to give a more accurate estimate of the model's performance. Eq. 3.13 outlines the calculation for Cross Validation, with a similar calculation being carried out for **Accuracy** and **F1-Score** (explanation to follow) by substituting the metric used ($\mathcal{L}_{CE}$ in this case).

$$\mathcal{L}_{CV} = \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}_{CE,i} \tag{3.13}$$

Accuracy - in this work is calculated as the macro-average of the accuracies whereby the arithmetic mean of each individual class is taken into account. Therefore, for C_s classes, the macro-average accuracy is calculated according to Eq. 3.14. The expression for Eq. 3.14 has the advantage of calculating the accuracy for each class independently, meaning class-imbalance is accounted for in the metric according the number of training examples for each class.

$$Average_{macro} = N_1 \cdot Acc_1 + N_2 \cdot Acc_2, \ldots + N_C \cdot Acc_{C_s} \tag{3.14}$$

F1-Score - is a performance metric, where $F1\text{–}Score \in [0, 1]$, that is used to evaluate the accuracy of a classifier. It is defined as the harmonic mean of precision and recall, where precision is the fraction of TP predictions among all positive predictions, and recall is the fraction of TP predictions among all actual positive instances. F1-score can be calculated according to Eq. 3.15; where $precision = TP/(TP+FP)$ and $recall = TP/(TP+FN)$. This metric accounts for class imbalance, providing a meaningful indicator of model performance on both the positive and negative training examples.

$$F1\text{–}Score = 2 \times \frac{precision \times recall}{precision + recall} \tag{3.15}$$

Chapter 4

Results and Discussion

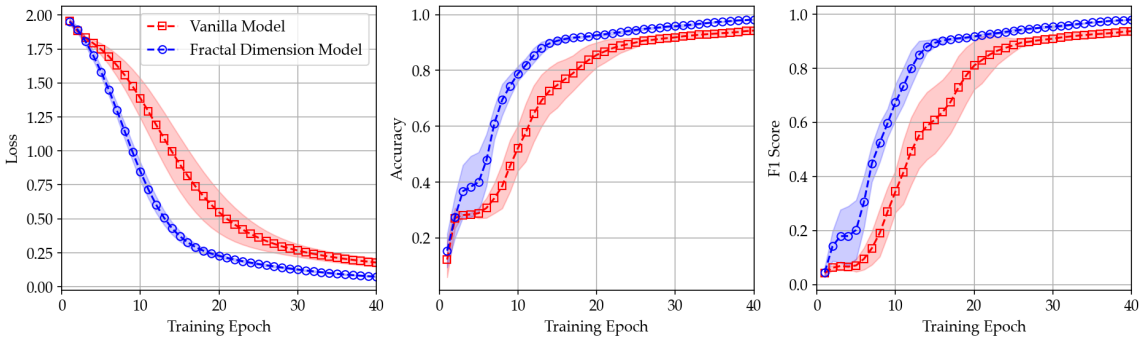
This chapter results of several stages of experimentation. The results are separated into two (2) broad categories: the results from initial benchmarking in Section 4.1, followed by the application of the XGAN architecture towards malware execution in Section 4.2. To ensure this chapter focuses solely on Results and Discussion, additional content related to model configuration and viewing performance results has been set aside to be discussed in Sections C.2 and C.4 in Appendix C. Where applicable, reference to the model architecture and configuration will be made to the relevant code listing in Section C.2. First, the benchmark datasets will serve to demonstrate some preliminary results on the large, static graphs, with additional insights on the hyperparameters used for the complexity layer.

4.1 Complexity Evaluation on Graph Benchmark Datasets

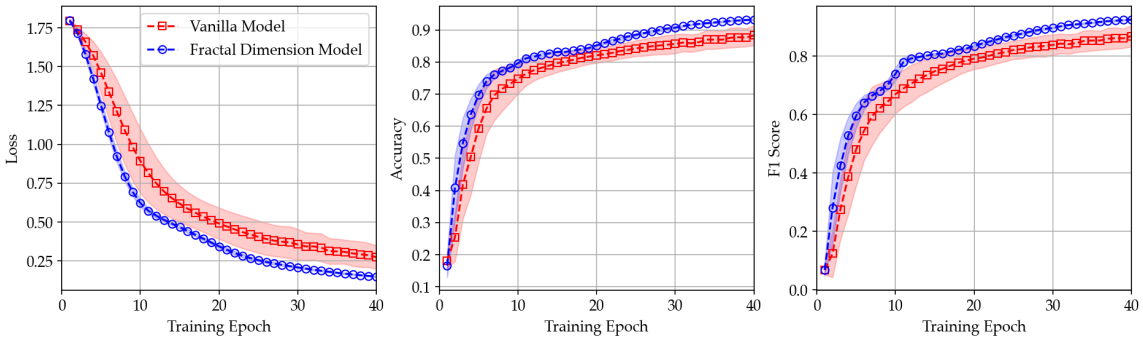
This section summarizes the performance of the Complexity layers as it is applied on a series of benchmark datasets that were described in Section 3.6. Table 4.1 provides the training performances for the series of benchmark problems when using the Complexity layer as the first layer, or in the zeroth position. This experiment included the Complexity measure being set to be the Radius of Gyration Fractal Dimension (referred to as RoG going forward), the measure being computed pre-activation, as well as a few other model parameters which are summarized in code listing C.1. More on the basis for setting these parameters and optimizing them are covered in Section 4.2.2.

Where the model labels are listed as **Vanilla** this refers to the GNN model without any complexity layer included. This provides a baseline to compare the XGAN model performance to. To visualize the model training process, the validation scores are shown in Figure 4.1 as the model is trained. This is used to visualize the change in performance of the model over training time. Table 4.1 presents the final results following model convergence after 40 training epochs, while Fig. 4.1 presents the performance on the validation set taken after each epoch. This is used to assess under-training or over-training on the training set as the model learns from the training data.

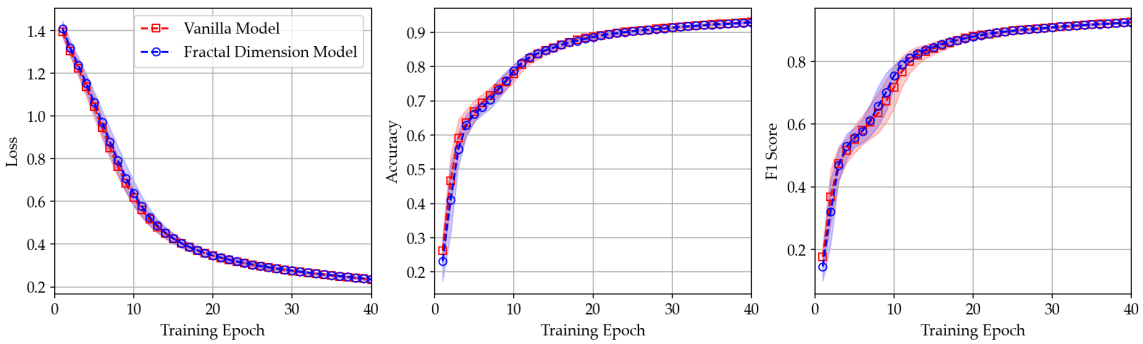
Results demonstrate a performance boost when the Complexity layer is added (labelled as **Fractal Dimension Model**) as compared to the **Vanilla** architecture, i.e. no Complexity layer. For a simpler model such as Cora and CiteSeer the improvement is more evident, with half the training loss after 40 epochs of training (0.04 versus 0.08). For the larger Facebook dataset the improvement



(a) Cora Validation Loss, Macro-Accuracy, and F1 scores



(b) CiteSeer Validation Loss, Macro-Accuracy, and F1 scores



(c) Facebook Page-Page Validation Loss, Macro-Accuracy, and F1 scores

Figure 4.1: Model performance metrics for the (a) Cora, (b) CiteSeer and (c) Facebook benchmark datasets compared for the Vanilla and RoG FD measure. Shaded regions are 95% CI for the 18 iterations tested.

Table 4.1: Model validation performance for a set of benchmark graph problems. Experiments were run for 18 iterations each, with independent t-test values for the training loss shown below in the table subcaption. Elements **bolded** signify the best performing model for each dataset. Model architecture is outlined in code listing C.1 in Appendix C

		Loss$^{\alpha}$ ↓	**Accuracy (%)** ↑	**F1** (10^2) ↑
Cora	FD	**0.04 ± 0.005**	**98.62 ± 0.2**	**98.40 ± 0.2**
	Vanilla	0.08 ± 0.014	97.45 ± 0.3	97.13 ± 0.4
CiteSeer	FD	**0.11 ± 0.01**	**93.97 ± 0.6**	**93.41 ± 0.6**
	Vanilla	0.14 ± 0.03	93.07 ± 0.4	92.41 ± 0.5
Facebook	FD	0.19 ± 0.015	**93.67 ± 0.7**	**93.20 ± 0.7**
	Vanilla	**0.19 ± 0.013**	93.60 ± 0.5	93.07 ± 0.5

$^{\alpha}$ t(18)=5.12, $p =< 0.001$, t(18)=8.37, $p =< 0.001$ and t(18)=0.0047, $p = 0.99$ for the Cora, CiteSeer, and Facebook datasets, respectively.

was not shown to be statistically significant, perhaps due to the variability in learning for such a large graph with a large number of nodes and edges. This is confirmed by the accompanying plot of performance versus epoch shown in Fig. 4.1c. It is also the case that the Facebook dataset has substantially lower node feature dimensionality than the Cora and CiteSeer datasets (128 versus 2,708 and 3,312, respectively) which may demonstrate the effectiveness of the complexity measure for high dimensionality feature sets but less so for simpler ones where the Vanilla models are more comparable. This was shown to be the case in previous work by Brezinski and Ferens [10], where the complexity measure was show to be more sensitive to small changes in higher dimensions due to the fact that the calculation for complexity computes the spread in all dimensions simultaneously. This is also typically the case for Deep Learning more broadly where the additional capacity of a model has the effect of either promoting overfitting or hindering model convergence when the additional capacity is not warranted for the complexity of the dataset. For the Cora dataset, Fig. 4.1a shows an interesting variability on the models Macro-Accuracy and F1 Score at epoch $\approx 3-8$. It is hard to attribute this to any specific cause, but given the additional complexity measure, it may be the model's way of modulating the complexity weight early on, leading to noisy Accuracy and F1 Scores which help to improve convergence later on during training.

In examining the model performance during training, Fig. 4.1 demonstrates how these metrics oscillate and converge during the training process; with the 95% CI shaded regions shown for each model in red and blue. The models were all shown to converge in approximately 30 to 40 epochs. In terms of variation during training, the Fractal Dimension Model (RoG) was shown to exhibit less oscillation for the validation loss throughout the training process. There are some potential explanations for this behaviour. The most likely reason is that the model is using the spread of the input features (calculated via the complexity layer) as a feature itself, and making use of that information alongside the node features. In this way, the complexity layer is operating as a high-dimensional clustering feature. Since the complexity measure produces a scalar output, the measure operates as a effective feature compressor which contains information of all the features simultaneously. This is due to the fact that the complexity is computed pre-activation via $ReLU(\mathbf{z})+D_M(\mathbf{z})$. While the node features contain local information, that is to say the node features are

created based solely on the node feature vector and the available information contained within that node (in addition to neighbor nodes after graph convolution), the complexity layer better provides global information that incorporates the feature embeddings of all node features at once through this clustering mechanism. The graph convolutions work to pass messages between neighbouring nodes at each layer, while the complexity layer is doing so for all nodes in the graph network and providing that information towards the learning process. This is the case for these benchmark datasets as the graphs are fed as large, static graphs, so the performance is expected to be different when the training step contains smaller subgraphs as we will see in Section 4.2.

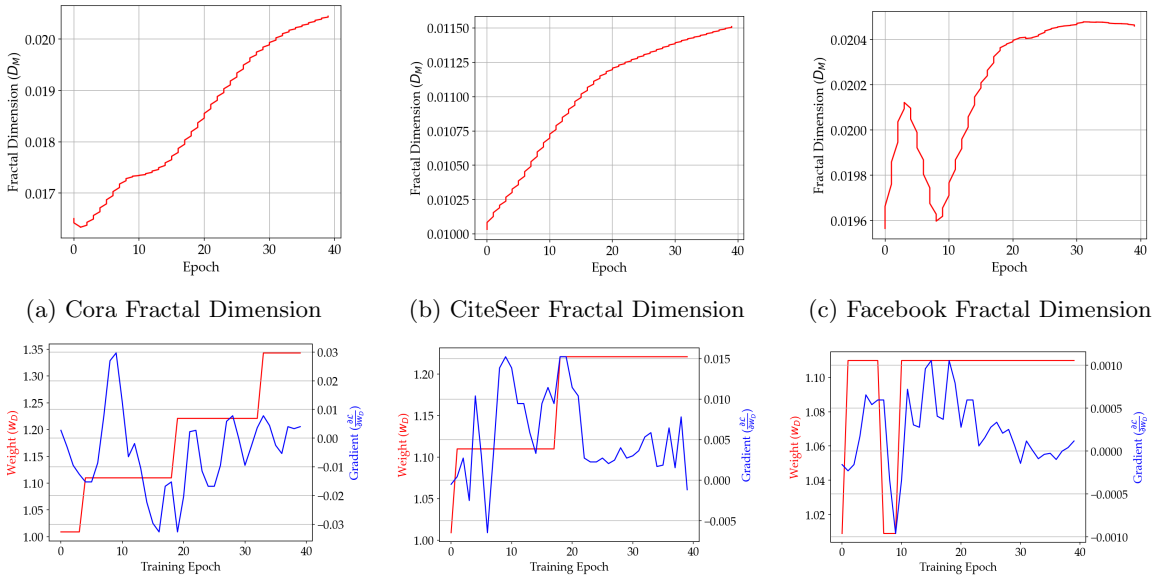
Based on the additional complexity measure, the model exhibits less variation and less oscillation as compared to the Vanilla implementation. Overall this is considered beneficial for the learning process and the final performance, as oscillations in loss lead to oscillations in gradients, which subsequently lead to slower convergence as the model attempts to search for an optimal global minima in the solution space. Some ways to resolve this is to use larger batch sizes, however, for these benchmark datasets all nodes and edges (based on training/validation sets) were batched for each epoch as a single large static graph.

4.1.1 Complexity Model Weights and Gradients

To provide further insight into how the Complexity layer functions during model training, an analysis was carried out on the complexity measures themselves, the model weights, and their gradients. This form of analysis is important when developing a new technique for two reasons. First, understanding how the complexity layer interacts with an existing model is important as it can provide further insight into how the model is being hindered or benefiting from the additional component. Since the layer is interwoven within the existing architecture, and plays a part in the computational graph that makes up back-propagation, how it interacts with the other components is an important and necessary step in assessing the feasibility of incorporating a new layer into a model. Secondly, since the complexity layer plays a part in backpropogation through the use of the trainable weight w_D, the layer should learn jointly with the rest of the model as the model as a whole converges to a globally optimal solution. This means that while the rest of model converges in terms of loss and generalization accuracy, the complexity layer should do so as well and should not oscillate widely and perturb the system unnecessarily. There are scenarios where perturbation is helpful, such as in *Dropout* where the model tries to improve generalization performance by learning from various combinations of model architectures during training.

In Fig. 4.2 we can see each measure of the Fractal Dimension (D_M) (see subplots 4.2a, 4.2b, and 4.2c) along with the complexity weight w_D and the gradient $\frac{\partial \mathcal{L}}{\partial w_D}$ (see subplots 4.2d, 4.2e, and 4.2f) over a single iteration trained for 40 epochs. The top subplots identify the change in the Fractal Dimension (D_M) over the 40 training epochs, while the bottom subplots superimposes the trainable weight w_D with the gradient of the weight at each apoch. What is noteworthy is that the complexity measure D_M converges to some value as the model converges as a whole. This is evident from the fact that the gradient oscillates early on, but seems to reach a somewhat steady-state with additional training epochs. Additionally, the gradient converges to 0 as the whole model converges, meaning the complexity weight value is hampered as it approaches steady state with the rest of the model. This is preferable in a DL application as we wouldn't want different layers of a model making wide changes to their learnable parameters while others converge, as this leads to oscillating losses and

gradients.



(a) Cora Fractal Dimension

(b) CiteSeer Fractal Dimension

(c) Facebook Fractal Dimension

(d) Cora learnable weight and gradient

(e) CiteSeer learnable weight and gradient

(f) Facebook learnable weight and gradient

Figure 4.2: Fractal Dimension measure and learnable weight and gradient plots.

It is also worthy to note that the benefits of the complexity layer have an alternative feature that has not been discussed yet. At each complexity layer added, D_M is added to the feedforward mechanism as $w_D D_M + \mathbf{z}$. In this case the network has full control over D_M via the ability to set w_D to zero, in effect skipping the layer entirely. This is useful in many cases where the model does not feel the layer is helping, it has the ability to ignore it entirely - adding to the robustness of the measure to different circumstances that may arise. Therefore, in cases where many complexity layers are added, if the model does not find them particular useful, or finds them to be useful fractionally, these behaviors can be controlled through the w_D^l for each layer l. This is not the case with a typical Linear Layer, where the model cannot set all the weights to 0 in a single hidden layer without disrupting the flow of gradients in downstream layers. Additionally, during backpropogation the gradients are distributed equally to w_D and $\mathbf{z}$ because addition distributes gradients equally to its branches. This can be shown to be the case if one were to compute the gradient of a weight w_{l-1} in the previous layer $l-1$, then $\frac{\partial w_D}{\partial w_{l-1}} + \frac{\partial \mathbf{z}}{\partial w_{l-1}}$. The uninterrupted gradients means the layer does not contribute to vanishing or exploding gradients; with the drawback being it does make the layer less expressive in how much it can influence the network. This consideration for adding a *residual* connection was inspired by the work of [447] and [448] in their applications of Transformers and Residual Networks, respectively.

4.1.2 Complexity Interpretability

It is worth briefly discussing model interpretability. DL models are often referred to as "black box" models because it is difficult to understand how they arrive at their predictions. These models are composed of many layers of interconnected nodes, and it is not easy to understand how the input

data is transformed as it passes from the complexity layer to the rest of the network to produce the output predictions. Beyond the complexity layer itself, the internal workings of the model are not transparent, making it difficult to understand how the model is making its decisions [271]. This is in contrast to more traditional machine learning models such as decision trees and logistic regression models, which are more transparent and easier to interpret [449].

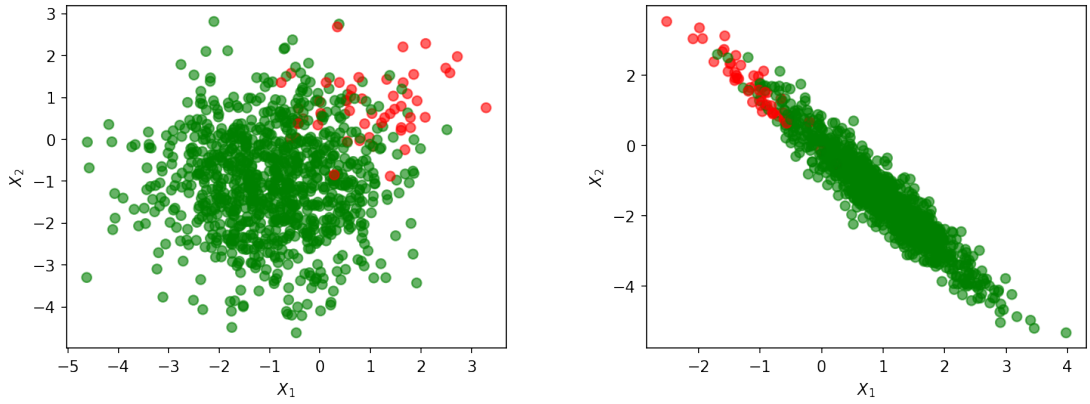
It is difficult to draw any conclusions from the raw values of D_M in Fig. 4.2a, 4.2b and 4.2c for several reasons, but previous work can provide some insight into interpretability. While the spread of the embeddings does produce unique values for the fractal dimension for each dataset ($\approx$ 0.017 - 0.02 over all three benchmarks datasets), the meaning behind these values is only understood by examining what the information fractal dimension represents.

In Fig. 4.3 we see an example of two clusters of points, with the complexity calculated (Mass Radius) and listed in their respective subcaptions. It is worth mentioning that the true value of the fractal dimension is $2 - D_M$, as these synthesized datasets are contained in 2-dimensions; therefore their fractal dimension is constrained to $D_M \in (1, 2]$. This minor calculation was excluded from this work as the $2-$ calculation is not important for the purposes of influencing the network through forward and backpropogation. The main takeaway is that the additional *spread* leads to a larger fractal dimension measure, as evident from the fact that including the red points in the calculation increased the value of the measure (1.08/1.04 and 1.4/0.97 for Fig. 4.3a and 4.3b, respectively). And whereby the *spread* is highly non-linear, as shown following an anistrotropic transformation in Fig. 4.3b, the measure is more sensitive to these changes in *spread*. Since the fractal dimension measures the *spread* from the centroid, it follows from the theory and equations that govern the fractal dimension that the Minkowski distance calculation in higher dimensional space is more sensitive to these changes away from the centroid. This is because are raising the difference between the points to the power of the order of norm P in Eq. 3.5. This provides a basis for why the complexity measure has exciting use-cases in the field of anomaly detection as it detects small changes in *spread*; but also in the case of an XGAN where we want to incorporate and relay information about the feature or embedding space into model training.

4.2 Complexity Evaluation on Sandbox Execution Graphs

This section demonstrates the results of a comparative analysis of the application of the complexity layer to a graph neural architecture similar to the previous section. However, in this section the model will now be trained and fine-tuned for performance and generalization on a sandbox malware execution dataset that was described in detail in Section 3.2.2. This represents the core work of this thesis document, and provides a real-world scenario for the application of GATs for malware security, as well as highlighting the benefits offered by the complexity layer to form an XGAN.

First, a brief overview of the malware execution dataset and it's network characteristics is described in Section 4.2.1. Next, fine-tuning the XGAN and its hyperparameters is carried out in Section 4.2.2, followed by Section 4.2.3 which discusses the trained models using different event types representing different behaviours of malware and benignware execution. This will demonstrate how well the feature sets for each event type correspond to malicious behaviour through a trained GNN. Finally, a new series of architectures are introduced in Section 4.2.5 which combines all the event types into a single model by combining the discriminatory power of separately trained



(a) Default cluster, $FD = 1.08/1.04$ (red+green/green)

(b) Anisotropic cluster, $FD = 1.4/0.97$ (red+green/green)

Figure 4.3: Synthesized datasets with anomalous samples introduced (red nodes) representing 5% of the total dataset size ($n = 5000$). (a) original clusters; (b) following an anisotropic transformation $C_{ij}X$ where $C = [[-0.5, 0.5], [-0.5, 1]]$. Illustrations retrieved from previous published work in Brezinski and Ferens, [10].

GNNs into a single, more powerful model.

4.2.1 Node Topology and Feature Characteristics

This section briefly summarizes the structure and attributes of the Sandbox execution graphs, first through through a discussion of network metrics as it pertains to the topology of the sandbox malware execution graphs, followed by a section covering the corpus of API calls developed in this work that act as the feature vectors for each node. A table summarizing the network metrics is shown in Table 4.2. A summary of these network statistics and their meaning follows.

Table 4.2: Summary network statistics for the sandbox malware execution graphs. Values are averaged over all malware executions.

Metrics	**Values**
# Nodes	40
# Edges	112
Avg. Node Degree	6.70
Min Node Degree	2.33
Max. Node Degree	11
Avg. Degree Connectivity	4.75
Degree Centrality	0.14
Degree Assortativity	-0.20
Density	0.07
Square Clustering	0.32

Avg. Node Degree: This is the average degree of a node in the network. The degree of a node is the number of edges connected to it. In this case, the average node has a degree of 6.70, meaning the process topology is highly interconnected.

Min Node Degree: This is the minimum degree of a node in the network. In this case, the minimum degree is 2.33.

Max. Node Degree: This is the maximum degree of a node in the network. In this case, the maximum degree is 11. This node would belong to a process such as `cmd.exe` or `explorer.exe` in the case of a execution initiated by the user using the OS Graphical User Interface (GUI).

Avg. Degree Connectivity: This is the average degree connectivity of a node in the network. Degree connectivity is a measure of how well connected a node is to its neighbors. In this case, the average degree connectivity is 4.75. The degree connectivity of a node is calculated as the number of neighbors of the node with degree greater than or equal to the degree of the node, divided by the total number of neighbors of the node. For example, if a node has four neighbors with degree 2, two neighbors with degree 3, and one neighbor with degree 4, its degree connectivity would be $7/7 = 1$. This is in contrast to Avg. Node Degree, which is simply the sum of the degrees of all nodes in the network, divided by the number of nodes.

Degree Centrality: This is a measure of the importance of a node in the network based on its degree. In this case, the degree centrality is 0.14. Degree centrality can be calculated as the degree of a node divided by the maximum possible degree in the network. For example, in a network with 10 nodes, the maximum possible degree is 9 (since a node cannot be connected to itself). If a node has a degree of 6, its degree centrality would be $6/9 = 0.67$. Given the highly interconnectivity of the graph, a value of 0.14 is fairly high.

Degree Assortativity: This is a measure of the degree homophily in the network. Degree homophily is the tendency of nodes to connect to other nodes with similar degree. In this case, the degree assortativity is -0.20, which means that nodes with high degree are more likely to connect to nodes with low degree, and vice versa. This value would be larger, or even positive, in the case of a larger network topology where servers are communication with each other to form of the backbone of the internet. In the case of process execution, there is no such behaviour to be observed as core OS behaviour tends to spawn daughter processes, but not other core OS processes.

Density: This is a measure of the number of edges in the network relative to the maximum number of edges that could exist. In this case, the density is 0.07, which means that there are relatively few edges in the network. Once process execution is carried out, then it does not make intuitive sense for the daughter process to spawn a process already running. It is even the case that through the use of mutexes processes are not allowed to execute more than once in the OS.

Square Clustering: This is a measure of the clustering coefficient of the network. The clustering coefficient is a measure of the number of triangles in the network. In this case, the square clustering is 0.32, which means that there is a relatively high level of clustering in the network. This follows

from the Degree Connectivity metric that noted the high inter-connectivity of the network.

Understanding the topology of the process execution is important for several reasons. First, process graph topology is fundamentally different than that of a social network or a citation network that were used as benchmark datasets in the previous section. Knowledge of the topology leads to better decision making in the architecture design. For example, if processes on average spawn 1 to 2 daughter processes, then creating a deeper GNN based on 3-hop neighbours (a 3-layer GNN) would be redundant and lead to over saturation - or a smoothing effect over the embedding space.

As an example of this impact, lets denote any ith process p_i^j where j refers to the hierarchy of the process, or depth if one were to consider some form of n-ary tree. If a user uses `explorer.exe`, denoted as p_i^1, to execute a malware executable p_i^2, then we can use GNNs to share the event behaviour of any process $p_i^2 \rightarrow p_i^1$ via message passing. However, the OS parent process that spawned `explorer.exe`, lets call it p^0 which acts as a root node, would not be helpful to understand the behaviour of p_i^2 since there are many processes that are Benign which share an ancestor with p_i^2. As a result, all p_i^2 processes that exist would share the same embedding information from p^0, leading to all the process nodes of p_n^2 becoming saturated with similar information. This would impact performance negatively, and is best resolved through domain knowledge of the dataset used and carefully considering the amount of capacity of your model. A visualization of this effect is shown in Fig. 4.4. We can see the difference in a 1-hop aggregation (Fig. 4.4a) and a 2-hop aggregation (Fig. 4.4b) whereby the malicious red node p_1^2 is smoothed over with information from p^0, leading to a node embedding that is more similar to that of its benign counterpart p_2^2. This is illustrated by the proportion of red, orange and green in its node embedding, which in high-dimensional space would coincide with the vectors p_1^2 and p_2^2 being closer or farther apart in d-dimensional space. Fig. 4.4 displays an example of unidirectional edges, where the embeddings are aggregated from the parent process p^{j+1} but not the daughter process p^{j-1} for any given node p_i^j.

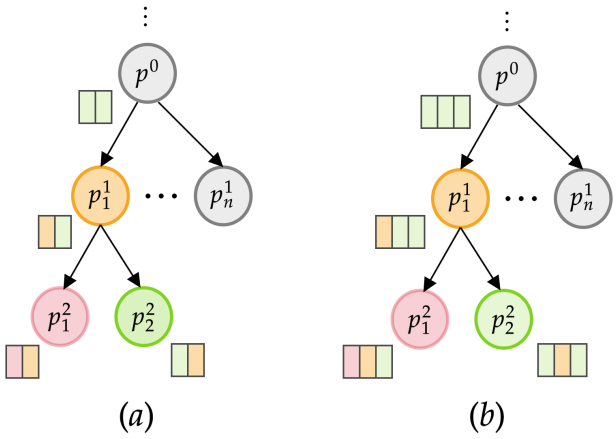


Figure 4.4: Process hierarchy for a directed graph, where the red, orange, and green coloured nodes refer to malicious, suspicious, and benign processes, respectively. Crude illustration of node embeddings after message passing and aggregation for (a) 1-hop neighbourhood and (b) 2-hop neighbourhood.

Additionally, for each feature set belonging to each event type, a corpus is created for the API

calls based on the discussion in Section 3.4.3. In Table 4.3 we can view the size of each corpus, which is the size of the node feature vector described in Fig. 3.8 which is being fed into the model as in input. Additionally, Table 4.3 provides the corpus sizes for 1, 2, and 3-grams - representing all unique combinations of n APIs in sequence (as discussed in Section 3.4.3). What is noteworthy is that the corpus sizes are much larger that previous related work in [3] which recorded corpus sizes in the 1500-1700 range. That previous work only looked at the stack traces of 9 malware executables, while this work looks at over 200; meaning we are covering a larger swathe of potential threats and behaviours in our dataset. The previous work also didn't look at benignware executables which further exemplifies the novelty and the necessity of this work. Looking at a larger pool of potential features has the benefit of encoding more useful information of the behaviour of running processes, at the cost of introducing noise, added dimensionality, and complexity if the features are not represented adequately in the dataset. This will be a topic of exploration in the next section.

Table 4.3: Summary table of the number of API calls belong to the corpus for different n-grams. The **Event Type** *All* refers to a combination of Registry, File, and Thread event APIs into a single corpus.

Event Type	**1-gram**	**2-gram**	**3-gram**
Registry	1917	6517	10,985
File	2017	6018	9669
Thread	612	1506	2200
All	2921	10,656	18,267

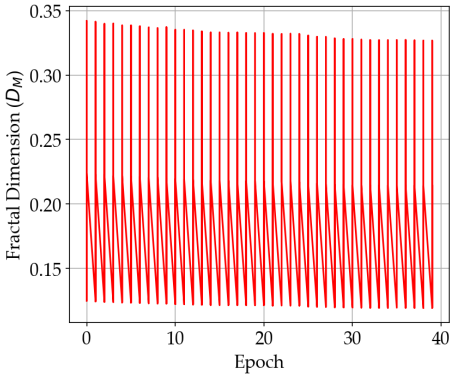
4.2.2 Fractal Dimension, Hyperparameters, and Gradients

In this section the possible configurations for hyperparameter settings and complexity layer positioning will be discussed as it pertains to the learning process through examination of the fractal dimension measure D_M and its weights and gradients. The purpose of this section is to demonstrate the sensitivity of the layer performance to different parameters set for the log-log plot (k), Radius of Gyration (RoG) versus Mass Radius (MR) FD, as well as note other considerations for integrating the complexity layer into the GNN. These considerations were discussed at length in Section 3.5.3 and will covered in this section when applied on the malware dataset.

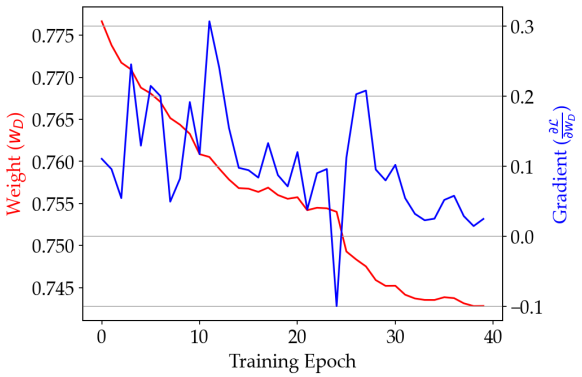
The first consideration for complexity layer integration is the position in which the layer is added to the model. A layer can be placed in a single layer, between each layer, or in none at all (as is the case for the **Vanilla** architecture). In the previous section on benchmark dataset performance, the complexity layer was added as a first layer in the GNN, where it showed a noticeable performance boost to the Validation Loss, Macro-Accuracy and F1 score.

Fig. 4.5 presents an example of the complexity layer implemented in the first layer of a GNN (Figs. 4.5a and 4.5b) in addition to the second layer (Figs. 4.5c and 4.5d). In Fig. 4.5a D_M oscillates between 0.15 and 0.35, where it does not quite reach steady state. More insight can be gathered from Fig. 4.5b where the gradient does reach steady state along with the weight, indicating the layer is converging as the gradients being driven to zero indicates it is finished learning.

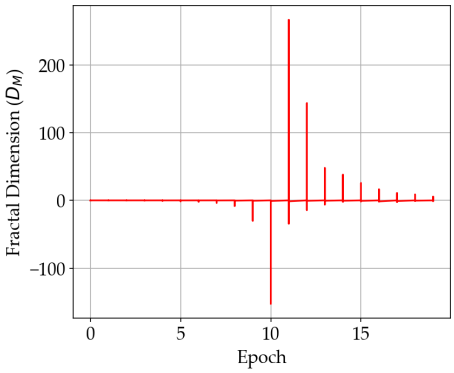
This is contrasted with Fig. 4.5c where the the value for D_M is driven > 2 during select epochs, which is theoretically not possible. This indicates that during the calculation of the log-log plot, one



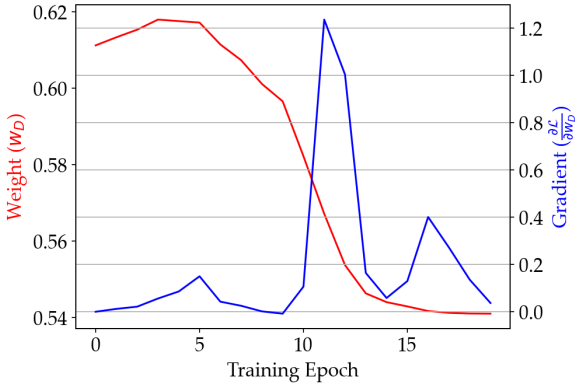
(a) First layer File System FD



(b) First layer File System Weight and Gradient



(c) Second layer File System FD



(d) Second layer File System Weight and Gradient

Figure 4.5: Fractal Dimension measure and learnable weight and gradient plots.

or more points in $\frac{\log(A_k)}{\log(r_k)}$ are outliers and the value for D_M is computed based on the slope of this log-log plot. It may be the case that the value of k is too high for this layer, as computing $\log(A_k)$ based on a subset of points that are too few may led to these erroneous results. For this reason we note poor convergence of the model and the model relying less on the layer over time as noted by the gradient in Fig. 4.5b approaching zero and the weight w_D being driven to 0.0 after 20 epochs.

Based on these results, it is noted that the complexity measure is highly sensitive to the hyperparameters used, and for this reason a comparative analysis was carried out for various hyperparameters ranges. Table 4.4 presents the results of the hyperparameter search for varying values of the number of log-log points k; the order of the norm P; the use of RoG or MR fractal dimension; whether or not M_D was computed pre- or post-activation; as well as the complexity layer architecture (*Arch*). The position of the complexity layer in the network is denoted as *Arch[(pos,)]*, where *(pos,)* is a tuple of the indexed position of the complexity layers in the network; whereby the first layer corresponds to the zeroth position in the network. For example, *Arch[0,1]* would denote a complexity layer located in layers 0 and 1 of the network, and *Arch[1]* would identify a network with the complexity layer in only layer 1. The motivation for testing this placement was discussed at length in Section 3.5.3. All results presented are carried out over a single dataset using **File** event activity and an n-gram of 1 in order to remove the variation between datasets.

Table 4.4: Model Loss and F1-score for various complexity layer hyperparameters chosen. The default dataset used was the **File** event type with an n-gram of 1 and `GAT` layers. Elements **bolded** signify best-performing models, and (n.s) signifies not statistically significantly different (where $p < 0.05$) from their model counterparts according to t-scores (values not shown). Full model architecture is shown in code listing C.2 in Appendix C.

Parameter		**Loss** ↓	**F1 Score** (10^2) ↑
Arch	[0]	38.7 (n.s)	**92.8** (n.s)
	[1]	31.6 (n.s)	90.7
	[0, 1]	76.8	**92.1** (n.s)
FD	MR	54.8 (n.s)	**92.3**
	RoG	54.2 (n.s)	91.0
Activation	Pre	62.4 (n.s)	91.3 (n.s)
	Post	64.8 (n.s)	91.0 (n.s)
k	3	72.5 (n.s)	**92.2**
	4	66.6 (n.s)	91.5 (n.s)
	5	73.8 (n.s)	91.5 (n.s)
P	1	14.4	**92.5**
	2	19.1 (n.s)	91.9 (n.s)
	3	19.5 (n.s)	90.3 (n.s)
	4	15.30 (n.s)	88.8

Based on F1-Score, the complexity architecture where the layer is positioned at the first and second position (zeroth and first array position [0, 1]) performed equally well and within error, meaning the additional layer does not provide any added benefit when included in the network.

This would follow from the explanation provided in Fig. 4.5 where the second complexity layer weight was driven to 0, thereby providing no influence on the model's decision making. It was also the case that the variance for a single layer was lower than that for two layers, therefore there is also a benefit for model stability and model convergence. When considering the use of the RoG FD, it was shown that overall the MR FD outperformed the RoG FD in terms of F1-Score and this was shown to be statistically significant. The variance of the RoG models were also shown to have a 10-fold increase in variance, meaning the computation for the centroid varies widely from model to model, thereby affecting the result for D_M. The consideration for pre- or post-activation computing of M_D was shown to have no effect on model performance. The embedding dimension is already highly non-linear once the feedforward reaches the second complexity layer, therefore a simple linear transformation appears to have no effect on model performance. When considering the value of k, a value of 3 did outperform considering 4 or 5 points on the log-log plot, meaning creating a larger subset of points with a lower k is perhaps more numerically stable than considering more subsets. Variances were all similar however.

Finally, when considering the order of the norm, all values were within error, therefore their is no benefit to considering different distance norms. A value of $P = 1$ did have a smaller variance than the other models, and this can largely attributed to the fact that the Manhattan distance is less sensitive to noise or errors in the data since it is not based on the squares (or powers for higher P) of the differences between the coordinates. Computing the spread of the feature embeddings seems to be highly sensitive to outliers, leading to larger oscillations when a P of 2 is used, for example. To further test this effect, another set of model runs (outlined in code listing C.3 in Appendix C.) were tested with larger hidden layers, with the embedding sizes of the convolution layers as large as 512. Larger variances were noted in these higher dimensions, further exemplifying the sensitivity of the measure at higher d to small changes in the embedding space.

One final examination is in the model Loss reported in Fig. 4.4. Validation Loss is typically used a predictor for model convergence as the Cross-Entropy loss function is directly tied to the computing of model gradients. The early stopping callback is also tied to the model loss, so that the model does not needlessly run when no performance is apparent. What is noted in the model losses in Fig. 4.4 is that many are not statistically significant, meaning the losses from one iteration vary widely, regardless of parameter selection. This makes model loss not a great metric as a surrogate for comparing performance between models, therefore it was excluded from further reporting in the summaru tables. The reasons for these variations are further discussed in Section 4.2.4.

4.2.3 Event Type and n-gram Dataset Performances

Further work was carried out on optimizing performance given the several formulations in datasets and feature representations generated. So far the investigation has been limited to the comparative performance of the complexity layer for a single dataset (**File System** Events) and feature set (n-gram of 1). Node feature vectors, noted in Table 4.3, included three (3) event types: File System, Registry, and Process and Thread activity. One (1) additional dataset was created based on the combination of all three datasets into a combined corpus (labelled as **All**). The investigation will also include different n-grams ranging from 1 to 3, representing the number of APIs analyzed in sequence; again summarized in Table 4.3. Therefore, twelve (12) datasets were prepared and compared based on their validation performance.

In Fig. 4.5 we can see the final model Macro-Accuracy and F1 scores for different $n$gram and event types. The parameters kept constant in these experiments are noted in the caption of Fig. 4.5. When considering n-grams, all models performed equally well within error. There was no statistically significant difference between n-gram selection alone, but an n-gram of 1 does provide some worthwhile benefits. First, the reduction in n from 2 to 1 corresponds to a reduction in the input feature vector size by approximately 3-fold as noted in Table 4.3, leading to a smaller model and faster training time. Additionally, a unigram is much more interpretable and easier to compute at inference time due to the smaller input dimension.

Table 4.5: Comparison Macro-Accuracy and F1 Scores between Vanilla and MS implementations for varying dataset configurations. When any given dataset parameter was not varied, the default dataset used the **File** event type with an $n$gram of 1 and the MR complexity measure. Elements **bolded** signify best-performing models based on datasets, and (n.s) signifies not statistically significantly different (where $p < 0.05$) from their model counterparts according to t-scores (values not shown). Full model architecture is shown in code listing C.4 in Appendix C.

Parameter		**Accuracy (%)** ↑	**F1 Score** (10^2) ↑
1-gram	Vanilla	88.5	88.6
	MR	92.2	93.0
2-gram	Vanilla	88.9	91.8 (n.s)
	MR	92.4	92.8 (n.s)
3-gram	Vanilla	87.4	88.8
	MR	**92.7**	**94.2**
File	Vanilla	88.5	88.7
	MR	**92.2**	**92.1**
Registry	Vanilla	79.2	84.4
	MR	90.7	90.6
Thread	Vanilla	88.0 (n.s)	91.0 (n.s)
	MR	90.8 (n.s)	91.2 (n.s)
All	Vanilla	76.4 (n.s)	68.6 (n.s)
	MR	77.2 (n.s)	70.1 (n.s)

The benefit in considering larger value of n could also indicate an issue with under-training, as bigrams and trigrams are much harder for the model to learn from since you are limited in the number of APIs that are available in the training set. For example, while API_i may appear 100 times in the dataset, the sequences $(API_i,\ API_j)$ in the case of a bigram or (API_i, API_j, API_k) in the case of a trigram may appear only three or less times. The number three (3) is hard-coded during preprocessing in Section 3.4.3 as the minimum number of times an API has to be present in any part of the dataset to be considered part of the corpus. This is due to the fact that the model will not have the ability to learn from a feature when it is present only once or twice in the entire dataset. If an API is present only once during training, it will have no effect on model performance on a validation set where it is not present at all. And if it is in the validation set, the model will consider the API as a part of its out-of-vocabulary set since it was not present during training time. Also, when tf-idf is used, the APIs which would otherwise be promoted by tf-idf due to contributions being inversely proportional to frequency, are removed from the training set. This

is a necessary trade off when carefully crafting a feature vectorizer that must respect the limitations of model training and generalization to an unseen validation set.

File activity was also noted as the best performing event type. Based on the corpus size, that would correspond to only 2017 APIs needing to considered. One of the rationales for the improved performance with File activity is that there is overlap in the API usage of malware and benignware for both Thread and Registry event activity; which makes the decision boundary for the classification less clear cut. Section 4.2.7 will investigate this API usage in more detail and provide concrete API signatures for each event type. Since tf-idf was used, this does indicate File system uses APIs that are more unique to malware than otherwise. Previous work in Brezinski and Ferens, [3] did indicate that based on frequency of API usage, malware uses File events APIs five times more than Thread activity (217 versus 45, respectively [3]). Larger stack traces correspond to longer API sequences, meaning more unique patterns for the model to learn from. This is akin to a NLP approach where a model has a single web-page to learn from versus the entire contents of Wikipedia. It is also worth noting that the corpus containing all event types (labelled as **All**) performed much poorer across the board. The reason for this shortcoming is the under-fitting that was noted earlier when considering higher values of n. Except this problem is exacerbated when considering a corpus size close to 3,000, many of which contain APIs that are only scarcely present in the dataset.

When examining the use of the complexity measure, the MR outperformed the Vanilla implementation across the board - with the exception of the Thread event type dataset where it was within error. Additionally, the variance in model performance for all models based on F1-score was 2-3 times better. This follows from the plots generated for the benchmark datasets in Fig. 4.1 which indicated a tighter confidence interval for the Accuracy and F1-Scores. It was also noted (not shown) that the RoG measure did not have this benefit, meaning the re-computation of the centroid carried out does not improve performance.

Based on previous investigations in Brezinski and Ferens, [1] on a dataset without benignware executed, but solely classifying malware from OS background behavior, the best performing model was also the File System events, unigram, and a 2-layer GNN. This confirms that the APIs used by the file system offer an adequate means of distinguishing malware from benignware. One additional point to mention is the discrepancy between F1-Score and Macro-Accuracy. The F1-Score is inherently skewed higher since F1-Score does not account for True Negatives (TN); meaning its a better metric for classifying malicious samples, but a poorer metric for monitoring benignware. Macro-Accuracy takes TN into account, but it tends to skew once the performance of the minority class - in this case malicious processes - improves. F1-Score tends to be recorded for both binary and multi-classification problems in the field of DL, so it is worthwhile to report both for completeness. In cases where their are large discrepancies between Accuracy and F1-Score, such as for the 3-gram dataset, this indicates difficulty in classifying benignware as the TN class. Macro-Accuracy in this case accounts for this discrepancy.

One other observation can be found from looking at the top performing models, with the top 10 models according to Macro-Accuracy shown in Table 4.6. This table serves to demonstrate that only a single Vanilla model is present in the top 10 models trained, and three are found in the top 20. Overall this demonstrates the improved performance in training model consistency when using the complexity layer. One other note is that while certain models, like Registry event type, appear multiple times in the top-10, overall the dataset performed poorer overall as per the results shown

in Table 4.5. This indicates that during select models that were trained, the presence or absence of certain difficult to classify samples were present in the validation set. So while an individual model may by chance show a high accuracy, overall the dataset performs poorer when ran for multiple iterations. These shortfalls in regards to randomization in the validation selection are described in the next section.

Table 4.6: Comparison Macro-Accuracy scores for the top 10 performing models ranked by accuracy. Full model architecture is shown in code listing C.4 in Appendix C.

Event Type	**Model**	**Accuracy (%) ↑**
Reg	MR	98.7
File	Vanilla	97.1
Thread	MR	97.0
Reg	MR	96.9
File	MR	96.7
Reg	MR	96.7
Reg	MR	96.7
Reg	MR	96.4
File	MR	96.3
Thread	MR	96.0

4.2.4 Model Performance Variance

One takeaway from the model performances is that model performances have larger than expected variances between model runs. This is evident in the high standard deviations in model Loss, Macro-Accuracies and F1 Scores shown thus far in this Section. The reason for this is due to the sampling procedure used and some of the considerations touched on in Section 3.2.4.

Some of the process execution graphs are populated with benignware activity that is either executed more than once in two different graphs, and some executables which are not present at all. This has the effect of the model learning very different execution graphs which adds to the robustness of the model but also leads to large variations between execution graphs which leads to larger than expected convergence times. If it were the case than a very difficult to classify benignware process is selected to be executed, then for those nodes in the execution graph the model would perform poorly leading to poorer F1-Score and Loss. It is always the case in DL research that edge-cases are some of the more problematic cases to classify, and in this work the dataset is small enough where edge-cases are not adequately covered. The same case can be made for difficult to classify malware samples, so the same thought process applies.

Secondly, the largest source of model variance is the selection of the training and validation set. A visualization of this effect is shown in Fig. 4.6. To overcome effects due to sampling bias (i.e. the manual selection of samples to validate on which introduces bias), the positive and negative training examples that fall into the validation set are randomly selected for each of the iterations that a new model is trained. For this work Neighborhood Sampling was employed, meaning neighborhoods are sampled and their connected edges remain intact during samples. This is in contrast to Random

Sampling methods, where nodes and edges would be randomly sampled in the graph, leading to poorer performance as a lot of the neighborhood structure would be lost if a malicious parent executable is not present in the training set but the daughter process is. This would have the effect of the learned embeddings of the daughter process not being present in the same training/validation set as the parent, which would not be a good approach to sampling a graph topology.

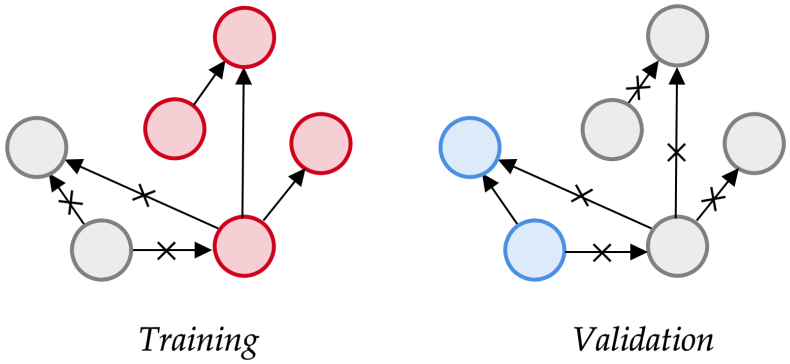


Figure 4.6: Training and Validation sets for nodes sampled using Neighborhood Sampling at the beginning of each iteration.

It is worth mentioning that iterations are different than epochs: where at each iteration a new model is initialized with random weights, a randomized training and validation set, and a new random seed; whereas during each epoch the model is trained and validated on the same model and training and validation set (to reach model convergence). So for each iteration the model experience is very different and this can lead to significant model variation. This is due to the fact that, depending on the iteration, if a hard to classify malicious executable is presented in the validation set, it would not have been trained on in the training set, and thus be harder to classify. In another iteration it would be present in the training set, in which case it would not belong to the validation set and the performance scores would improve. Repeated iterations tend to smooth over this effect as the effect averages out, but this presents a persistent effect due to the nature of sampling over 200 independent process execution graphs with heterogeneous topologies.

There are several remedies for these effects: one of which will be tackled in the next section when using ensemble methods. This work already applies Regularization and Early Stopping to ensure the validation performance is as close as possible to the training performance to avoid overfitting [450, 451]. One other remedy is to increase the size of the dataset which is always a solution to problems where overfitting leads to poorer generalization on the validation set in DL research. But largely due to the time commitment, and in other cases the cost of manually labelling training examples, this is infeasible and impractical in practice [452, 453, 454].

4.2.5 Graph Ensemble Methods

In the previous section GNNs were trained separately on Windows event APIs in order to note the benefits of relying solely on the Window events activity from either Registry, File System or Thread activity. In this section a new series of architectures will combine the discriminatory power of these models and event datasets into a single pipeline in order to leverage the features of all event types simultaneously. There are several approaches to creating an ensemble method. In summary, three

(3) types of ensemble methods are evaluated in this chapter.

Feature Concatenation

One approach is to combine the feature vectors of each of the event types, and train a single, larger GNN on this new feature matrix. This method was tried and tested in previous sections when the Event Type was set to **All**, where it was noted performance did not improve when combining all the event types into a single corpus to be fed into the network. While this approach would combine the information of all event types into one, the message passing mechanism of the GNN layers would have no way of knowing which APIs belong to which event type, and pass them all indiscriminately and with similar importance via the weight feature matrix $\mathbf{W}^{m \times E}$. It is also the case that with larger input feature dimensions, the model has a tendency to over-fit on the training examples and perform sub-optimally on the validation set. More formally, the node feature vector $\bar{h}_i$ for node i is the aggregation of all event types which are aggregated according to Eq. 4.1. The model proceeds as normal, generating a single set of embeddings over all nodes and events types.

$$\bar{h}_i^{'} = CONCAT(\{h_{i,t}, \forall t \in T\}) \tag{4.1}$$

Embedding Concatenation

Another approach would be to combine the final embeddings of each GNN trained on each event type independently, and then train a final discriminator on their combined embeddings. This has the benefit of using separate optimizer for each event type, and concatenating the learned embeddings into one to be used in a final classifier. This method does away with the feature concatenation discussed previously, and simply concatenates the model's output embeddings after they are already learned separately. This method preserves the expressiveness of each individual event type trained, but combines the collected knowledge of their final embeddings into a final discriminator. The only additional detail would be fine-tuning a final layer after the embeddings are combined.

A graphical illustration of this approach is shown in Fig. 4.7. This new architecture takes in input the final embeddings of each trained neural network $\bar{h}_t^{'}$, and trains a final Linear layer on these embeddings. Before this layer a concatenation and normalization layer is added as shown in Fig. 4.7. First, concatenation combines all model embeddings $\|_{i=1}^{N} \bar{h}_i^{'}$ so that each process node has the correct vector corresponding to the embedding of the different event types. The normalization layer implements a normalization similar to batch normalization, with the difference being the normalization is carried out for each embedding rather than for each batch. The correction made to each embedding dimension E is shown in the series of equations in Eq. 4.2. Normalization minimizes the variation between embeddings and ensures the optimizer converges to an optimal global minima by normalizing the gradients that are produced as well [455, 456]. In this fashion $\bar{h}^{'}$ is modified through normalization, and the final feedforward layer aids in the final classification.

$$\begin{aligned} \mu_i &\leftarrow \sum_{i=1}^{T} \bar{h}_i^{'} \\ \sigma_i^2 &\leftarrow \frac{1}{T} \sum_{i=1}^{T} (\bar{h}_i^{'} - \mu_i)^2 \\ h_i^{'} &\leftarrow \frac{h_i^{'} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \end{aligned} \tag{4.2}$$

Where μ and σ are the mean and standard deviations, respectively, and ϵ is some small constant $(1e-14)$ to prevent a division by zero error. The final embedding, denoted as $h_i^{''}$ in Eq. 4.3, is generated from the normalized embedding layer after being passed through a $Feedforward$ block that can include one or more Linear layers followed by a non-linearity.

$$\begin{aligned} \bar{h}^{'} &\leftarrow CONCAT(\{\bar{h}_t^{'}, \forall t \in T\}) \\ \bar{h}^{''} &\leftarrow FeedForward(\bar{h}^{'}) \end{aligned} \tag{4.3}$$

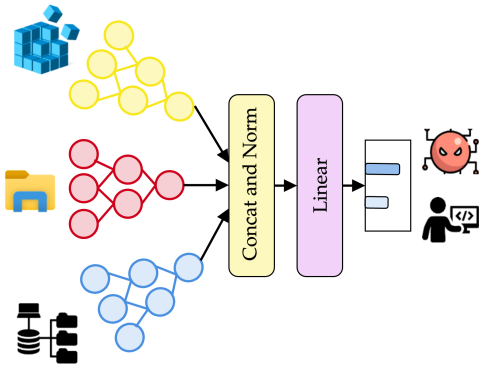


Figure 4.7: Illustration of embedding concatenation which combines all the separate model embeddings and concatenates them for further downstream processing by one or more Feedforward layers.

Heterogeneous Graphs

A potentially more expressive approach would be to allow the model to learn each event type separately through a unique feature matrix $\mathbf{W}_t^{m \times E}$ where t corresponds to the event type, of which there are three (3) for File System, Registry, and Thread. As a result, the model architecture becomes a heterogeneous graph, with T different types of edge connections between process nodes. So rather than a graph being represented by the set of edges $\mathcal{E}$, now there are a set of edges for each event type $\mathcal{E}_t$. This way each event type is treated uniquely and prevents over-smoothing of the node embeddings. The only change in terms of notation is that there is a unique edge list for each event type, now denoted as $\mathcal{E}_t^{2 \times |\mathcal{E}|}$. The message passing and neighborhood aggregation is a trivial change to the existing computation, with only a small change to account for the feature matrix $\mathbf{W}_t^{m \times E}$ and node feature vector $\bar{h}_{i,t}$ to account for the edge type t as shown in Eq. 4.4.

$$\bar{h}'_{i,t} = \sigma\left(\sum_{j \in \mathcal{N}_i} a_{ij} \mathbf{W_t} \bar{h}_{j,t}\right) \tag{4.4}$$

The event types are considered independent connections in a single graph, and the node embeddings are updated at each layer. This preserves expressiveness in a single graph, and allows the computation message passing of the GNN to be conditioned on the edge type t - all the while using a single optimizer and loss function for back-propagation. A visualization of this approach is shown in Fig. 4.8. Put simply, while Eq. 4.1 concatenates the node feature vectors and used the same weight matrix, a heterogeneous graph keeps the node feature vectors separate, and uses different weight matrices to develop the model encoder which ultimately generates the embeddings of the model. The implementation is a little less straightforward in practice, and includes duplicating existing message passing layers and creating connections between the convolution layers. This will be described in the next section when discussing performance and model capacity.

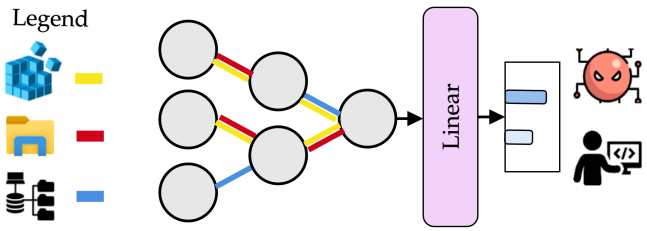


Figure 4.8: Illustration of a heterogeneous graph approach which carried out feature aggregation and message passing in tandem for each edge type.

4.2.6 Graph Ensemble Model Performance

Based on the discussion of ensemble models in the previous Section 4.2.5, a summary of the results are shown in Table 4.7. Similar to previous investigations, the Feature Concatenation architecture has troubles generalizing based on the representation of the corpus and the prevalence of select APIs. Both the other architectures train on the event types separately, but do so in slightly different ways within a single framework. Results indicate the Heterogenous architecture outperforms the others in both Accuracy and F1 Score, and within error when considering the MR model between architectures (95.4 and 96.4 for Embedding Concat and Hetero GNN, respectively). Results also demonstrate broadly similar performance between Embedding Concat and Hetero GNN, which raises the question if there are alternative benefits to developing the models.

This brings the discussion to comparisons in model capacity which are also tabulated in Table 4.7. This section will provide a brief overview of the differences in complexity, but a thorough deep-dive explanation can be found in Appendix E. The three (3) ensemble models have very different space complexities based on their architectures and configurations.

The feature concatenation architecture, which is simply a Vanilla GNN, is a product of the input feature dimensions obtained from the corpus m, the size of the embedding layer E, the number of layers l, and the number of nodes N. A heterogenous graph duplicates the layers for each edge type, and creates an intermediary layer to provide a flow of gradients towards the other edge types. Fig.

Table 4.7: Model Macro-Accuracy and F1-score for several ensemble methods with space complexities ($\mathcal{O}$) described in Section 4.2.5. The default dataset used was the **File** event type with an n-gram of 1. Elements **bolded** signify best-performing models, and (n.s) signifies not statistically significantly different (where $p < 0.05$) from their model counterparts according to t-scores (values not shown). Full model architecture is shown in code listing C.6 and C.5 in Appendix C.

Model	Space Complexity		Accuracy (%) ↑	F1 Score (10^2) ↑
Feature Concat	$\mathcal{O}(E \cdot [m + \{l-1\} \cdot E + 2] + l \cdot N^2)$	Vanilla	76.4	81.6 (n.s)
		MR	77.2	81.4 (n.s)
Embedding Concat	$\mathcal{O}(T \cdot (E \cdot [m + \{l-2\} \cdot E] + l \cdot N^2 + E \cdot 2))$	Vanilla	94.7	97.1 (n.s)
		MR	95.4	96.7 (n.s)
Hetero GNN	$\mathcal{O}(T \cdot E \cdot [m + \{l-1\} \cdot E + 2] + l \cdot N^2)$	Vanilla	94.2	96.2
		MR	**96.4**	**98.8**

4.9 provides an illustration of this behavior. The illustration scales to any number of different edge types T, with the shared layer, in this case `GATConv`. What is clear is that the flow of gradients is allowed through both branches of $\bar{h}'_{file}$ by re-routing the output of the `GATConv` into the *ReLU* activations of the other branches.

In Table 4.7 it is noted the complexity scales with the number of event types T for both heterogenous and feature embedding frameworks, therefore more capacity is required to support these. The Embedding Concatenation framework also scales with the edge type T but does so less efficiently since it re-computes attention coefficients ($\mathcal{O}(l \cdot N^2)$) for each edge type. More on this proof in Appendix E. Overall, asymptotically the Heterogenous framework is smaller in terms of space complexity than simply concatenating the embeddings, while the Feature Concatenation architecture is the smallest overall.

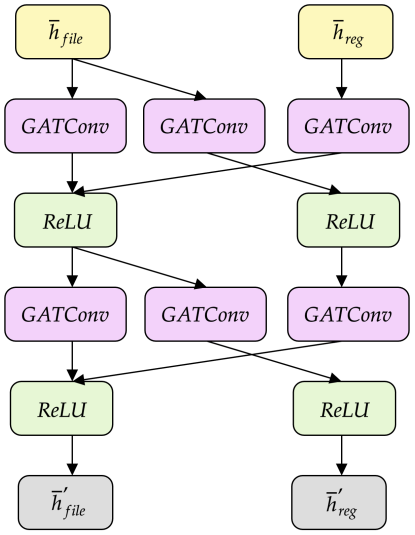


Figure 4.9: Illustration of a heterogeneous graph block diagram for the duplication of the message passing layers. Adopted from the official Pytorch Geometric documentation found here.

4.2.7 Graph Feature Importance and Signatures

Based on the experimental results generated thus far, it has been demonstrated that models can generalize well to never-before-seen examples of malicious behavior - at least based on the dataset generated in this work. One advantage of training these models is being able to use them to draw inference to make predictions towards future behaviors, but also to identify which features are the strongest indicators of malicious behaviors, which in this case are sequences of APIs. These signatures can be used to improve the products of HIDs, or be implemented into new YARA rules [457]. An example YARA signature developed by the SANS Technological Institute for adding known malicious APIs can be found here.

Additional investigations have been carried out to examine these trained GNN models and uncover the most impactful API sequences towards the models output. One of such tools is GNNExplainer, developed in [419] and [411], which provides insights into the importance of nodes and features towards the predictions made by a trained model. GNNExplainer operates by providing explanations for the predictions made by GNNs in two main steps: edge masking and message-passing. In the edge masking step, GNNExplainer systematically masks edges in the input graph and observes the changes in the model's predictions. By iteratively masking different edges, the model identifies the most influential edges that significantly impact the model's output. For example, if a process p_i is a malicious process and has its edges masked with another malicious daughter process p_{i+1}, one would expect the edge between these two to have a great influence on the malicious scoring of p_i. Theoretically, both these nodes contributed to the final embeddings and the malicious score of p_i, therefore by pruning one or more neighboring nodes $\mathcal{N}$ one can have an idea on node importance.

The second step involves a message-passing process, where GNNExplainer computes relevance scores for each node based on how important they are in the context of the masked edges. This process helps to identify the key nodes contributing to the prediction by also masking the features of nodes, thereby providing an indication of feature importance. For example, if API_i (or a series of features in bigram or trigram) contributes greatly to the maliciousness of node p_i, one would expect the removal of this feature to sharply affect the maliciousness scoring. The model does this by masking the feature and tracing the output in the feedforward step. A visualization of this process is shown in Fig. 4.10.

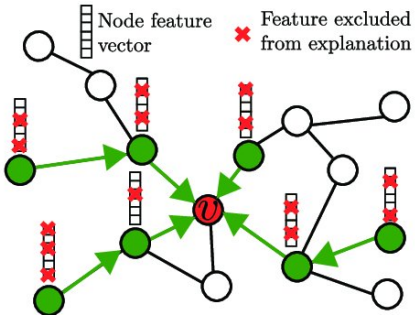


Figure 4.10: A sample illustration for identifying the subgraph features and nodes used by GNN Explainer to generate explanations for graph predictions. Retrieved from [419].

By combining the information from both steps, GNNExplainer generates an understandable explanation of the GNNs decision-making process, revealing the most critical features and connections

in the graph that lead to a particular prediction. GNNExplainer was carried out on the best-trained models for each n-gram tested and each Event Type. This would highlight the importance of APIs toward scoring maliciousness, and doing so for each event type. This would generate a table very similar to Table 2.11, but include additional details regarding *how* important APIs are relative to one another. This table is shown in Table 4.8 and organized by Event Type, and includes an assessment by the author of the malicious techniques along with their MITRE Att&CK tag. The ordering in the table is scored across all malicious executables that appeared in the validation set. The MITRE Att&ck techniques listed are only those presumed to be associated with the behavior, and would require further confirmation by Security Researchers. The API sequences listed in Table 4.8 are also not exhaustive, and are the highest scored for the purposes of cross-referencing with potential MITRE Att&ck techniques and for further deep dive. Other API sequences were scored by XGAN, but the were either a much lower contribution to malicious scoring than the listed APIs, or they were singular unigrams that are already part of the API sequences shown in Table .

Table 4.8: Compiled list of API subsequences scored by XGAN and GNNExplainer as being key indicators for maliciousness based on their subgraph.

Event Type	API Sequence	MITRE Technique	MITRE ID
Thread	`OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`	DLL Injection	T1055.001
	`NtCreateThreadEx`, `NtAllocateVirtualMemory`, `NtClose`	Thread Injection	T1055.003
	`FreeEnvironmentStringsW`, `GetConsoleMode`, `GetEnvironmentStringsW`	Impair Defences, Indicator Removal, Highjack Execution Path	T1562.003, T1070.003, T1574.007
	`CreateSemaphoreA`, `CreateToolhelp32Snapshot`	Process Discover	T1057
	`TlsAlloc`, `TlsFree`, `TlsGetValue`	Thread Local Storage	T1055.005
File	`NtCreateFileA`, `NtCreateProcessA`, `FindResourceA` `LoadResource`, `LockResource`, `SizeOfResource`	Process Injection	T1055.009
	`CopyFileA`, `DeleteFileA`, `CloseHandle`	Data Obfuscation	T1001.001
	`CreateProcess`, `NtUnmapViewOfSection`, `WriteProcessMemory`	Process Hollowing	T10565.012
Registry	`getSystemDirectory`, `RegQueryValueExA`	Registry Modification	T1112
	`RegCreateKey`, `RegCreateKey`	Registry Modification	T1112
	`RegOpenKeyExW`, `RegQueryValueExW`	Registry Modification	T1112
	`RegEnumKeyExW`, `RegisterShimImplCallback`	Application Shimming	T1546.011

There are a few noteworthy differences between the table shown in Table 4.8, and that compiled from the literature in the form of Table 2.11. Table 2.11 does not include information related to sequencing, and places no importance on the relative importance of API usage in detecting maliciousness. Table 4.7 contains sequencing in the form of n-grams, and does so ranked by importance. So while typically YARA signatures are blanket signatures for any binary analyzed, the signatures developed in this work are prioritized for researchers to examine and make use of in their analysis. Secondly, sequencing in the literature, such as that presented in the form of LCS vectorizers, is based on common APIs found in malicious vs benignware binaries. In this form LCS simply creates its own signatures based on subsequences, and contains no information about how that sequence is useful for detecting maliciousness, besides simply using it as a feature in a classifier and scoring the feature that way. Therefore, the work presented in Table 4.8 presents a novel contribution to the research community in the form of a series of tangible signatures to make use of and incorporate into rules.

4.2.8 Analysis of XGAN Malicious Behavior

This section will provide a more thorough examination of the APIs identified in Table 4.8. One takeway from the APIs listed that APIs that are synonymous with each other by being functionally equivalent are present. For example, the API `NtAllocateVirtualMemory` is used to inject payloads into remote processes, and is very commonly used as a replacement for `VirtualAllocEx` [458]. We see both APIs used as a part of a malicious sequence for Thread activity in Table 4.8. The highest scores API sequence included (`OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`) which is widely known as a precursor to DLL injection using CreateRemoteThread via MITRE technique T1055. Following from DLL injection is Thread injection in Table 4.8, which is a widely used method by malware to inject itself into the memory of a running host. These two forms of injection are commonplace and are well known to the security research community.

Another familiar sequence is `CopyFileA`, `DeleteFileA`, `CloseHandle` which was identified in previous work by the author in [11] and noted in [338]. The use of `CloseHandle` is used by anti-emulation malware as a form of obfuscation via junk code insertion. If a process being debugged tries to close an invalid name (which it generates using random values for "hObject") it generates a `STATUS_INVALID_NAME` error. Additionally, `LoadResource`, `LockResource`, `SizeOfResource` are used for memory mapping for resource section handling. This is typically used to avoid writing to disk by loading an encrypted PE file from the resource section, and mapping the file into the hijacked process' own memory to dynamically resolve it.

Another noteworthy sequence was the use of `GetEnvironmentStringsW`. While this was a single API, it was scored the highest among unigram features by XGAN. While the use of `GetEnvironmentStringsW` varies, it can be associated with the enumeration of the environmental variables on the host OS. This can be part of anti-emulation behavior, or alternatively, to hijack the execution path by placing the malicious executable earlier in the list of directories stored in the `PATH` listing, thereby executing with priority over the legitimate binary. Other API sequences that are not attributed to a MITRE techniques in Table 4.8 were not found to have a clear and convincing behavior attributed to them. While the individual APIs may be used in a malicious context, as a sequence this was not found to be the case. It may be the case that these sequences are part of a broader behavior that is ill-defined, but this remains to be investigated.

Investigation into the Registry APIs provided less of an explanation of behavior than the other event types. Tracing Registry API sequences did not result in clear cut behaviors, but rather, fell under the generalized behavior of Registry modification T1112. Many of the APIs used, such as `RegQuery*`, `RegOpen*` and `RegCreate*` all fall under the category of registry modification. This may be one explanation for why registry activity was the poorest performing event type as many of its behaviors are general. The one exception was the `RegisterShimImplCallback` API which may be attributed to Application shimming. This is a vulnerability which exploits backwards compatibility in Windows OS, whereby an application shim is used to act as an intermediary between the IAT and the OS. This allows for legacy code to be written that is interoperable with newer OS. The list of shims are found in registry locations `hklm\software\microsoft\windows` and `hklm\software\microsoft\windowsnt\currentversion\appcompatflags\custom` which helps to explain the connection to Registry activity. While this investigation has so far been limited to examination of the signatures found in the executables analyzed in this work, the next section will present an analysis of a series of unknown binaries following XGAN.

4.2.9 Analysis of Unknown Binaries using XGAN

It is also worth discussing potential limitations to the signatures generated in Table 4.8. XGAN was run on a series of unknown binaries, that were subsequently subject to further investigation using the help of VirusTotal to corroborate the findings following XGAN and GNNExplainer. When analyzing `trojan/fsysna/dvqw` (MD5: `6c1bc15e883088990d00f9debbde9575`) this binary mainly makes use of Visual basic functions imported from `MSVBVM60.DLL`. Since none of these APIs appear in the shortlist in Table 4.8, it indicates these may not be present in the corpus, and may have not been present in this dataset. This was found to be the case as the binary was not flagged as malicious by XGAN and many of the APIs used by `trojan/fsysna/dvq` were not found in the corpus list.

In another binary, `trojan/mptamperpshell/powershell` (MD5: `81027b6ce032a089d280567c5cd3ef8b`) makes use of imports from over 36 DLLs. The XGAN architecture was able to flag this as malicious while only 3 out of 60 AV scans identifying were able to flag binary. Additionally, no commercial sandboxes (Microsoft SysInternals, CAPA, C2AE, VirusTotal Jujubox and VirusTotal Observer) were able to flag this binary as of the writing of this document according to Virustotal. GNNExplainer identified that Registry activity accounted for the greatest signal for maliciousness, with over 30 registry keys set and over 300 registry keys being opened. This coincided with the threat MITRE threat technique for Privilege Escalation T1548 based on the registry keys altered. The binary also opened various processes, mainly `cmd.exe`, `conhost.exe`, `fltMX.eve`, `net.exe` and several calls to `reg.exe` containing registry key additions for privilege escalation. While this process spawn behavior was captured, several processes were terminated, which the sandbox did not account for (only the APIs used in their termination). The sandbox does not account for processes terminated in the process graph as it is not generated dynamically. This presents one future of this work to expand the XGAN capabilities to dynamic graphs, where the model is trained on fed graphs at various timesteps of execution rather than all at once.

What is unique about the results presented in the work of this thesis document it is the first examination of API sequence performance generated via a GNN. In Section 2.5.3 there was a thorough discussion on the many ways in which researchers develop API signature via frequency and subsequence analysis, but this work is the first examination which incorporates the information obtained

from neighboring nodes as well. For example, in [459] researchers included network traffic, and used longest API sequences generated using the LCS algorithm. Their longest common sub-sequence algorithm generated API subsequences, but only based on the presence of these subsequences for malware families based on frequency. If a subsequence was present in many malware variants, it was considered a signature for malware. The sequences generated by the Sandy framework work act as more generalized API subsequences since they capture the behavior of malicious activity outside of a single process since the scoring of maliciousness is based on the scoring of nearby nodes.

It was also found that the APIs of interest did not differ between the complexity layer and the vanilla architectures. The examination of feature concatenation using the dataset **All** was also excluded from the reporting since the generalization performance was poorer and the information for the important events are already incorporated into Table 4.8. Additionally, there was no statistical difference between the API scoring when investigating the different ensemble architectures (shown in Table 4.7). This indicates that these architectures do a better job as encoders of the malicious behavior via improved model convergence, but they do not generate any additional information about malicious API sequences. It was also found that examining the imported DLLs of the processes looses a lot of the discriminating behavior of the process, as many processes - both benign and malicious - use `ws2_32.dll`, `winmm.dll` and `KERNEL32.DLL` as an example.

Chapter 5

Ongoing and Future Work

5.1 Conclusions

This work makes the following core contributions

- A Malware collection sandbox was developed which can execute and trace malicious behavior as it is executed on a host OS. For this, over 200 malicious executable were run and tracked alongside benignware. The stack traces of each and every running process was traced along with their process spawn behavior. Process events based on Registry, File System and Thread activity was traced through their API usage;
- Two (2) complexity technique were envisioned and developed based on the concept of informational complexity. This included the Mass Radius and Radius of Gyration Fractal Dimension (FD). These measures were incorporated into a Deep Learning framework and developed as layers which can participate in forward and back-propogation in a DL architecture. When incorporated into a graph-based DL architecture this new network is coined an XGAN;
- Two (2) sets of datasets were developed for the purposes of testing the FD measures: a benchmark dataset based on real world applications including the SiteSeer, Facebook Page, and Cora dataset. The second set of datasets included the malware behavior collected in the sandbox noted previously;
- Experimentation on the benchmark datasets demonstrated improved convergence of the models when a complexity layer was added. Investigation also looked into the various hyperparameters that are suitable for the complexity layer to operate with. It was noted the Mass Radius FD measure outperformed the Radius of Gyration FD on measures of Accuracy and F1 Score.
- The XGAN model was run over several model iterations on a variety of datasets based on different Event Types and n-gram of the API sequence. Models were compared to one-another, and the XGAN was found to converge better with better variance than its Vanilla counterpart;
- When testing various datasets, it was found that shorter API sequences and the File event type corresponded with the greatest indicators of maliciousness.

- This was further examined using GNNExplainer, a technique which uses node and feature masking to resolve the subgraph which best explains the model predictions. Several API sequences were examined, and attributed to specific MITRE techniques;
- Investigation into unknown binaries highlighted shortcomings of the methods, but also demonstrated an example of XGAN flagging an executable which was not associated with maliciousness according to VirusTotal and the built in commercial malware honeypots.

5.2 Engineering Significance

There are two main ways in which the completed work provides beneficial contributions to the engineering community. The first is through the Complexity measure and the ways in which it can be integrated within existing DL architectures and applications. And secondly, through the contribution to the security research community in developing a novel detector which can alert to malicious executables and their metamorphic variants.

The Complexity measure, which was integrated in this work as a learnable model layer, has several potential uses in DL research. It was demonstrated in this work that earlier convergence and increased classification score (measure via Macro-Accuracy and F1 score in this work) leads to better performance when tested on several benchmark problems and a real malware dataset. While this work was limited to applications in GNNs, it has been shown to be an effective measure in simple ANNs [438] and potentially other frameworks. The concept of complexity and self-similarity are powerful tools when one wants to investigate the behavior of a self-similar or self-affine signal. The topic of topological complexity was suitable for measuring the complexity of high-dimensional embedding spaces, and through this application, it was shown to have a use-case in Deep Learning for Malware Detection. The work carried out in this thesis document serves as one application, and through further research in this space, the applications can broaden to other domains where informational complexity is suitable.

As for the application towards Malware Research, there are plenty of suitable applications for detectors in an HIDS, and more broadly for any SIEM system. The models developed in this work can serve a single model, or serve as a signal for an ensemble model as a part of a larger detection system. As Cyber Security becomes an ever-growing concern for individuals and enterprises alike, the tools required have to follow suite. Many popular security deployments, such as Windows Defender for Endpoint, include host-based security as a top-priority, usually in tandem with other sources of telemetry from the network or from other machines on the network. It is also the case that the work covered in this document covers many MITRE ATT&CK indicators including Indicator Removal on Host (T0872), Exploitation for Privilege Escalation (T0890) and many of the techniques covered in Persistence tactics (TA0110). Therefore, there is space for growth in this field of research as the tactics change over time. While the models trained in this work are specific to the dataset provided, the model training pipeline is still applicable no matter what the application is. As long as there is a mechanism for feature vectorization, the model pipeline can generate detectors necessary to perform classification on the graph framework.

5.3 Limitations of this Work

This section outlines some of the limitations of this work as it pertains to methodology, approaches, and results. It is important to outline these shortfalls as it ensures the relevance of the research is constrained to the domains in which it suitable, and the situations in which it is best served.

First and foremost the findings of this research are limited to Windows OS and several versions of the OS. While Windows does represent 69% of the market share for desktop and laptop computers, it represents a much smaller proportion of embedded devices, cloud machines, and web severs. For this reason this limits the applicability of this work towards security deployments, as HIDS should encompass cross-telemetry sources from many different machines and sources simultaneously. It can be the case that the work carried out serves as one form of indication of maliciousness, among several potential sources of telemetry and scores. Many modern day security solutions operate as an ensemble of indicators, and this work would serve as one indicator specifically for Microsoft OS hosts. Additional work would have to be carried out to develop new models for other OS and other sources of telemetry besides Windows API calls.

Additionally, one of the other limitations of this work was the focus on a single host machine, without the contribution of other devices or hosts on the network. It was also emphasized that this work did not focus on network traffic at all, as many Malware are equipped with anti-emulation to prevent itself from running at all in circumstances when it detects its executed in a virtual machine. Therefore, this work only covers the situation in the aftermath of a successful lateral movement for example, when the host machine is compromised after the network has already been infiltrated. Or, in cases where the host machine is the primary source of the breach in the network. The only networking that was simulated in this work was the simulated DNS server and the simulated port traffic which attempts to mimic real traffic with forged responses. While this does simulate popular file transfer protocols, network shares, web traffic and the network time protocol, it does not take into account actual devices on the network that partake in broadcasting and ARP requests.

Another shortcoming is that the APIs recorded were restricted to the three (3) event types, and some behaviors may not appear under the three types, such as some privilege escalation APIs such as `AdjustTokenPrivileges`, `OpenProcessToken`, `DuplicateTokenEx`. This reduces the feature landscape to only those available via the three event types, reducing training and inference time but also reducing the model's potential discriminatory power. Additionally, the graph topology is generated and exported as a static graph after execution in the sandbox environment. This has the effect of eliminating the time dimension that would be available if the graph were considered a dynamical graph with nodes and edges being generated over a time interval δ. As a result, this work also didn't account for processes that were closed during capturing of the snapshot, which would serve as an important indicator for maliciousness. The APIs would have still captured this behavior however.

It is worth stressing that this work focused on the application and development of XGAN layer, and less so on developing the most comprehensive malware detection tool. While XGAN did show a performance boost over a vanilla model, it was not assessed to be the best tool available for detecting malware, or the most suitable for malware research as no comparisons were made to other techniques. The graph-based benchmark datasets simply demonstrated an outside application besides malware scoring, and the malware detector provided an practical engineering use-case of the technique. This work can be considered one of many in a series of works that aim to demonstrate a theoretical

foundation for the complexity layer and its application as an accessory towards detecting Malware behavior by detecting malicious subgraphs and the strongest predictors of malicious behavior via the API calls used. Fig. 5.1 demonstrates the timeline of a series of published works that act as precursors to this thesis work.

Figure 5.1: Timeline of published works by this author towards dynamic identification of malicious binaries using complexity techniques.

Another consideration is the use of the Malware Dataset retrieved from VirusTotal that made up the malicious executables that were used for behavioural analysis. While VirusTotal is a leading internet security site for sharing malicious binaries and threat telemetry, the only binaries that are available for research purposes and for AV researchers are those binaries that have been flagged as malicious by one or more AV products or sandboxes. This in effect means the viruses are already known to the community, and flagged as malicious by AV vendors with known signatures. This is a necessary limitation of this work because zero-day attacks, by their nature, are not known to the community and are unpatched exploits. It is the belief of the author that the work is still applicable to users with unpatched machines, and for those host users which still ignore best security practices and are susceptible to being exploited by practices such as phishing attacks - which still occur at large enterprises with well-established security postures. It is the case that hosts commonly run older devices and patches for the purposes of legacy support, therefore there is always an avenue where hosts are not fully up-to-date on their security updates.

The next consideration for this work is in the applicability of a host-based solution and the overhead that comes with deploying such a solution. By their nature, HIDS are computationally heavy loads, and graph models require knowledge of the while host environment in order to draw inference on the maliciousness of processes. For example, Windows Defender checks the maliciousness of executables by taking a hash of the telemetry of the file contents and the host environment, then those features are sent to the cloud for inference. As another example, NVIDEA Morpheus AI framework makes use of graph autoencoders with specially designed NVIDEA Triton Inference Server for their deployment (see example here). These graph autoencoders operate at a device and user level, and they operate at a much higher level than at the Windows API level and kernel level which was covered in this work. This does serve to demonstrate that graph techniques are applicable in the field of Cyber Security, but care must be taken to ensure the computational overhead is suitable for the organization and their various resource constraints.

5.4 Future Work and Final Thoughts

As a follow up to this work: many of the capabilities generated through this research are in the works to be deployed to an experimental web server for user analysis of their own windows process executions. The experimental work operates using a Streamlit Server which can be used to generate a dashboard with a UI for user inputs, similar to the one deployed here by the author.

Simply put, the backend server handles user-generated `.XML` files from Process Monitor from

their own machines, which is then analyzed and formatted for prediction by the model checkpoints generated in this work. The application handles various user inputs that follows from the work completed, with the functionality including the following:

1. A user produces an `.XML` formatted Process Monitor log, that is a collection of File System, Registry, Thread Activity or all event activity. This can be generated by running Procmon obtained from the Official SysInternals page and selecting the desired filter type for each event type.

2. The user can use the module level classifier or the Windows API classifier. It was demonstrated in this work that the Windows API classifier was a better discriminator than the module level.

3. The application displays a visualization of the process execution, and labels some interesting edge and node attributes for the user to visualize. Some graph level statistics are shown as well.

4. Based on the model checkpoint selected, the user can use their execution and run it against the classifier to generate a score of maliciousness for each process. These results are tabulated and the anomaly scores are shown below.

This Streamlit application is a work in progress, and the codebase can be found in the official project repository.

Additionally, one of the major advantages of the Mass Radius and Radius of Gyration Fractal Dimension is the sensitivity, especially when the dimensions of input dataset/embeddings grows. Therefore, a broader investigation would include testing the Complexity Layer on different types of embeddings beyond simple ANNs and graph embeddings. There are domains where the complexity of the embeddings would not be suitable, such as in Computer Vision embeddings; however, even for Computer Vision the final Linear Layers make the decision when being used for a classification task. In previous work, information complexity embedded into a CNN architecture in Brezinski and Ferens, [2] used a vel max pooling layer to compute the box-counting fractal dimension. For other applications the complexity measure would have to be integrated creatively in order to take into account the discriminatory power of the measure. As it currently stands with the implementation, computing the complexity of the embedding layers directly is the only application. The next section briefly describes the several forms in which the Mass Radius and Radius of Gyration Fractal are integrated in code, and how they can be applied more broadly in the field of DL and ML.

5.5 Code Contributions

As an extension to this work, the code base has been made freely available on the Github for this work found here. The main contribution of this work is the Complexity layer itself, which has been made available and fully compatible with Pytorch 2.1.1+cu118 as of the writing of this document. Initialization of the model layer is as simple as importing the `FD` class and passing in the relevant arguments, as shown in Code Snippet 5.1. The layer also inherits from `torch.nn.module`, so the layer has all the capabilities of a Pytorch layer, including interoperability with torch optimizers which provides gradient tracking and backpropogation through the computational graph.

Code Listing 5.1: Fractal Dimension Layer Initialization

```
from models.definitions.FD import MassFD

layer = FD(
        k=5,                # num of scales to use on the log-log plot.
        skip=0,             # num of points to skip on the log-log plot.
        gyration=False,     # flag to enable radius of gyration instead of mass-
                                                    radius
        use_weight=False,   # flag to enable the use of learnable weights
        norm=2.,            # order of the norm for the minkowski distance
        **kwargs)           # any other key args to pass to the layer
layer = layer.to(device)    # option to cast to GPU if available
```

Additionally, the layer is compatible with `torch.nn.ModuleList()` and `torch.nn.Sequential()` classes, meaning the layer can be appended in any configuration needed as part of a larger deep learning framework. Inside each layer is reference to the computation which actually computes the Fractal Dimension. This is shown in code snippet 5.2. Some additional code is written for error handling, numeric stability, and interoperability with PyTorch tensors so that gradients flow through the layer. The only other notable difference is the use of Cholesky Decomposition as it is a more effective way of solving Linear Regression in matrix form. This would be especially the case with larger matrices, but in this case it is also used for numerical stability. The code snippet of 5.2 exists in a more general form for computing the FD of non-tensors. This implementation can also be found in the GAT-Malware repository. Many of the details for model inference and viewing model performances can be found in Section and C.3 in Appendix C.

Code Listing 5.2: Fractal Dimension Layer Computation

```
def calculate_fd(self, x):

    # remove zero values; useful for numeric stability when n_dim is low
    x = x[~torch.all(x == 0., axis=1)]

    # mean along each axis
    centroid = x.mean(axis=0)

    # find distances and sort by proximity to centroid
    dist = torch.pow(torch.sum(torch.abs(x - centroid) ** self.norm, axis=1), 1. /
                                                self.norm)
    dist_idx_sorted = torch.argsort(dist)

    # split distances array into k partitions
    N = np.array_split(dist_idx_sorted, self.k)
    # points for log-log plot
    Rg, Nk = [], []

    # loop through scales
    for i in range(self.k):

        # concatenates sucessive nodes away from centroid (e.g Nk_1, Nk_1 + Nk_2)
        Nj = x[torch.cat(N[:i+1])]
```

```
        # compute the centroid for each k if using ROG
        if self.gyration:
            centroid = torch.mean(Nj, axis=0)

        # compute the std. dev in all d dimensions; can use mean instead of sum
        Rg.append(torch.pow(torch.sum(torch.abs(Nj - centroid) ** self.norm) / len(Nj
                                                ), 1. / self.norm))
        # append the number of nodes considered
        Nk.append(Nj.shape[0])

    # convert lists to tensors to make them (k, 1) matrices
    Rg = torch.Tensor(Rg).unsqueeze(-1)
    Nk = torch.Tensor(Nk).unsqueeze(-1)

    # vanilla implementation, just used the raw std. devs.
    log_Rg = torch.log(Rg)[:-self.skip] if self.skip != 0 else torch.log(Rg)
    log_Nk = torch.log(Nk)[:-self.skip] if self.skip != 0 else torch.log(Nk)

    # closed-form least squares linear regression (X.T X)^-1 X.T y
    D = torch.matmul(torch.matmul(
        torch.cholesky_inverse(torch.matmul(log_Rg.T, log_Rg)), log_Rg.T), log_Nk)\
        .squeeze(-1)

    return D
```

Appendix A

Research Contributions

This chapter will outline my research contributions and deliverables, internship experience, as well as my award and scholarships received during my doctorate studies. I believe this chapter serves to demonstrate my completed progress at the doctorate level, and all my academic contributions as a student of the University of Manitoba.

A.1 Peer-Reviewed Contributions

This section will briefly summarize my research contributions to the field of academia, and provide justification for the research and the impact it had broadly in the field. The series of works is listed by publication date, with included links to the full-text and video presentation can be found at my ResearchGate and Personal Page. Publication are listed in chronological order from least recent to most recent; and includes ten (10) first-authored, peer-reviewed publications in Book Chapters, Journal Papers, and Conference Papers.

Cognitive Hybrid PSO/SA Combinatorial Optimization

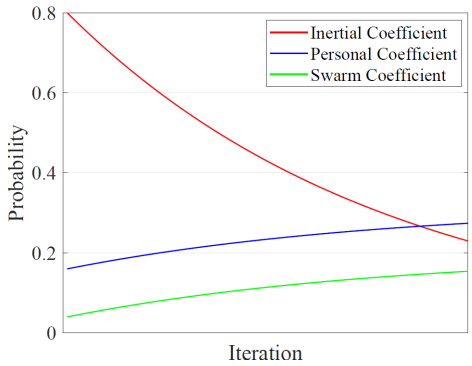


Figure A.1: Illustration retrieved from personal work in [460]

In this work a hybrid model based on two techniques, Particle Swarm Optimization (PSO) and Simulated Annealing (SA) were used to solve a series of combinatorial optimization problems. The two techniques were used to balance the exploration versus exploitation of the solutions space in

order to find a better global objective. This work also looked at different scheduling for the inertial, personal and swarm coefficients (as shown in Figure A.1) which balances the degree in which PSO members communicate with one another.

Population Based Equilibrium in Hybrid SA/PSO for Combinatorial Optimization

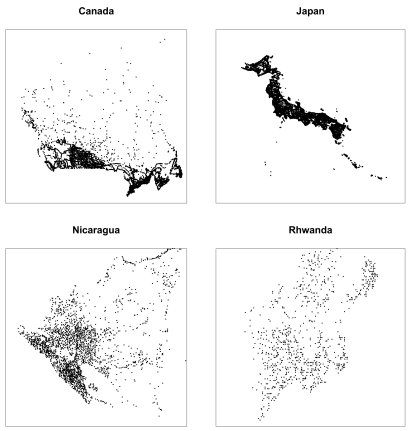


Figure A.2: Illustration retrieved from personal work in [8]

In this work the findings built off the previous work by modifying the objective function to optimize for the travelling salesman problem for a list of countries with progressively larger number of cities, as well as some benchmark problems. The findings demonstrated 34-44% improvement in running time with a 0.5-1.2% penalty to the objective function evaluation as compared to SA alone. The methodology is also directly transferable to other optimization problems, and is not restricted to those which are combinatorial-based – thereby adding to the novelty of this work.

An Adaptive Tribal Topology for Particle Swarm Optimization

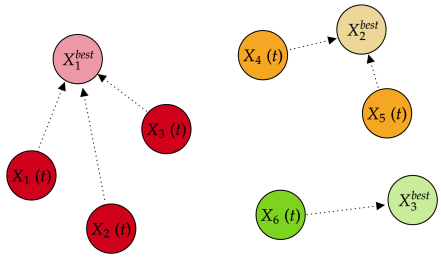


Figure A.3: Illustration retrieved from personal work in [5]

Following on the previous two works, this manuscript introduced a novel tribal topology that dynamically managed the size of the population by spawning and killing off members depending on their productivity in finding better or worse objective solutions. The work also introduced a new tribal coefficient to supersede the swarm coefficient so that tribes acted independently from one another.

Depending on the nature of the solution space, some tribes thrive while others die off, while others fluctuate based on their immediate and previous progression.

Complexity-Based Convolutional Neural Network for Malware Classification

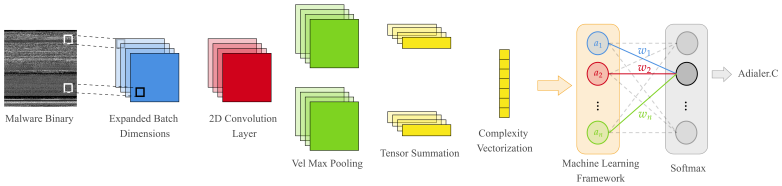


Figure A.4: Illustration retrieved from personal work in [2]

In this work a modification was introduced to a standard convolutional neural network (CNN) layer that computes the Kolmogorov complexity of a malicious binary formatted as an image. The complexity was computed for different sections of the binary, which were then passed through a dense layer to produce the final activations and predictions. This work was carried out for 9300 binaries belonging to 25 malware families, with the modified CNN architecture achieving a categorical accuracy of 96.54%.

Sandy Toolbox: A Framework for Dynamic Malware Analysis and Model Development

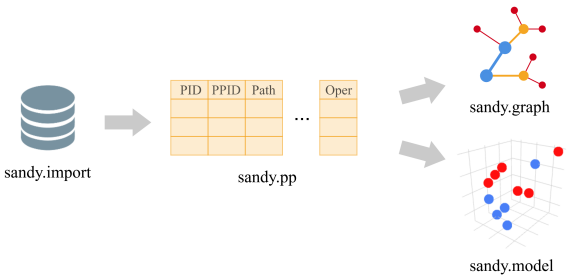


Figure A.5: Illustration retrieved from personal work in [7]

This work sets the stage for the execution of mawlare samples through the creation of a pipeline which sets up a virtualized environment to collect the dynamic feature of process spawn behaviour. The pipeline coined *Sandy* can extract raw static features of the process along with the APIs and DLLs used by the process. The toolbox of scripts also tracks the parent-daughter spawn behaviour of the malware sample to use for further classification approaches. The work also demonstrates a use-case for the collection of dynamic behaviour for the malware variant Emotet.

Transformers - Malware in Disguise

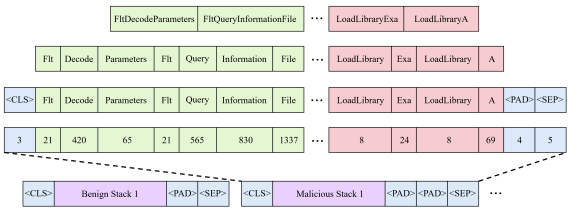


Figure A.6: Illustration retrieved from personal work in [3]

In this work the stack traces of benign and malicious binaries are collected and classified using a transformer architecture. The transformer encodes the sequence of API calls, and learns the context of the APIs used to learn a pattern of behaviour. In this work it was demonstrated that encoding the API calls of registry activity resulted in the best performance of 96.2% F1-score for the 9 malware variants executed and tested.

Complexity-Based Lambda Layer for Time Series Prediction

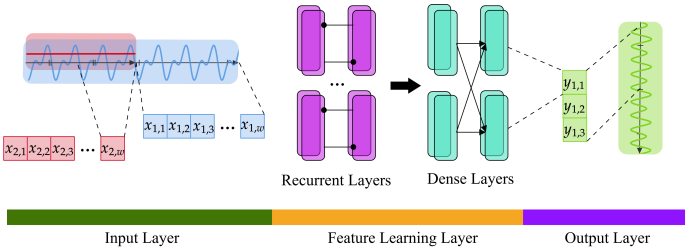


Figure A.7: Illustration retrieved from personal work in [9]

This manuscript focused on developing a novel complexity measure to compute the long-range dependency of a synthesized time series and use that as an additional feature for prediction. The complexity measure can be computed on the fly, and has the benefit of being able to be dynamically computed on the testing set as new data points are being predicted. The addition of the complexity layers, implemented as a lambda layer in a recurrent neural network (RNN), resulted in improved performance as compared to the time series used as a feature alone.

Incorporating Topological Complexity into a Multilayer Perceptron

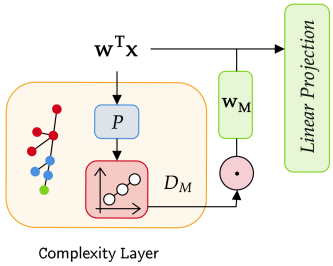


Figure A.8: Illustration retrieved from personal work in [1]

In this work a novel technique to compute the topological fractal dimension was developed, tested and integrated into a multi-layer perceptron. Two flavors of fractal dimension were tested: the radius of gyration and mass-radius. Results showed improved accuracy at the cost of increased loss when tested on synthetic data of varying dimensionalities. This work also set the stage of a trainable layer that would incorporate the fractal dimension as a parameterized metric that can be tuned by the model during backpropogation.

Metamorphic Malware and Obfuscation - A Survey of Techniques, Variants and Generation Kits

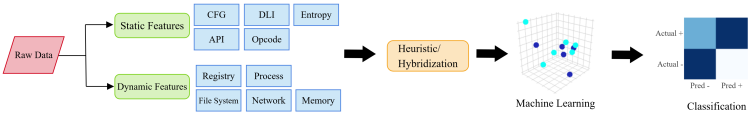


Figure A.9: Illustration retrieved from personal work in [11]

This review paper covers the many forms of obfuscation and encryption used by metamorphic malware to avoid detection by signature-based detection methods. Some examples of each form of obfuscation is given, with some opcode snippets demonstrating how the obfuscation is carried out. Additionally, a chapter covers some common metamorphic generation kits and how prevalent they are used in the literature.

Graph-Oriented Modelling of Process Event Activity for the Detection of Malware

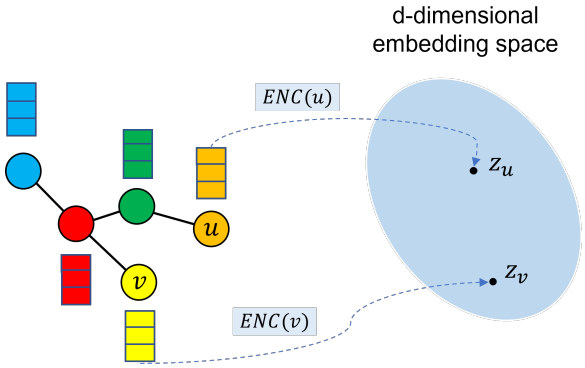


Figure A.10: Illustration retrieved from personal work in [11]

This paper presents an approach to malware detection using Graph Neural Networks (GNN) to capture the complex relationships and dependencies between different components of an operating system (OS). Specifically, this paper investigates the use of GNNs for malware detection based on the API call sequences of different event types, including File System, Registry, and File and Thread activity. A representative dataset of host process activity of malware is collected in a custom sandbox environment which comprises of over 239 malware executions with randomly executed benignware samples. The paper describes the GNN model trained on the dynamic process behavior generated from process execution graphs, with independent models developed based on each category of API events.

A.2 Internship Experience

The following list details the several internship experiences that I took part in over several semesters as a doctoral student at the University of Manitoba. The internships that were salaried are noted as well.

Canadian Tire Corporation - Lead Research Intern

My work with Canadian Tire Corporation (CTC) included a 3-year internship (2019-2022) that allowed me to research the development and application of Machine Learning and Deep Learning techniques for the purposes of classifying malicious portable executable (PE) files. This work also included consultation with Canadian Tire security technicians, as well as quarterly meetings with CTC security executives on ongoing progress and future directions. The bulk of the completed work carried out was done in close collaboration with, and with financial support from, CTC through the Mitacs Accelerate Award from October 2019 to October 2022.

Amazon Web Services - Applied Scientist II Intern

My work with Amazon Web Services (AWS) included a paid summer internship in 2021 in New York, USA that involved working alongside a security and machine learning team of 20+ that focused on developing novel approaches to protecting AWS customers' cloud infrastructure and sensitive information. My contribution included developing the code base for applying weak-supervision for generating weakly labeled training examples on ELF binaries in cloud honeypots. The contributed work was later published in [461].

Microsoft - Data Scientist Intern

My work with Microsoft's Defender for Endpoint team included a paid summer internship in 2022 in Redmond, USA where I worked on developing a detector to alert customers to an early stage ransomware attack. This work including big-data modelling of customer's events from several Microsoft 365 products, and compiling those insights into events and actionable alerts that are presented to endpoint customers.

Mitacs Globalink - Visiting Researcher

This academic collaboration was carried out in collaboration with the University of Manitoba and the National Institute for Informatics in 2022 in Tokyo, Japan as a part of a larger Mitacs Globalink Research Award that was awarded in 2021. The work included analyzing internet backbone data to identify malicious types of scans targeting Japanese research institutes using graph autoencoders using deep learning and explainable AI techniques.

A.3 Awards and Scholarships

The following list of awards and scholarships were awarded during the term of my doctoral studies at the University of Manitoba. Awards are listed in chronological order, from most recent to least recent.

Award	**Date Awarded**	**Awarded Amount (CAD)**
Research Completion Scholarship	2023	5,000
Emily and Lynette Hain Graduate Engineering Scholarship	2022	11,150
University of Manitoba Graduate Fellowship	2021-2022	36,000
Edward R. Toporeck Graduate Fellowship in Engineering	2021-2022	9,675
Philip and Marjorie Eckman Scholarship in Engineering	2021-2022	2,775
Mitacs Globalink - JSPS	2021	7,888
A. Keith Dixon Graduate Scholarship in Engineering	2020	1,500
Mitacs Accelerate – Ph. D	2019-2022	45,000
NSERC – CGS M	2019	22,500
	Total	141,488

Appendix B

Online Complexity Proof

Starting with computing the variance of x to produce Eq. 3.9:

$$
\begin{aligned}
\sigma_x^2 &= \frac{1}{N_k}\sum_{j=1}^{N_k}(x_j-\bar{x})^2 \\
&= \frac{1}{N_k}\sum_{j=1}^{N_k}\left(x_j^2-2x_j\bar{x}+\bar{x}^2\right) \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}\left(x_j^2\right)-\sum_{j=1}^{N_k}\left(2x_j\bar{x}\right)+\sum_{j=1}^{N_k}\left(\bar{x}^2\right)\right] \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}\left(x_j^2\right)-2\bar{x}\sum_{j=1}^{N_k}\left(x_j\right)+N_k\bar{x}^2\right] \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}x_j^2-2\bar{x}N_k\bar{x}+N_k\bar{x}^2\right] \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}x_j^2-2N_k\bar{x}^2+N_k\bar{x}^2\right] \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}x_j^2-N_k\bar{x}^2\right] \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}x_j^2-N_k\left(\frac{1}{N_k}\sum_{j=1}^{N_k}x_j\right)^2\right] && \text{substituting the full expression for } \bar{x} \\
&= \frac{1}{N_k}\left[\sum_{j=1}^{N_k}x_j^2-\frac{1}{N_k}\left(\sum_{j=1}^{N_k}x_j\right)^2\right] && \text{producing Eq. 3.9}
\end{aligned}
$$

A similar proof applies for calculating σ_y^2.

Appendix C

Model Architecture and Hyperparameter Tuning

C.1 Complexity Layer and Hyperparameters

Weight matrices were all uniformly distributed using He initialization [448, 462]. He initialization uses a uniform distribution ($W^l_{i,j} \approx N[\mu = 0, \sigma^2 = 0.01]$) to randomly initialize the weights in a range that is determined by the number of input and output units in the layer. This helps to ensure that the signal is propagated correctly through the network, and that the gradients are distributed evenly, which can improve the performance of the network [462]. It has been shown to improve weight initialization by preventing saturation of the activations which can lead to zero gradients if the inputs are very large, positive or are negative values. The equation for He initialization is shown in Eq. C.1:

$$W^l := W^l \cdot \sqrt{\frac{2}{m^{(l-1)}}} \tag{C.1}$$

Where m is the number of input units to the next layer l (i.e. the feature dimension in the first layer, or the size of the hidden layers in subsequent layers). The scaling factor $2^{0.5}$ takes into account the fact that with *ReLU* activation the mean of the activations are not centred at 0 - which is the case for *sigmoid* or *tanh* functions. A bias vector $\mathbf{b}$ was used and tested but it was shown to not improve performance for all models. We refer readers to Section C.2 which indicates whether bias was set to `True` in the final `Linear` layer.

Additionally, *Dropout* is used to regulate the network to prevent the model from overfitting [463]. Unlike in [4] where the authors implemented *dropout* of the original feature vector $\bar{h}_j$, in this work the sparsity of the vectorized API calls makes it so that *dropout* would have a minimal effect. *Dropout* is however applied after the linear projection and after determining the attentions coefficients, as this is typical in a deep neural network to regulate overfitting and prevent the network from relying on a particular set of weights. So for each forward propogation and at each layer, a binary mask is set for each input unit j, which is drawn with probability p where $r^l_j \sim Bernoulli(p)$ at each layer l. Each input layer is then masked with r^l_j where $\tilde{\mathbf{x}}^l := \mathbf{x}^l \odot r^l_j$ to produce an output with fraction p of layers set to 0. Dropout probability for this work was tested at $p \in \{0.2, 0.5\}$. Since the model

learns with a proportion of layers p set to 0, during inference the model output needs to be scaled by a proportional amount equal to $\tilde{\mathbf{x}}^l := \mathbf{x}^l \odot (1-p)$ to account for the scaled back activations which the model is familiar with during training.

Additionally, the *LeakyReLU* activation function which was set with a negative slope $\alpha = 0.2$; where the function evaluates to αx when the input value $x < 0$, otherwise it evaluates to x in the piece-wise function. This function has the advantage of producing large gradients when needed similar to *ReLU*; with the added advantage that gradients do not die out as the gradient can still recover when $x < 0$.

The Adam optimizer was used as the training optimizer which combines stochastic gradient descent with momentum with RMSProp [464]. Readers can refer to the original paper for the procedure, or the Pytorch documentation for an overview of the algorithm. The optimizer's β parameters were set to $(0.9, 0.999)$, with a learning rate $\eta \in \{5\times10^{-2}, 1\times10^{-2}, 5\times10^{-3}\}$ depending on the model architecture (see Section C.2). L2 regularization was also implemented through weight decay λ at 10% of the learning rate [465]. So for a η of 5×10^{-3} the decay rate would be 5×10^{-4}. In the Adam optimizer, the decay rate is multiplied by the weights w_t at iteration t which penalizes the magnitude of the weights at each epoch.

C.2 Model Architectures

This section outlines the model architectures and configurations used for each model runtime in Section 3.5.3. The notation `GCNConv(128, 32)` refers to an input dimension size of 128 and an output dimension size of 32 for the `GATConv` layer. A `GCNConv` layer is the simplest form of graph convolution which includes a message passing and an aggregation step as noted in Section 2.6. `input_dims` refer to the size of the input dimensions, which is the node feature vectors size which varies based on the benchmark dataset tabulated in Table 3.3. `num_classes` refer to the number of classes which are predicted on which also varies as per Table 3.3. Various model architecture were tested in tandem during the course of this work. Refer to Section C.3 for a full description of these activities.

Code Listing C.1: Sequential GNN architecture for evaluating graph benchmark performance for experimental results found in Section 4.1.

```
XGAN(
  (layers): ModuleList(
    (0): FD(k=5, skip=1, gyration=True)
    (1): GCNConv(input_dims, 32, heads=1)
    (2): ReLU(inplace=True)
    (3): GCNConv(32, 16, heads=1)
    (4): ReLU(inplace=True)
    (5): Linear(in_features=16, out_features=num_classes, bias=True)
  )
  (cfg): {
  'learning_rate': 0.005
  }
)
```

Code Listing C.2: Sequential GNN architecture for evaluating process execution performance for experimental results found in Section 4.2.

```
XGAN(
  (layers): ModuleList(
    (0): FD(k, skip=0, gyration)
    (1): GATConv(input_dims, 256)
    (2): ReLU
    (3): Dropout(p=0.2)
    (4): FD(k, skip=0, gyration)
    (5): GATConv(256, 128)
    (6): ReLU
    (7): Dropout(p=0.2)
    (8): Linear(in_features=128, out_features=2)
  )
  (cfg): {
  'learning_rate': (0.005 - 0.01)
  }
)
```

Code Listing C.3: Sequential GNN architecture for evaluating process execution performance for experimental results found in Section 4.2.

```
XGAN(
  (layers): ModuleList(
    (0): FD(k, skip=0, gyration)
    (1): GATConv(input_dims, 512)
    (2): ReLU
    (3): Dropout(p=0.2)
    (4): FD(k, skip=0, gyration)
    (5): GATConv(256, 128)
    (6): ReLU(
    (7): Dropout(p=0.2)
    (8): Linear(in_features=128, out_features=2)
  )
  (cfg): {
  'learning_rate': (0.005 - 0.01)
  }
)
```

Code Listing C.4: Sequential GNN architecture for evaluating dataset performance for experimental results found in Section 4.2.3.

```
XGAN(
  (layers): ModuleList(
    (0): FD(k=3, skip=0, gyration=False)
    (1): GATConv(input_dims, 256)
    (2): ReLU
    (3): Dropout(p=0.2)
    (4): GATConv(256, 128)
    (5): ReLU
    (6): Dropout(p=0.2)
    (7): Linear(in_features=128, out_features=2)
  )
```

```
  (cfg): {
  'learning_rate': (0.005 - 0.01)
  }
)
```

Code Listing C.5: Heterogeneous GNN architecture for evaluating process execution performance for experimental results found in Section 4.2.6.

```
XGAN(
  (layers): ModuleList(
    (0): FD(k=4, skip=0, gyration=False)
    (1): GATConv(input_dims, 128)
    (2): ReLU
    (3): Dropout(p=0.2)
    (4): GATConv(128, 128)
    (5): ReLU
    (6): Dropout(p=0.2)
    (7): Linear(in_features=128, out_features=2)
  )
  (cfg): {
  'learning_rate': (0.001 - 0.005)
  }
)
```

Code Listing C.6: Ensemble GNN architecture for evaluating process execution performance for experimental results found in Section 4.2.6.

```
XGAN(
  (branch 1): ModuleList(
    (0): FD(k=4, skip=0, gyration=False)
    (1): GATConv(input_dims, 128)
    (2): ReLU
    (3): Dropout(p=0.1)
    (4): GATConv(128, 128)
    (5): ReLU
    (6): Dropout(p=0.1)
    (7): Linear(in_features=128, out_features=64)
  )
    (branch 2): ModuleList(
    (0): FD(k=4, skip=0, gyration=False)
    (1): GATConv(input_dims, 128)
    (2): ReLU
    (3): Dropout(p=0.1)
    (4): GATConv(128, 128)
    (5): ReLU
    (6): Dropout(p=0.1)
    (7): Linear(in_features=128, out_features=64)
  )
    (branch 3): ModuleList(
    (0): FD(k=4, skip=0, gyration=False)
    (1): GATConv(input_dims, 128)
    (2): ReLU
    (3): Dropout(p=0.1)
    (4): GATConv(128, 128)
```

```
    (5): ReLU
    (6): Dropout(p=0.1)
    (7): Linear(in_features=128, out_features=64)
  )
    (final layers): ModuleList(
    (0): LayerNorm(embedding_dim=512)
    (1): ReLU
    (2): Linear(in_features=64, out_features=2)
  )

  (cfg): {
  'learning_rate': (0.001 - 0.005)
  }
)
```

C.3 Model Binaries and Inference

Model binaries for the best performing models are logged and stored for future inference. For this work MLflow was used for model registering and keeping track of model states, checkpoints, and binaries. Once the project repository is cloned in any given directory, the MLFlow tracking server can be started using the command `mlflow ui --port $PORT$` which will start the server with the UI at `http://localhost:$PORT$`. Navigating to this URL in your browser will show a dashboard screen similar to that shown in Fig C.1. This screen presents all the models trained as a part of this work, with the columns providing a listing of some of the model performance metrics (such as *val_acc* and *val_loss*) and model specific parameters such as architecture, learning rate, whether or not the Fractal Dimension was used in the model (`use_fd:bool`), etc.

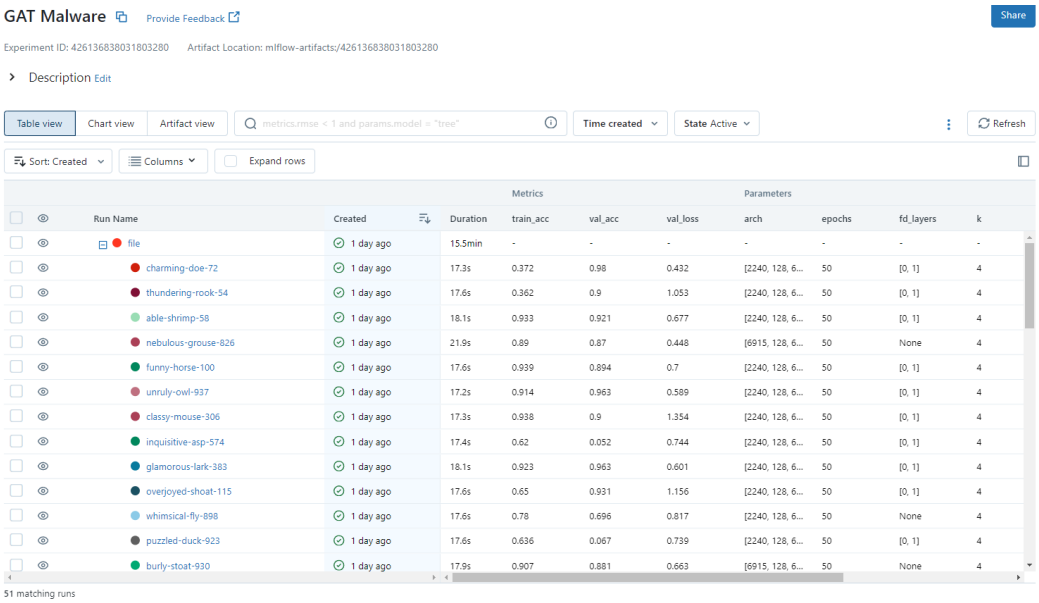


Figure C.1: MLFlow dashboard for comparing runs, model artifacts, and performance metrics.

For any given `ROOT` directory in which the code repository of this work is cloned into, the model binaries are stored in `$ROOT$/mlflow-artifacts:/$ExperimentID$/$RunID$/artifacts/model/`

`data/model.pth`. *Experiment ID* refers to the parent identifier, which for this work is broken down by dataset (e.g. file, reg, thread, Cora, Facebook, SiteSeer) and *RunID* is the specific identifier randomly assigned for a given set of hyperparameters tested. `the model.pth` is the raw model which can be used for inference, or alternatively, one can use any of the built in tools in MLFlow which can serve the model locally as a RestAPI, or to various environments such as Kubernetes, or as a self-contained Docker image (see the reference documentation).

For local inference, the model can be loaded using `model = mlflow.pytorch.load_model("runs:/$Run ID$/path/to/model.pth)` and then used to predict like any other model. Since the graph model expects certain types of input from the graph framework - such as graph edges and feature vectors - the instructions on the official Github Repository README.md should be followed.

Additionally, the `/model/` directory contains the various artifacts that can be used to create the virtual environment to run the model on the local machine. This includes `conda.yaml` for creating and loading the environment into an Anaconda Environment, or `requirements.txt` for creating an environment using `venv` or `pip`. Instructions are also found at the project Github repository.

C.4 Viewing Performance Trials

Given the several thousands of performance trials carried out in the work, it was decided that models and performance metrics be stored and logged as well in the MLFlow workflow. This allows for quick comparison between model trials using the dashboard shown in Fig. C.1. This allows for the models performances to be visualized using the several built-in tools, such as the parallel plot shown in Fig. C.2. In this Figure the models whereby the Radius of Gyration was used (`gyration=True`) are compared against those that the Mass Radius was used (`gyration=False`) and where the Vanilla model was used (set to `None` by default). Models can then selected and deselected on the side panel to compare particular models based on hyperparameter condition, model size, performance, etc.

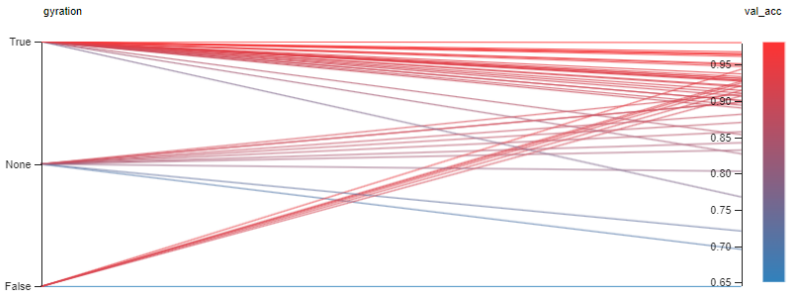


Figure C.2: Parallel plot for visualizing the use of Radius of Gyration Fractal Dimension and the effect on validation accuracy (*val_acc.*)

Alternatively, during the training process learning rate is always a difficult parameter to tune as it very model and dataset specific, and there are plenty of ways to traverse the solution space to find an optimal learning rate. For this a log-uniform sampling strategy was used select the learning rate for each trial. Fig. C.3 presents a contour plot showing validation accuracy of the model as a function of learning rate (shown as logarithmic). In this way future model performance can be fine-tuned based on expected performance of the learning rate selected. Any model hyperparameter (model size, architecture, layer type, etc.) can be selected and compared to any performance metric.

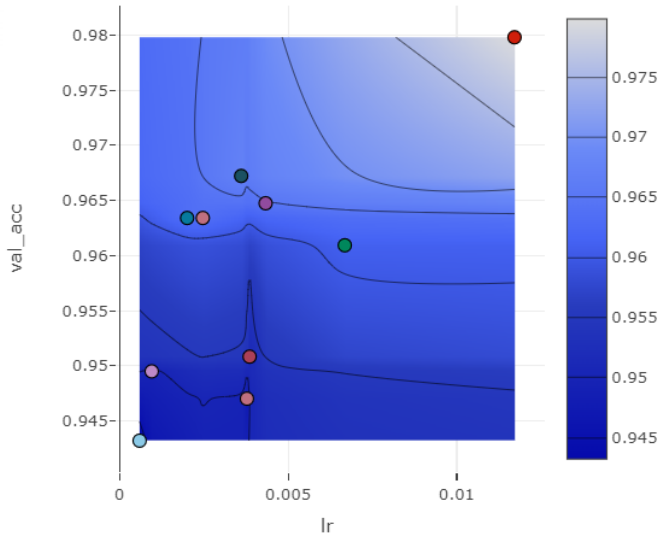


Figure C.3: Contour plot for visualizing the choice of learning rate and the effect on validation accuracy (*val_acc.*)

C.5 Complexity Layer Runtime Performance

The complexity layer performance was determined both Mass Radius and Radius of Gyration implementations (see Alg. 1 and 2) and are shown in Table C.2 and C.1, respectively. As the main difference between the two implementations is the recalculation of the centroid, the run-times are nearly identical and are not statistically significant (see below). For run-time performance the metric used to evaluated this is GPU execution time which is the execution time of the kernel itself after all tasks on the GPU are completed. For testing an NVIDEA RTX 2060 Super with 8GB of VRAM was used running CUDA version 11.8 on Pytorch 2.1.1+cu118. The process being benchmarked is a single `forward()` method call on the complexity class, which is one passthrough of Alg. 1 or 2 with the implementation shown in Code Snippet 5.2. This computation also includes the scalar product with the learnable weight w_D which takes constant $\Theta(1)$ time.

Runtime performance was ran on varying sizes of input feature size d and input sample size n. In typical applications, hidden layers sizes vary between 16 for smaller networks to 2048 for larger image networks such as ResNet-50 [448]. For this reason feature dimensions $d \in \{81, 256, 625, 1296, 2401, 4096\}$ were tested with fairly large sample sizes $n \in \{10^3, 10^4, 10^5\}$. There are two considerations for the sample sizes used. First, for tabular datasets that are relatively small (>1,000,000) it is not uncommon to load the whole dataset into memory. However, for large datasets in the billions, which are typical in Deep Learning applications, small samples sizes are passed into the network as batches; with batch sizes of 64 - 512 not being uncommon. Therefore, the range used covers both applications of practical usage with modern DL applications, as well as circumstances where the dataset is loaded into memory.

The results shown in Table C.1 demonstrate a linear trend in the GPU execution time from 10^4 and 10^5 for high values of d, where $d \in \{2401, 4096\}$. This is consistent with the run-time noted in Eq. 3.5.2 whereby the run-time is dominated by the $\Theta(n \times d)$ term which is linearly dependent on both the size of n and d. Similar trends are seen in Table C.2 for large feature and sample sizes.

For small n and d the run-time is dominated by other functions, such as sorting, so we don't see the same linear trend.

Table C.1: Mean runtime performance and standard deviation in *microseconds* for the execution of the Radius of Gyration `Forward()` method. All mean and standard deviations are recorded for 30 iterations of 10 loops each.

		Input Sample Size		
		10^3	10^4	10^5
Input Feature Size	81	3.85 ±0.112	3.92 ±0.15	9.42 ±0.12
	256	3.62 ±0.109	4.29 ±0.121	24.2 ±0.99
	625	3.61 ±0.118	7.04 ±0.032	57.2 ±0.79
	1296	3.60 ±0.099	13.0 ±0.028	116.0 ±1.26
	2401	3.88 ±0.089	22.9 ±0.043	203 ±1.03
	4096	4.85 ±0.063	37.4 ±0.314	421 ±0.99

Table C.2: Mean runtime performance and standard deviation in *microseconds* for the execution of the Mass Radius *forward* method. All mean and standard deviations are recorded for 30 iterations of 10 loops each.

		Input Sample Size		
		10^3	10^4	10^5
Input Feature Size	81	3.93 ±0.142	3.73 ±0.177	8.8 ±0.105
	256	3.92 ±0.152	4.41 ±0.174	23.1 ±0.31
	625	3.62 ±0.217	6.81 ±0.108	54.4 ±0.92
	1296	3.72 ±0.175	12.2 ±0.099	112 ±2.77
	2401	3.96 ±0.182	21.1 ±0.081	241 ±1.03
	4096	4.83 ±0.169	35.1 ±0.726	421 ±0.99

Comparison of means was carried out between the values two independent samples which in this case are the values between Table C.1 and Table C.2. For **Input Sample Size** of 10^3 and **Input Feature Size** of 81, the Standard Error is 0.044 (-0.0085, 0.1685; 95% CI) for $t(38), p = 0.075$. The conclusion is that these two means are not significantly different. This is possibly due to the time taken to cache the model when initializing the model during tests, which leads to a larger than expected standard deviation.

C.6 Model Memory Requirements

Along with runtime requirements, there is memory overhead associated with the model binaries that include the model weights and auxiliary storage. The memory requirements for the Complexity layer are small in comparison to the rest of the model, as evident in the table presented in Table C.3. Table C.3 presents a single candidate model that has been used in this work, comprising of two (2) FD layers, two (2) GATConv layers, followed by a final linear layer. The single parameter associated with the FD layer is w_D, with the rest of the parameters being allocated for the weights in the convolution GAT layers. This demonstrates that while the computational cost of the FD

layer is non-zero as noted in Section C.5, the memory requirements are constant $\Theta(1)$ with the size of the input features, the number of layers, and the number of neurons in the previous or next layer. Compare this to a GATConv layer that scales with the input dimensions, *d_in*, and output dimensions, *d_out*, of the layer in question.

Table C.3: Summary table of the memory overhead of an XGAN model architecture. Memory requirements for the optimizer state, and any other auxiliary storage are excluded from this table. **Est. Size** in KBs are based on each learnable parameter, such as weights and biases, assigned as a 32-bit floating point (float32 tensors).

Layer	**(d_in, d_out)**	**# Parameters**	**Est. Size (KBs)**
FD	N/A	1	0.004
GATConv	(2240, 512)	1.1 M	4595
FD	N/A	1	0.004
GATConv	(512, 32)	16.5 K	64
Linear	(32, 2)	66	0.26
Total		**1.2 M**	**4660**

Appendix D

Sandbox Configuration

D.1 Process Monitor Configuration

Table D.1: Process Monitor configuration for filtering out relevant events. With the exception of the final entry, all other entries are disallow filters. Configuration is set to Allow By Default unless otherwise covered by a rule.

Entity	Relation	Value
Process Name	is	`Procmon.exe`
Process Name	is	`Procexp.exe`
Process Name	is	`Autoruns.exe`
Process Name	is	`Procmon64.exe`
Process Name	is	`Procexp64.exe`
Operation	begins with	`IRP_MJ_`
Operation	begins with	`FASTIO_`
Result	begins with	`FAST IO`
Path	ends with	`pagefile.sys`
Path	ends with	`$Mft`
Path	ends with	`$MftMirr`
Path	ends with	`$LogFile`
Path	ends with	`$Volume`
Path	ends with	`$AttrDef`
Path	ends with	`$Root`
Path	ends with	`$Bitmap`
Path	ends with	`$Boot`
Path	ends with	`$BadClus`
Path	ends with	`$Secure`
Path	ends with	`$UpCase`
Path	ends with	`$Extend`
Event Class	is	`Profiling`$^{\alpha}$

$^{\alpha}$ In addition, alternates between Process, Network, File System and Registry depending on which events are to be collected.

D.2 Malicious Binaries

Table D.2: Malicious executables used in sandbox process execution. Entries with similar names but different hashes correspond to metamorphic and/or updated variants.

Name	File Size (bytes)	MD5 Checksum
All.ElectroRAT	903091	7ff8d31ad43f62f1c6876b725a1ebb1f
AndroRat_6Dec2013	28237699	2dd2ced8aa357e7e4a6bd98ff52e4b9a
AntiExe.A	3178	8cee47cd109adfa5c5816685af873909
Artemis	13362720	2599847a7535908f7c0db0a6b16dbf0e
Backdoor.MSIL.Tyupkin	588477	69fbc6a70b315d827c524bea4b899c44
BAT.Drop	1838	935ce64b55d3462931375e344da1ce38
BAT.Pot.A	1221	fe0363bd9b28b1d919086e642cb0e03c
BAT.Skul	1181	b46113a11b853c258a5480eb3648cd83
BlackEnergy2.1	72496	c3adb3e2370c964162babae20c88c142
Brain.A	61148	c56f135fdaff397ad207f61b4f2042fe
Careto_Feb2014	448438	19fac82fa4ef163ab1b06ef282366393
Cascade.1701.W	2206	5c3cba0b07805a977612de03c93713d6
Catapillar.E	960	910c2d354eff59b037b957f2a7262992
Civil_War.282	1475	772e9c91dddef0a3e297d21edd1a653d
Coll.CozyBear	13594106	e049fd6d80d9285d156cdf5785a6e28e
Coll.DarkHydrus	222618	97f1f90f90e3bc5419c99788dde14f60
CryptoLocker_10Sep2013	289354	22078ff56e3fcd674ec4b9322a7dee5b
CryptoLocker_20Nov2013	604410	eb5eb336636e3f6cacf6c8db6bf4ea00
CryptoLocker_22Jan2014	343590	3c877dfd0d60572be7c939c08c39866d
Dino	255989	3cdb9b91b2d8b971a2cc708968097337
DOS.Yesmile	5103	560a3a31bf1b43e344e3a83d33ef991f
Dropper.Taleret	75163	32a5b4a5d8010926990f3d5ff08018d2
Duqu2	430393	e64d31ef596e86997ca0fffcfb3d1ce8
Dyre	2267060	6d1f649d90313b7e3624c0e86563b5dd
EquationGroup	25976318	88106b8b1ef1a00644a90ca67ad57e1f
EquationGroup.DoubleFantasy	94197	6ce2f698864ac5acf73c4ddbee430299
EquationGroup.EquationDrug	338787	bfe8ccc7c521a7f307e5339fc9d26a23
EquationGroup.EquationLaser	77334	49fb03c99aef6941045d16d82b315db0
EquationGroup.Fanny	100962	fec2f8b0db204081bce309eb049f5992
EquationGroup.GrayFish	486963	b3e74a076efaa5c85af764c2f88a4840
EquationGroup.GROK	82975	26926bb2b72c0d16d7d49bc3d1efdef1
EquationGroup.TripleFantasy	9260	da1837cb0827fe4fbdddbfacd604fcd3
FancyBear.GermanParliament	10311	86c8ebffbee876f992cb844147974d3f
Form.A	1324509	2d2da6a6c24a6154e825aaa0b44db836
Friday_the_13th.408	763	2485d09c7b996855eded9ee500625773
Friday_the_13th.416.A	773	9c03c2bc8ee01dc0d7fdfc4bca438c45
Friday_the_13th.416.B	774	3b413c3fdad43837232c3cffa4ff233d
Friday_the_13th.540.A	862	6a3cc7b35c4ccb10b76ee87f6ccfc708

Green_Caterpillar.1575.A	63475	fadaaa30c6429f5640b0805b222b5315
IllusionBot_May2007	231871	19eb6befb4f1f965ad0e1f06747b1742
INTC.A	319004	da926b3920872bd982dc811ef3f49f18
JS.JScript.A	1510	399e6dc863d08d6c038c61675b452fcb
JS.Lame	1563	888928a4788f62fd6895310e6a3cf45e
Jumper.B	1013644	69e7d8e5bb9bd34fc55d3dbe8a667f73
Junkie	13104	b2736c97ab0b3bc23cbfa9e7d073e16d
Kampana.A	109191	eaa87f2c9fb21db6e6b73e4e1dd2d65c
Kelihos	4156806	49ed03d641ec291d81e5967e90f7ba8c
Keylogger.Ardamax	797130	5de75a478ffb3aa01a88f4e539f3edc0
KRBanker	5514454	b3f20aa476f6191a7b939fc3703462a2
Linux.Chapros.A	218248	bfacc2ba5a37100d03fb301a77fc1d4a
Linux.Encoder.1	1101443	b643673e1f2b0496785873657e56b03f
Linux.Mirai.B	182620	23282f4301170e8ee3b157dcfaae6317
Linux.Snoopy.A	5392	fa1727d1de4943dfc8ff96113b8830ed
Linux.Snoopy.B	3897	112a4f6d91a66f8d9fff1aa4df0dbc7e
Linux.Snoopy.C	4593	328da526d939473d1319004654721368
Linux.Wirenet	35349	840489a0707aafb9b6cde992652de1d9
Michelangelo	11587	c3fe84eb644cd1a19f71217118178160
Net-Worm.Win32.Kido	61073	15054c1e49e004b8011dbef3bfa97d08
Neurevt.1.7.0.1	7121691	989d63919aa1a35cc5579cec78b06b9e
Nitlove	98382	d818e58a8309933204e5347c168022b2
Nivdort	502678	2cf9704b9ad48c05501f372a26d14636
njRAT-v0.6.4	1613723	3ccce9d87ce9ea751abea094d1639d0a
NYB.B	31110	3e3f37e2b0db8614544af45316f046b8
Parity_Boot.B	3066	72d4c82283497a8a13c1862ee86203e6
Phoenix.2000	932	96c7937b12d729106ed6ca96870146b8
PlugX	725523	eeb04e18280b5027f1c299f3b1780961
PotaoExpress	24532141	2677e57cd46328182af9ad3ea6dbe8d3
Poweliks	114874	1a1c9e567fb8a496e59ec64a2053282d
Proteus	2568548	66cac61deace1dcceab136a6a0b367e4
Quax.A	1172671	2c6e3ac3c48792d3f52bb8b75a8c5af8
Raccoon.Stealer.v2.sha	603997	0831d0df9d7696f6aed73600539cdb3f
Ransomware.Cerber	220210	5c571c69dd75c30f95fe280ca6c624e9
Ransomware.Cryptowall	103016	8710ea46c2db18965a3f13c5fb7c5be8
Ransomware.Hive	6943657	33dc6cf9108fa7a395d632c29021791c
Ransomware.Jigsaw	245631	3ad6374a3558149d09d74e6af72344e3
Ransomware.Locky	128770	b265305541dce2a140da7802442fbac4
Ransomware.Mamba	1089078	f94d1f4e2ce6c7cc81961361aab8a144
Ransomware.Matsnu	64025	0a3487070911228115f3a13e9da2cb89
Ransomware.Petrwrap	1156716	6884a35803f2e795fa4b121f636332b4
Ransomware.Petya	551523	e8fb95ebb7e0db4c68a32947a74b5ff9
Ransomware.Radamant	60911	fce365d60e13df34a6843894ac9be499

Ransomware.RedBoot	1222010	51250dabf7df7832640e4a680676cb46
Ransomware.Rex	2843585	50188823168525455c273c07d8457b87
Ransomware.Satana	58656	82f621944ee2639817400befabedffcf
Ransomware.TeslaCrypt	491332	f755a44bbb97e9ba70bf38f1bdc67722
Ransomware.Thanos	148809	00184463f3b071369d60353c692be6f0
Ransomware.Unnamed_0	855552	abc651b27b067fb13cb11e00d33e5226
Ransomware.Vipasana	653467	8d2c4c192772985776bacfd77f7bc4d9
Ransomware.WannaCry	3481589	efe76bf09daba2c594d2bc173d9b5cf0
Ransomware.WannaCry_Plus	2402668	5641d280a62b66943bf2d05a72a972c7
Ransomware.XData	176725	1f73d8a54aa818cfe2596ab397b5a613
Rombertik	633795	e39bfb63f8febee08eb8eda80bda7151
Rustock	817321	ca397168c14dd681ea47a5bd57ac4af4
Sampo.A	1978	31ce1fd251cb983dcb166fe315d2fb9d
Shamoon	931363	fd7445210bc60baeeab77f69e1ba51b8
SheHas.A	2348	ee0e62a5ac34ca6c0abd210586846674
SillyC.160.B	1567	9df1dad46fbb988843ad1540f9213c2b
Skywiper-A.Flame	670751	af34546979079042a99b99b5613ad26b
Slow_Format.705	1348	797da07293aa4a2cba10e71245626a0f
Somoto	133351	3387f0112fa04d36c45a82ee0a5322db
SpyEye	1068641	2e0bb844572de2e88cbd23d76101bd16
Surtr	348633	38aec0ac2b0d8fdeaa22bda66b94926d
SymbOS.Lasco	17445	f86f2de279fc17b6806373c9decb78d0
Telefonica.3784	11244	8e64801a1ffd181aac1ee77ad1af78c5
Trivial.881	1767	7e8987b2e159de99bff4bd1155e803a5
Trivial.Html.883	1088	2dbd43b574211d12a197d1f4249ead79
Trivial.LSD	3629	9dbfa879457bf368718b5e88d05dfbe1
Trojan.Alienspy	305477	94e05c5774a48a39bf345ffd54dc3e65
Trojan.Asprox	229680	9077ec98bd1a022616452acdc2d59799
Trojan.Bladabindi	39492	434254111fbe2b2a287aed9211375384
Trojan.Destover-SonySigned	43319	194d5bc3f4f0fbca0cc04c0fdba9a82d
Trojan.Dropper.Gen	1625280	b448ec18ba0e19ca48f4762f1caa0c46
Trojan.Kovter	339822	35a6e61b6cd45e8d1dc6ce9cb5d35cd8
Trojan.Loadmoney	116788	b44418afb1bd4d88d9b515392dfb4e81
Trojan.NSIS.Win32	1751767	934677f6e35807bda7b54377d69e99c9
Trojan.Ransom.Hells	82184	7c796cea93854bff9ca198a45c71b811
Trojan.Ransom.Petya	186884	089dcd9480fa127da76645e445c3e909
Trojan.Regin	104108	b4c4793b8a1ad09c256377c38d3eaafc
Trojan.Shylock.Skype	181329	1dbee013e7788a749f9e55a2b1bd20d5
Trojan.Sinowal	1218141	9538a123b2e1489b39c5f86be4c11010
Trojan.Stabuniq	71181	bd0f5e8fad6ec11bebae4a989f5e5d73
Trojan.Tapaoux	270228	c08cb5a53420f3812e9d6a57145711ba
Trojan.Win32.Bechiro.BCD	158962	2728f56ded381f275f11eec9daf8d68b
TrojanWin32.Duqu.Stuxnet	13303	03bb47f461c51203d6799919dbb37012

Variant.Kazy	54080	a751492c7a969c080d11e0e23239e5ca
VBS.Carnival	2929	82dc117dcfe7e10469ade8cdfa995874
VBS.Hopper	2091	53bee6a0cb9aae28898d839cb2797249
VBS.LoveLetter	100358	85cff1dc0007ea90f164c78b9baae8a4
VBS.NewLove.A	77995	12b8101a28129eb79819b6d3e9b87413
VBS.NoMercy.B	2675	fe8aa88331fc7d50e1b6cdfed2e1a083
VBS.NoWarning.A	1537	4943bc4b8f1b610a7da61f98b098660f
VBS.Redinal	1119	cc3a9aeccdabedd57aafcd43a581130b
VBS.RunScript	1233	64aa1cc0143488e20c0b12c23efaa0bd
VBS.Vquest.ow	1118	ffee646accc0e5c96f9fa59b711956e8
VolatileCedar.Explosion	5393078	1b6ddfe2a7ff1fe134f7582703bab2b7
W32.Beagle	22182	1d8547b6a440017631f5eed263b44c19
W32.CodeRed.Worm.C	474309	48957f52b0ae0505fd529727c724a65b
W32.Duni.A	242914	2a02292fc0be132f2931ff9bfada2131
W32.Elkern.B	127844	7527a0cdb8b2929e000ef5b86ad91e60
W32.HLLP.Hantaner.A	1078000	71eeca330cda987793a013358223b9b8
W32.Hybris.Worm.B	153808	34b5039109db86c92c1fdb982e6a22b6
W32.Klez.E	380329	3ecb9deec89e646d56a6ba932977721c
W32.Klez.H	294815	ccde73ed3c6cbc2124db399359ae6ff9
W32.MyDoom.A	300207	ee70b23f67565ce4822f0f5f8d24525e
W32.Mytob_EJ	34524	6a330b16b3d31b92bd8fe9891bc07f62
W32.NetSky	542	8940be7fb841a1827ed9733a8f5b0260
W32.Nimda.A	177274	b62632c8e6aafa05a07cad52b2e25aa5
W32.Nimda.E	22416	e67dba390d932bb29dfb408859d9d72e
W32.Slammer	39591	23ee946c2226e153f1fc91f222dd2aa2
W32.Swen	52159	be087c652e2d89ac9c54809256724c77
W97M.Class.AU	14881	0709ff9a47954e8add7af911d111854c
W97M.Melissa.A	23545	04b5305038d4101a58b9f2b6d5649c88
W97M.Pri.A	10438	d951d21f6deb1608f1769ccf8b466248
W97M.Pri.AB	14899	7485d9a8637afd89b0763ec24065736a
Waski.Upatre	72465	23091f66349407849cd1d8198881be54
Win32.AgentTesla	334230	7ce44be24584e94d69bbf4cbdc8c1115
Win32.Alina.3.4.B	60828	6ec4bb6df9ddd7a59734b79b96271327
Win32.APT28.SekoiaRootkit	13482	4a51bc4d7dc6eeb91894f6d88e08b3dd
Win32.APT32.WindShield	123343	5a6a546c7dcca9da2fd6688c2ad243c7
Win32.Avatar	121216	b26a58fe73560793831182e0beb94103
Win32.BigBang	1584645	574a0e7644ab1f6c16b98f56d34c09f9
Win32.Boaxxe.BB	92417	9c6a1317b6ddf6c11ec34f3e3240b3c7
Win32.Cainxpii	241	3bff9442a43fa6b11addbaf97776133d
Win32.Caphaw.Shylock	148207	aef8aafb5952a772b36b398bc59d8f0e
Win32.Carberp	147488	dd34fed8a105ad224a98f7f0058afb49
Win32.Cridex	83889	d1a667d57ef9bcd460a3871d639240ce
Win32.Cutwail	173812	db2cc70364a13c3e10789a53043371f3

Win32.DarkTequila	874679	4e81117c03c8cf89b86ac4192ef71aaf
Win32.Emotet	63758	099ec2767271a59ae4fd2cfa9844c9bf
Win32.EternalRocks	21406189	4f18cdbcc1d0e71c3a984a6db0beadde
Win32.FamousSparrow	208274	15f53dd7a7f04394c6c69fea6761b486
Win32.Fareit	1110249	c79b33f2c6d15af48a485350904f02dd
Win32.FASTCash	571600	7441af195c3eeaefe1cc6f2d9ea237b6
Win32.GravityRAT	380140	3ac1ede478f83cd857b5e4fcd06ef1cc
Win32.GreenBug	1573182	be17b07115f832878c47f86e142bb630
Win32.Hupigon	361048	58c573eb875041158453a9c7f3e38588
Win32.Infostealer.Dexter	85758	bd7bad534d1e5a2ad6c11829b96a23e4
Win32.Invicea_Tunnel	1573626	596cd9e046037f548d5cc8ac8da88001
Win32.Ixeshe	15940	a978ee2a371cf7d26a3a94464cc9a377
Win32.Jerusalem	230	a46e53b43a92d463643b375420330fe5
Win32.KerrDown	200109	96271199e51969ab669faa8ebb6de6d9
Win32.KeyPass	1324738	f831ffa7faa4da66482aa252536e1b0f
Win32.Lephic	71084	91daa09da806c089dc5f6f605c59cbca
Win32.LuckyCat	249729	ec9d64b66c3ecaa59b4e62c737a7b3c8
Win32.MyLobot	126300	0247d726d3e34b9494b78cec12bd0374
Win32.Narilam	497791	3e26ce63f78dd29acc974383edd5b38b
Win32.OnionDuke.B	61562	16ed0cf7d5d132c46d6fb9d3ff723b7a
Win32.Pay2Key.B	5646445	2c0c40c3c0441a915730638c7ae10c44
Win32.Powerstats	1222290	ae855e7c228e5649e3935e91f221158f
Win32.RedDelta	569637	761d0c4b38af5d9fed3b1abd2b8ebed2
Win32.Reveton	1081376	a4ddc950732f827525bceceab231e145
Win32.Sality	29679	a3088cf1da75891293bf1c94995169dd
Win32.ShadowHammer	14061	247f551bda2c681332e24bc3da8fdb56
Win32.Sofacy.A	59957	93d9031291d074ad45ea3dd132410144
Win32.SofacyCarberp	100541	003b2c09b78db2b8ddbf043051dd7d00
Win32.StrongPity	122204	645a5380540b135ad0d839b2f7366df7
Win32.Stuxnet.A.Duqu-C-Media	13295	1c35a13a0da0ea687100a7b273f9a5af
Win32.Stuxnet.B.Duqu-Realtek	56467	c5d82342430cfd542bc287b178025e5f
Win32.Taleret	414684	f4405083fd152923747f361ad6feb606
Win32.TransparentTribe.B	10939495	4561598ba221d6fcdaf3fa4446c9a8ca
Win32.Triton	44407	a7c5efdc9d55d815d32c66809efb3100
Win32.Turla	433019	806f6d32670941011893896069bcedd8
Win32.Turla.v1	98849	3c2dfe47b8f5f80055a382309f3622d0
Win32.Unclassified	379608	aed0d25af8dbf513a0cdba6c1a144ec0
Win32.Unknown_SpectreMeltdown	100292	98b25e3dcf67d0c5362fbb0514564fdf
Win32.Unnamed_SpecMelt	1323652	fdf42a36b17c95acd91a8f0ba267a80a
Win32.ValeforBeta	503788	d42340fd31d8d10603aee565ce85d708
Win32.VBS.APT34Dropper	3830	4e5bc720dc68ed5b3bb4fb73ede729cf
Win32.Vobfus	292329	3f0a46b1febcd33e25da42f6b491a273
Win32.WannaPeace	489151	96593e22646caafcd606ae75f816c989

Win32.XAgent	112527	112b866457f79cb4ab0ba930f2a47069
Win32.ZeroCleare	574286	a63b2fdcc4a32b00d5b475f56f9c4ac9
Win32.ZeusVM	426970	b73f3134bb5ee95d8deb3abdfc9b1263
Win32.Zurgop	28077	bd5159dc10fdf21f4df234af07217023
Win32Dircrypt.Trojan.Ransom.ABZ	781366	96af3ed583a3d47b9e1225a351ffe268
Win64.NukeSped	305252	c92ac2a5a2c6d2b1826a0bbc1a56b01a
Win64.Trojan.GreenBug	393117	9ed25c4a6ae99f9eb28fd3c654109006
WinX.HiddenCobra.Supply	3934133	86e65806c9bcd187efd2be734e76626e
WinX.OperationDianxun	3407359	5151bba22faaa41af6b96287c82b3351
WinX. SignSight	59930778	56a49cd8423289dbcf0b4c84c46eef7f
WinX.SUNBURST	1018148	31b50e5fbf4b123b6f32fc28edd0ba86
WM.Alliance.A	20778	dee8a79fedb64ad879383529e3036df0
WM.Cap.A	5720	e822534759a9f10957d849c66c52226f
WM.Concept.A	6052	1200d732295ddbdf339a62a59c94dace
WM.Concept.S	4634	6028f32a62baeb99e9540da9d7c40860
WM.Minimal.AB	6551	22abfbcbb300de8ee94c05c9c641c058
WM.NJ_WMVCK2_T	22941	dc34265295c5fe482fedd05a5b363894
WM.Npad.A	6716	c6e8114152471773c7e58324c5ef4c0c
WMIGhost	542956	104ac67de3c26a8970c407bd4a1bbef6
Yankee_Doodle.2881.A	25686	0dfb32542cb1e81e9c00fc3c0faec6ad
Yankee_Doodle.2997	17654	92b77f8e832f84bee37d4c54e1013440
Yankee_Login.3052	3389	fd3b1441c5f268d8ec3a016357b20c1e
Yaunch.2537	4100	f30af5a24e767575867fbdf70a35ac61
Yeke.1204	10892	204e0f107d02be491496f309b0759d9d
ZeroAccess	169686	25b0dfbf8d762ddf965d62760af11895
Zerolocker	260325	9eaecc8bd2d980fb2f3f38a249021a97
ZeusBankingVersion_26Nov2013	175338	858e2aed6ba9b096679967da540d40c3
ZeusGameover_Feb2014	830608	79f9d8468f9d354dfc1a90be4aa0157f
Zherkov.1958	4185	1bb8ef5a86ecd4e0da83a8410ceb4c87
Zherkov.2970	6400	88b3c9691b93d17866a91e9cb42008ca

D.3 Benign Binaries

Table D.3: Benign processes used in the sandbox environment during the execution of Malware. All these executables were retrieved from the cnet.com Apps for Windows category representing popular windows applications.

Name	**File Size (bytes)**	**MD5 Checksum**
ActivateApplication.exe	24008	20631875fba6fb648c7d324001fb9b75
adaminstall.exe	34564	26b5bf08641a27dfaefa286ea01d2914
AddInProcess.exe	42168	6b1643fda04002af6288c2b77c81f8fc
AddInProcess32.exe	42176	b0c6e3f8709401c521d2f44c159d4a9c
AddInUtil.exe	42152	db83b86ff82c2bdc2f28a42b2b3f82e6

AnimationCompiler.exe	16384	8859bb442653b4e966d01234edab23f5
antialias.exe	43960	14565ce4c5a73944db06e53cb5b4ca71
ApacheMonitor.exe	36352	962257481a219e1eb8d103e9afb63401
applygeo.exe	17408	f5c59ab2867b2b29799ade2e5ca49607
aspnetca.exe	39472	dad7c63c827130ae3a18a1db8ddccdd8
aspnet_compiler.exe	36864	aac10cd188ee6bb486290c3d59aeb96d
aspnet_wp.exe	48296	f8a4a783f0466535f604f7d6b05fba11
AtBroker.exe	41472	7231ac444b2b25c4441416a1ef758feb
BackgroundTransferHost.exe	34304	3b08e65385b6e0fa42a8e039f51b56f1
basename.exe	20992	d7214bcb23a788ac99983987062c834f
blur.exe	44016	7d89a1da4463470c47658e85af5bb2ff
BootExpCfg.exe	36462	17b615753e3f637b8f4265118213a6ab
bsqldb.exe	39936	ab91c57e9c091c0933be687b197f23a7
bsqlodbc.exe	33792	128adfa3adc10fc5f5f35d04f5f1756c
bthudtask.exe	36864	0857bd25d96b83a849b08e72517dc2cb
ByteCodeGenerator.exe	38400	c9963015ed6d17e8a2ab253d56aa974c
cat.exe	24064	f991df9fc0d585c66055c5ea57e4215a
cdpreference.exe	41984	870b6972e20b38f559bbbec1119c23ae
checkerboard.exe	49448	21c1bf71d37653b529b79393e6ac796f
chmod.exe	37376	9cdd506670f469cf3a12dfd23a09d9b1
cipher.exe	39936	1ea15973a58783ab884aad0e3866e872
cldr-plurals.exe	30720	c746f41ac81449ac2c635acf48ff4243
CloudNotifications.exe	49080	cd954bffd3607faec6e46d92b65e255f
cmmon32.exe	37376	cdab139df2cde1eefec0c3d1c0b76157
cmp.exe	27136	b3df3e6358cecf36c4d0fcedb6a354c5
color-enhance.exe	42512	ecae96e2f1269a1a00745033c7c253e8
color-to-alpha.exe	48584	54c6e1e90ec9417f06398725f6927ba7
colorify.exe	47232	57fba2babbbb5218492cf3ae629d0649
comm.exe	23040	69f26507c13964a866daf63917b1933d
Common.DBConnection.exe	38600	7fc385ad92c2aa9d2894ed535380fb16
Common.DBConnection64.exe	37576	fd846602cf0c8895ba7dc808c2787258
compact.exe	35840	cf167d60de9fb21f5691833831860f7b
ComputerDefaults.exe	36864	ce97af285fc29115582dc611e7151dbf
ConflictViewer.exe	31832	69e644a0892ced9d78bde31a2adf1e9f
contrast-normalize.exe	42304	9175b16b232b6d6717cf5a2c391fbc9c
contrast-stretch-hsv.exe	42464	d86237871a4fc590b3ec4fc763508406
contrast-stretch.exe	42336	cde9233233420a94e41779de99e6c505
CorelPS2PDF.exe	41840	64049215787885cce5079ff6ad7b1b32
CortanaDssServer.exe	41984	c507cf73e270f095941efd26d8fcac96
CptInstall.exe	34992	7dd9d2054b8e4662a0f36b56c004f623
CrashDumpWatcher.exe	19968	1522abd6a6de92b2d80a67e3305d873e
credwiz.exe	37888	86a26663dbbdbb38c18d6c88c26fbd2f
crop-auto.exe	44880	dc4c49115cd23457de6e3c7ec9c70d8e

crop-zealous.exe	43952	30db74e63b4e69b1c19783d1508e34de
cs2cs.exe	47616	20d7401b8ff111054fa5b5d41606e790
cttunesvr.exe	37376	108f492dda2a06797d5b611ee5b4ab29
curve_keygen.exe	16896	6e2a60b9dda787caca6bd9033655d055
cut.exe	30720	97a35d3fabb539404200e136ba75ce24
CVAnalysisService.exe	30704	71630a9e2e15b2f58d2e5c04c9582560
CVCollectionService.exe	30200	4ba9aa9cea8759ca34ab95b5690af61b
cvtres.exe	42432	b90e3c77ac49d767369dd3da98db26ce
d2u.exe	30720	56765f722d5237eb50c1a3280f8cebb2
DatabaseMail.exe	40024	2c00d31db451e97a8139108d4514ce6f
datacopy.exe	30720	1cf1fc9c217268bd3c931c1dc4b2a03d
dbus-launch.exe	19456	8ab0d1038a72860aba7e311bbaec7eeb
dbus-monitor.exe	28672	42f2d3e4a6ba33a29a30a5d4f472a293
dbus-test-tool.exe	27136	95372cb0f9885ae80c49c42c5be9a26b
dbus-update-activation-environment.exe	20992	83baff9586ae7c89a91af994244c898a
ddodiag.exe	38912	702a25be2e1f4da37d85b7b4e1916764
defncopy.exe	31744	6e840051853eb86c217b8f93e9cb692b
deinterlace.exe	47744	c2975e24fe5c12a816e4aaf00c490433
dialer.exe	36864	8229d13ffe3187e3a85ad47e526ce157
diff3.exe	28672	221f5fd8d3febfdcdfde83e3418aff00
dirname.exe	21504	7d590258f9c7828c955677a47adb98ec
dmcfghost.exe	38912	149f65628d5dbb619a928773bbb15b2d
dos2unix.exe	30720	56765f722d5237eb50c1a3280f8cebb2
dumpbin.exe	25032	a658334aadd24b35970eef1c420f5a32
dw20.exe	33936	9593e3977f2e11f34241c788a4adfc67
Easeware.ConfigLanguageFromSetup.exe	17776	964ca74d6b2114742c546bdd535a0b5c
echo.exe	22016	ae1d555afcd9cc5839d745c6f63c172f
edge-laplace.exe	44016	7bf7a556210c00236329f97d497611c6
editbin.exe	25032	c1160a76cf19d6bec0afb03ac6fca6ba
elfedit.exe	41472	75907929df192674fab44cfa343bc88a
empty.exe	9728	523d5c39f9d8d2375c3df68251fa2249
engrave.exe	49080	8d1937b56309d2820e03c388d1ba281b
env.exe	20480	a820cff76e664a38c2b262d6b4b989a5
error-mode.exe	48128	5e096efdc9b90981552afb15afb56d78
eventcreate.exe	40448	26a13fbba483291b5e14329fc1f04837
evntwin.exe	39926	37f80016cb1614be66c4863f38f2d2ed
extrac32.exe	34816	8069d11277df55336c99960cc3544fce
false.exe	14848	d40e907c1bdfc0458f709c0e72d54752
fc-cache.exe	26112	ead913a792d1e48d45b04c15d1562aa6
fc-cat.exe	24576	df510f4c865acfe2bb9113a2540fa34f
fc-list.exe	23040	74eec1c2bbdb6363779b15283ccaf41f
fc-match.exe	23040	8e66a8d8b0742fed3900a46a58d459db
fc-pattern.exe	23040	5eb18afa17445021a8fd60b539f98531

fc-query.exe	22528	8198aacf524fa42671c2dbe269893a1c
fc-scan.exe	23040	a84b6f90a5a3d4d54fde7714d72dde1b
fc-validate.exe	23552	a590a027f0646580443d8218d1700837
file-desktop-link.exe	40840	5544e1c78f732ac50e6c4a208ae9c4a0
file-faxg3.exe	48456	34c4c03b8512eb605482b8b82f17c196
file-glob.exe	41984	f1af69f1dfbbf87bd0c20cc94c219e49
file-header.exe	43360	92770d6dfce1363f9c086fc2e329998d
file-jp2-load.exe	46608	b5577f50c9c2f4a4b2b21b9c3eb7d92d
file-pcx.exe	49432	9f9ffa82fd64aea9b6e08530d885952a
file-pix.exe	45528	b80f9142071f80cef88bc2df5bd42e33
file-uri.exe	47240	982cefbc22583ab743f9f9fe8f1656a3
file.exe	15872	eada9895a974315872599c0b9267cf45
findstr.exe	34304	4c408063941cadff2ec81da2ac5234de
FixSqlRegistryKey_ia64.exe	46936	97929f19f4d1ceff260be032f6756b63
FixSqlRegistryKey_x64.exe	46952	ff1488a9d3e544dc608e8174f0d44f26
fodhelper.exe	48640	6276ab316b1a3b93676553d3f6b7e14b
fold.exe	23552	1886b7b3d70c38e566beec76eb9cfbdf
forfiles.exe	41984	e6ba997f0d10d691832bf30d84b00018
freebcp.exe	30208	8b6b74e95c39eb15f472d218d4e9f310
fsynonym.exe	19968	169219a50af2729b113e57dcbc493fdb
ftp.exe	49152	2357757e8c5c8862db0d2f0e6e74dacc
funzip.exe	19968	18fc9f6013143ad8453181a05b08319c
gdbus.exe	40448	6d26f9babc2f7e3a896abce74004e025
geod.exe	40960	45d864eafc719f0c51aea2311226734b
geotifcp.exe	45056	5089750b4aaf1e2f629f92071c5a1e84
gimptool-2.0.exe	43392	256ae3175865b29ea4e249e90517f803
gio-querymodules.exe	18944	b39668365ab11e437da681db5a9f5bd9
glib-compile-resources.exe	40448	bf6bf69e9a5fadd1feeb461e5c4a3141
glib-compile-schemas.exe	46080	1af18daa37849d24fd7219ca1c58df35
glib-genmarshal.exe	38400	702328530725121be03e6914c122605f
gobject-query.exe	18944	10802d94a1e700e8bd7a15d48c821a86
gpscript.exe	38400	018063c4e232e12d3d4a3e9a4c902adb
gradient-map.exe	43568	63092a0561a7e31b37c8fa67a8f5bf57
GraphCmd.exe	33232	45ea58abb3ff8dd0412c50e7292a6720
gresource.exe	22016	fb7d74a4817d77ddaeacd67d0577fc87
grpconv.exe	36864	a4cbbdc2a2c96f81d7ddba45ff32ce47
gsettings.exe	29184	f9f582aac351098c42ce89f860400e8b
gsl-histogram.exe	21504	dda79b2c0e30ed61477fb91681f4002f
gsl-randist.exe	26112	0d2080ab21dddfd7f8038f8aef27e00e
gspawn-win64-helper-console.exe	18944	fd5113a78098e5991039f286b0ad9b07
gspawn-win64-helper.exe	19456	4a888d62f94675f6d660e68d30bdb040
guidgen.exe	40392	240ff55d89bac595ba7a5db872fbba2e
guillotine.exe	44032	7e4b3372bd199d9326af33ee46090163

h5debug.exe	26112	fee5e85f029455b7ca3189168948ab88
h5repart.exe	23040	0ec708e5feeac8f020b2243ad6221c96
head.exe	33280	f6fa0f06ca7f40bd1ae3f99d35c0dad3
hostname.exe	20992	f1c97579c8d2459c3dfc405f20d9b851
httpd.exe	23040	d134556e0087ca2eef99d78c31858000
hwrcomp.exe	48128	9f24e6975b423f50959b713c73737381
icacls.exe	37376	ac14d9063658ceb6366a4231df08da92
iconv.exe	42496	b0e3461f1fbbfdeee0fccf64e943b121
id.exe	23552	a8cfc5a349e25f16291b96d43b255fe3
IisExpressAdminCmd.exe	48504	bdac365c55879f554c915f385757e0c8
InstallUtil.exe	40112	1e6a2d071e075f8b6a2d70680da70330
IntelliTrace.exe	42968	3df3e5983bc39a55b879f485535890d1
invgeod.exe	40960	45d864eafc719f0c51aea2311226734b
ipconfig.exe	34816	523ca6c1845c5fe955e62dc217d3176e
join.exe	33280	11818a674a4a35bbab62ff20880c24f2
jpatch.exe	23680	89300d4498bdf4094c362c555f83b22a
klist.exe	37376	eb4ff64bbb06418654ec367eab0138b9
ksetup.exe	38400	3a69781fea49805f6672e77386c3b7b8
LaunchWinApp.exe	36864	2d7f63f6153781785f5b79be59fe39ef
lc.exe	38720	8815977a898f2449227cec209a0b2d9d
ldifde.exe	35842	2b0e59a6cb4734f7c79df5d916ec0fa2
lesskey.exe	13312	c047e888b95166f9bbb0ad2c4c1ad117
lib.exe	25016	6dc87cffe46c4017bb258c26e3f5818e
listgeo.exe	21504	4ece45806723eaa89375c694c608cb2d
LocationNotificationWindows.exe	41472	a9ea4d3b98f72ea1172f43c3bbbeeb37
LockAppHost.exe	39649	a7a69b3cee7b6afe896e2d1db26f29b6
LogCollector.exe	41778	3b96398910c395b087fa438d3633befa
mac2unix.exe	30720	56765f722d5237eb50c1a3280f8cebb2
makecat.exe	34248	960e0ce7b5129ba0e76d60f12c9b7eb6
makehm.exe	34240	dbe718b95522abdcac6e2d7dd72b2396
malias.exe	43850	4c7f9407a143580984bf21b69c788177
MathWorks_Privileged_Operation.exe	42904	51724c328ddee39dccd9f4f43356a964
MATLABStartupAccelerator.exe	47104	30fc4392b98f2b8728460ad078c49465
max-rgb.exe	45776	1692d3197535d18f03885af7c11d0233
md5sum.exe	39424	2cafb4312bfb686e115736d1e11523bf
mfpmp.exe	35656	3fc92dd93579b34d3b1718a8139dda44
mgmtclassgen.exe	48040	616b8b23e3806855dc9f7b3891b9b43f
Microsoft.Mashup.Container.exe	26848	7bac1830b397243f2b17fc20f9e24f40
mkdir.exe	29696	1b5847e6d3ac11b692aec9d93d704b09
msoev.exe	48840	ba9e1a90e875237b3dbde10c9b2e0335
msotd.exe	48840	80093e4e55b0fffc7ecb9eea4a24f52c
MSTest.exe	26048	037b5753894b9ede0bd539bb0e185f51
nad2bin.exe	20480	d7431622d35dca1608834e0db5f07a74

NEInstX64.exe	30672	39b50ec6573034802090750969d89d4f
netcfg.exe	37376	ce79e340483bf9c675293cdf5c3dfc46
NetEvtFwdr.exe	38912	ed61448e50cbcde808a93e1e0aca9c37
NETSTAT.EXE	34304	e0c8ac0427fe71d9c5913fadb5d19d1b
newmail.exe	23080	b4a3abd703a1bc33eecafc4dd8056478
octave-4.2.1.exe	39424	f51285e5f200ce06bd3bbaa4ecb7805f
octave-cli-4.2.1.exe	19968	2ccef34a9f4e9d8a58d477b49227540d
octave-cli.exe	19968	2ccef34a9f4e9d8a58d477b49227540d
octave-config-4.2.1.exe	38400	79cd4997890e99e4c6bb9d5c2d29dac6
octave-config.exe	38400	79cd4997890e99e4c6bb9d5c2d29dac6
octave-gui.exe	20480	bedc4144a5d53d3b944f106a852a7d92
octave.exe	39424	f51285e5f200ce06bd3bbaa4ecb7805f
od.exe	46592	49a7de999da9e27c4b3b3408e23488ed
oisicon.exe	34144	2d40a8f6b34cf2d1085519cbd6cacf33
pango-view.exe	41984	89d657570247a2e30549d7057b4680f5
paste.exe	23040	4f7efbcc11809906b07b56c01192bb42
pcaui.exe	39424	5e3df859e975a5ccf10b3fa452d118c0
PdbDownloader.exe	36320	03906f92b79ae41c9c82b079fe328594
perl.exe	36352	fc932153ab59d92fd10f7518d8b39e73
perl5.8.8.exe	26112	0e122206fe7f526f039c39c4b864e9e2
perlglob.exe	12800	af045228f2902e9f7a6525e63b780eee
php-win.exe	29184	e8656b078a3b961cd12deaa90bfe1429
pmgrant.exe	30640	aaa30eaf7e7bfd95ce180ad72a0dce58
pmsort.exe	17376	69e1469445f7318ecbf075cab1e79a39
PreviewProcessingService.exe	48216	7d66fcea235acbc376b7da04ea0284e6
printf.exe	31744	3065dc38aed8fabba1e141a1baec3c95
procedure-browser.exe	40824	df054b79e91fa7d1b0b5ffebb1b81bfb
proquota.exe	34304	ea8e0ae71f8c0725a67cf0d87a7413a5
pstoedit.exe	17408	c8ecf9869a5fa63f05c40485ab00fd9e
pvk2pfx.exe	35272	be348a2dbf72c80e4d0c813922de8606
pwd.exe	22528	4ac1b5904c6ff1b8a92af1d36c2ece88
pwlauncher.exe	35328	4d42b5373aaa4c5556e4ae18339170eb
python.exe	26624	423d7330dc73f2a3487c80cce0ddc99f
qconvex.exe	25600	dbb74968fff91a6f49ecc70d4b9ce35e
qdelaunay.exe	25088	e055619df82ae215d2fe705d0bb7609b
qhalf.exe	25088	562429713e0315957b939823472aa8aa
qhull.exe	28160	770765733324684153949b923761c577
QTAgent.exe	31688	840f6985d0090d918bd1002486ebf653
QTAgent32.exe	31696	0e6488a69c82459d053877f76f76e4da
QTAgent_40.exe	30672	734f17346c9e513dbd51e0a2ed2fc5c2
qvoronoi.exe	24576	0e41796ad7635a6af0b96cc023d76a38
rasphone.exe	34816	5344828ed618b34b21c248b957ef494c
RdpSa.exe	41472	567347d7169e759e8706f5ac7eb99f62

rdrleakdiag.exe	39424	9780787cb62d3b1f2b286a4ec63867ab
red-eye-removal.exe	48440	238bc2d18b25b341f61bfc76c3c7ee6c
RegAsm.exe	49152	c288a1a69655a5c1379300eb6896b575
regini.exe	41984	99726ec50c03d9967a4cab85c5fa5bc3
relog.exe	38912	392c971fe99412652f3e3e94cc557e57
rmdir.exe	21504	b63e1f6b7492dc2f34d2d411e0eea7e1
rotate.exe	47168	9c53ca8032c8c98c01a2540573358049
rrinstaller.exe	38912	e704fd5c3a7381f495fd969fff46ccef
RSReportHost.exe	36952	7dd9e21941b5538ba666c4d627634123
runonce.exe	36352	4e970be80b6e380a5cdfcaf0624f51f3
SCANPST.EXE	39328	40206037af16cb0240c7458e431d60c1
sdchange.exe	41472	d993e9b7a450aada63f8b68d26db69c8
sdiff.exe	32768	9a6b43fae6cc6cf565d842cf410c2414
SecEdit.exe	37888	c83bd8df769bca0489502db1e9624264
semi-flatten.exe	41584	ea235197228eefbd6a95e844781dcebd
SetEHCIKey.exe	42200	a199bb131aa3d1e387e99ed774cdaa54
SETLANG.EXE	33152	8401118a57db08496b03766012a3f47d
SettingSyncHost.exe	39144	89d0150c4d4c217ed104a2ef925f84d5
sfc.exe	36864	1efcca9077f7fbbad8fd89de4f1e43e2
shutdown.exe	33792	09670e5e523e9b366799e82729719658
sleep.exe	21504	f952d3abccac96e82a3a6ea6a70d4cef
split.exe	32768	931c6a015f6b35cc58f8277fd492e11a
SqlLocalDB.exe	38488	5aea7be9528cbf8acfcfc7715837b367
sqlmonitor.exe	29784	50cbc0d02e0d9546c8e100fe4af7e98c
ssi_standalone.exe	46424	87a4873ae23008661c329b86272efff9
StoreAdm.exe	38736	5f1b19185ad6bed4ebe6b4c667b33685
svchost.exe	35176	a412dedac6a1ff7ba06feb3b6725495e
sxstrace.exe	36864	829612b22963538294021274626dc012
sysprep.exe	35603	988102d589f03141518b9e4eebb28dfe
T4VSHostProcess.exe	23528	5778a9274f811a8609e8f56660907a41
tail.exe	46080	15dc9dd6d9f8bc76cbc4e3035dd0a81c
TDEnvCleanup.exe	27096	29c24902fe968094551c99fc46802f6a
tee.exe	21504	44223173ecc1d2cb79c7911acb080b27
test-geos.exe	16896	c5d57d1c24ba85d6cb707a73c94e708c
TfsAdmin.exe	35272	6ad7e36df2187ea1216d78211d0ce626
TFSDeleteProject.exe	31728	56ef53c5afbc211c13ab7a30cfae8ae4
TFSFieldMapping.exe	40128	6f396d2ee7fce875bc421cef7599a96c
threshold-alpha.exe	46688	2577f4b17a2994089ae8c01991a8726b
ThumbnailExtractionHost.exe	34816	cf6e609ebdbc3dfd4579ff4b398952f7
tile-glass.exe	49328	4ff2a463f79211e2462db27809721875
tile-seamless.exe	43400	7c64758e1709c57f7663cc2517e01b0e
toast.exe	20760	c9f9ba7574a4d640303272a53574eaae
touch.exe	45056	100df8f04d71b2604848d3eb524dc1ed

tr.exe	38912	5f436dbcc09c8afe4124ababb6be416c
true.exe	14848	3bd48c82b244603d749c3e017f2606fd
TsWpfWrp.exe	35480	26a67f58d21ece5650005f43393c19fc
twain.exe	47832	938061332b308edba250f5fc9c9855c5
tzutil.exe	49152	fb91f0ecc65a75721c05af1c952bbeb4
u2d.exe	30720	7b6d28032a9b765548f9b64dc179df60
uname.exe	21504	a223b5d2aeb97b22589675f266771114
undname.exe	27056	4ab7f428d09175025a4c55a08b0d2eb6
uniq.exe	29696	2f2d2a2c92faab42fae052f0b5989a10
unix2dos.exe	30720	7b6d28032a9b765548f9b64dc179df60
unix2mac.exe	30720	7b6d28032a9b765548f9b64dc179df60
unlodctr.exe	35328	247300bd8410c35e52a227802b428967
UpgradeResultsUI.exe	35328	f2dd834ad4bfb8bc0cd3f17cc340d6d4
upnpcont.exe	41472	f37dd4ec75bc3bb11c9129ee18511291
urlget.exe	23040	bddd56951af250c6c9778a215e3e4642
urlproxy.exe	9360	43a8d07d5661f67ae09a603dba48d514
UserAccountBroker.exe	35664	2aef72adb15ad1266cfdc240572b8cf1
UserControlTestContainer.exe	36880	d005ea840060488e4f8375ed89987444
value-invert.exe	42504	c70452a0cdc74029e4a318e661757257
vmUpdateLauncher.exe	35544	d59270a57da4c1394b086815cfd036b6
vshost.exe	22472	378dd10936aaff40eb34d94dc29f2366
VSPerfASPNetCmd.exe	46056	0677ae4ce16df45fa997cb4a92280cdb
vssphost4.exe	29136	78c86b834102279fadb2bf8dd7dbab8f
vstest.executionengine.appcontainer.exe	33800	f14ad85c0facd59a9c83c108330dd58b
vstest.executionengine.appcontainer.x86.exe	33824	ce38a3985a61577c54bda8cb024695b2
vstest.executionengine.clr20.exe	22000	2bbbde68f82af7032dcb26b2de99b1e4
VSTestConfig.exe	30168	05c657ec96d53f4d60a1218f9f5df7ea
VSTestVideoRecorder.exe	40928	ca13cceb2cefd46fc8d5d4fb39b410d3
VSTST-FileConverter.exe	25080	b8c22e0364ca56cfd9e9e4be8c64714d
waitfor.exe	39936	f82e3591111e5b9d9c70636390237d65
wc.exe	24576	c3245673c595de58a05fc8409980f19a
web-browser.exe	41368	f55ff54f710442012838d3431c1cd3f3
where.exe	34304	7af6ef1e27edb0659a3410c67f85b427
wiawow64.exe	38912	77fd7120839b9eae88f47010d113a285
win7appid.exe	21504	05bec43bc5dcace27fc17a43839b611e
WindowsActionDialog.exe	39936	f0699407b677a90257a5707994d2d631
winrs.exe	49152	26fae9900410053ddc14abdbdf90f856

Appendix E

Ensemble Model Complexity

This section provides an explanation for the model space complexities for the models defined in Section 4.2.5. Note, Fig. E.1 shows the simplified version of the expressions.

Table E.1: Ensemble method space complexities ($\mathcal{O}$) based on the methodology described in Section sec:4:sandbox-ensemble.

Model	**Space Complexity** $\mathcal{O}(\cdot)$
Feature Concat	$E \cdot [m + \{l-1\} \cdot E + 2] + l \cdot n^2$
Embedding Concat	$T \cdot (E \cdot [m + \{l-1\} \cdot E] + E \cdot 2 + l \cdot n^2)$
Hetero GNN	$T \cdot (E \cdot [m + \{l-1\} \cdot E] + E \cdot 2) + l \cdot n^2$

First, the space complexity of the *Feature Concat* model in Fig. E.1 which is a function of several model inputs. The first layer of the GNN model is of size $m \cdot E$ to account for the size of the input feature vector m and the embedding dimension E. Depending on the number of layers in the GNN l, we have $l-1$ number of layers of size $E \cdot E$ since the size of the hidden dimensions was not varied throughout the model. The $l-1$ term accounts for the final hidden layer, which is where the term $E \cdot 2$ comes from. The final term $l \cdot n^2$ accounts for the attention coefficient matrices which are stored between each process, which is $n \cdot n$ in the worst case if all nodes are connected to each other. Since attention coefficients are computed for each layer in the GNN, we have l number of these matrices.

Next, the space complexity of the *Embedding Concat* model in Fig. E.1 is a small variation to the prior explanation. T number of models are created in parallel with a similar space complexity as *Feature Concat* with the exception of the final layer which is of size $T \cdot E \cdot 2$ since its a concatenation of the event types. We are also computing attention coefficients for all edge types models in parallel, so $T \cdot l \cdot n^2$ accounts for this which can be prohibitively expensive for higher node count n.

The *Hetero GNN* model architecture accounts for this deficiency by computing attention coefficients for only a single process matrix $n \cdot n$ for all edge types. Since the Hetero GNN works by sharing outputs from previous layers of the GNN through duplication of the layers, it does not require creating T number of models in parallel and computing independent attention coefficients as well. This small change results in a significant savings in storage, which is illustrated in Fig. E.1 for all three models showing $\mathcal{O}$ with increasing embedding dimension E. We see a similar behavior in Fig. E.2 with increasing input feature vector size m.

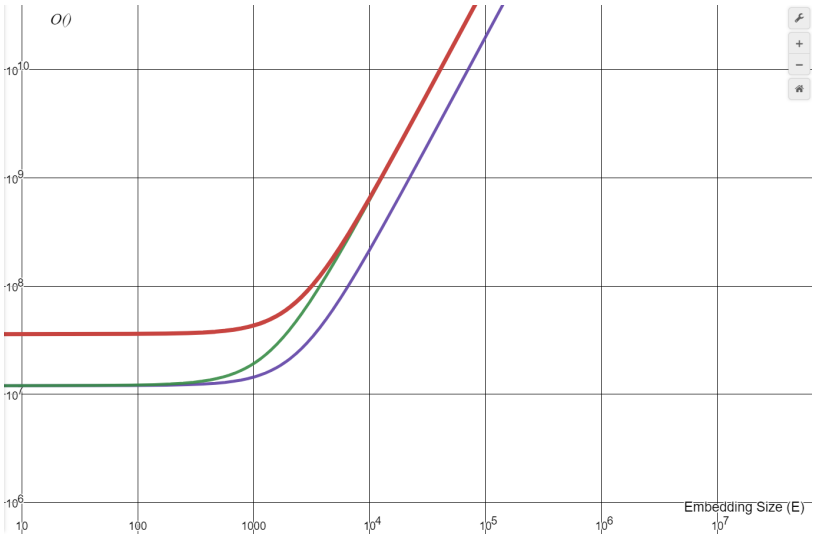


Figure E.1: Big $\mathcal{O}(\cdot)$ as a function of Embedding Size E for the feature concat, embed concat, and hetero architectures. Image plotted for a $m = 2000$, $T = 3$, $l = 3$, and $n = 2000$.

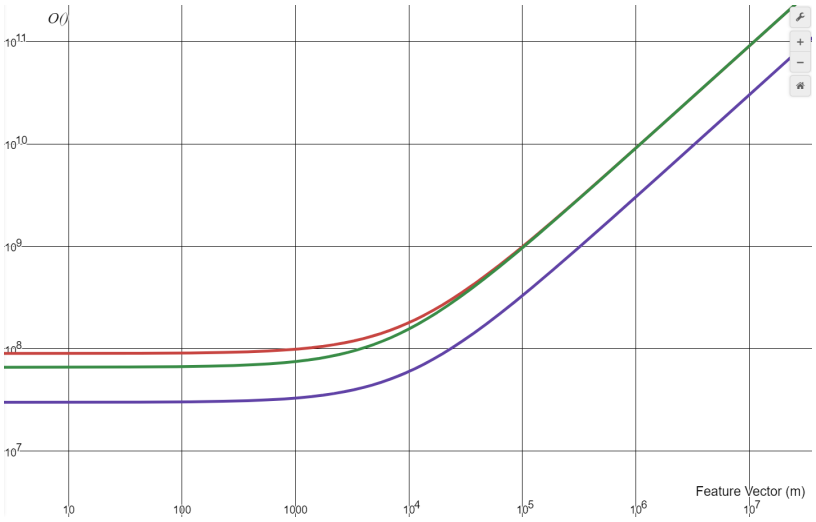


Figure E.2: Big $\mathcal{O}(\cdot)$ as a function of Input Feature Vector size m for the feature concat, embed concat, and hetero architectures. Image plotted for a $E = 3000$, $T = 3$, $l = 3$, and $n = 2000$.

Bibliography

[1] Kenneth Brezinski and Ken Ferens. "Metamorphic Malware and Obfuscation -A Survey of Techniques, Variants and Generation Kits". In: *Security and Communication Networks* (2023). URL: Submitted;%20Under%20review.

[2] Kenneth Brezinski and Ken Ferens. "An Adaptive Tribal Topology for Particle Swarm Optimization". In: *Transactions on Computational Science & Computational Intelligence*. Advances in Security, Networks, and Internet of Things. Accepted; In Press. Springer Nature, 2020. ISBN: ISSN: 2569-7072.

[3] Kenneth Brezinski and Ken Ferens. "Transformers - Malware in Disguise". In: *Transactions on Computational Science & Computational Intelligence*. Advances in Security, Networks, and Internet of Things. Springer Nature, 2021.

[4] Petar Veličković et al. "Graph Attention Networks". In: *arXiv:1710.10903 [cs, stat]* (Feb. 2018). arXiv: 1710.10903. URL: http://arxiv.org/abs/1710.10903 (visited on 08/15/2021).

[5] Kenneth Brezinski, Michael Guevarra, and Ken Ferens. "Population Based Equilibrium in Hybrid SA/PSO for Combinatorial Optimization: Hybrid SA/PSO for Combinatorial Optimization". en. In: *International Journal of Software Science and Computational Intelligence (IJSSCI)* 12.2 (2020). Publisher: IGI Global, pp. 74–86. ISSN: 1942-9045. DOI: 10.4018/IJSSCI.2020040105. URL: https://www.igi-global.com/article/population-based-equilibrium-in-hybrid-sapso-for-combinatorial-optimization/252216 (visited on 09/07/2020).

[6] Kenneth Brezinski and Ken Ferens. "Cognitive Hybrid PSO/SA Combinatorial Optimization". In: *2019 IEEE 18th International Conference on Cognitive Informatics Cognitive Computing (ICCI*CC)*. July 2019, pp. 389–393. DOI: 10.1109/ICCICC46617.2019.9146062.

[7] Kenneth Brezinski and Ken Ferens. "Complexity-Based Lambda Layer for Time Series Prediction". In: *Symposium on Classification, Clustering, and Data-mining*. Krakow, Poland: IEEE Conference Publishing Services (CPS), 2021. DOI: 10.1109/CEC45853.2021.9504995.

[8] Kenneth Brezinski and Ken Ferens. "Complexity-Based Convolutional Neural Network for Malware Classification". In: *Symposium on Cyber Warfare, Cyber Defense & Cyber Security (CSCI- ISCW)*. Las Vegas, USA: IEEE Conference Publishing Services (CPS), 2020. DOI: 10.1109/CEC45853.2021.9504995.

[9] Kenneth Brezinski and Ken Ferens. "Sandy Toolbox: A Framework for Dynamic Malware Analysis and Model Development". In: *Transactions on Computational Science & Computational Intelligence*. Advances in Security, Networks, and Internet of Things. Springer Nature, 2021.

[10] Kenneth Brezinski and Ken Ferens. *Incorporating Topological Complexity into a Multilayer Perceptron*. Mar. 2022. DOI: 10.13140/RG.2.2.27132.00641.

[11] Kenneth Brezinski and Ken Ferens. "Graph-Oriented Modelling of Process Event Activity for the Detection of Malware". In: *Transactions on Computational Science & Computational Intelligence*. Advances in Artificial Intelligence and Applied Cognitive Computing. Springer Nature, July 2023.

[12] Weijie Han et al. "MalInsight: A systematic profiling based malware detection framework". en. In: *Journal of Network and Computer Applications* 125 (Jan. 2019), pp. 236–250. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2018.10.022. URL: http://www.sciencedirect.com/science/article/pii/S1084804518303503 (visited on 01/08/2020).

[13] Alisdair A. Gillespie. *Cybercrime: Key Issues and Debates*. en. Google-Books-ID: 0N4sCgAAQBAJ. Routledge, July 2015. ISBN: 978-1-134-66033-9.

[14] Malwarebytes. *2020 State of Malware Report*. White Paper. Santa Clara, CA: Malwarebytes Labs, Feb. 2020. URL: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf (visited on 03/25/2020).

[15] Kaspersky. *Kaspersky Security Bulletin 2019, Statistics*. White Paper. Moscow, 2019.

[16] Ethan M. Rudd et al. "A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions". In: *IEEE Communications Surveys Tutorials* 19.2 (2016), pp. 1145–1172. ISSN: 2373-745X. DOI: 10.1109/COMST.2016.2636078.

[17] Hewlett Packard Enterprise. *HPE Security Research Cyber Risk Report 2016*. en. Tech. rep. Feb. 2016, p. 96.

[18] Verizon. *2019 Data Breach Investigations Report*. White Paper. New York, NY, USA: Verizon, 2019. URL: https://enterprise.verizon.com/resources/executivebriefs/2019-dbir-executive-brief-emea.pdf (visited on 04/25/2020).

[19] IBM. *2019 Cost of Data Breach Report*. White Paper. Armonk, NY: IBM, 2019.

[20] Denise Berard. *DDoS Breach Costs Rise to over $2M for Enterprises finds Kaspersky Lab Report — Kaspersky*. Feb. 2018. URL: https://usa.kaspersky.com/about/press-releases/2018_ddos-breach-costs-rise-to-over-2m-for-enterprises-finds-kaspersky-lab-report (visited on 04/30/2020).

[21] FireEye. *APT28: A window Into Russia's Cyber Espionage Operations?* White Paper. Milpitas, CA: FireEye, 2014.

[22] CyberEdge. *2020 Cyberthreat Defense Report*. White Paper. Annapolis, MD: CyberEdge Group, 2020.

[23] Symantec. *ISTR 20 - Internet Security Threat Report*. White Paper Vol. 20. Mountain View, CA: Symantex Corporation World Headquarters, Apr. 2015. URL: https://its.ny.gov/sites/default/files/documents/symantec-internet-security-threat-report-volume-20-2015-social_v2.pdf (visited on 03/20/2020).

[24] CSO. *2018 U.S. State of Cybercrime*. en-US. Survey. Framingham, MA: IDG Communications, 2018. URL: https://www.idg.com/tools-for-marketers/2018-u-s-state-of-cybercrime/ (visited on 04/30/2020).

[25] PwC. *Key findings from The Global State of Information Security Survey 2018*. en. White Paper. London, UK: International Data Group Inc., 2017, p. 22.

[26] Crowdstrike. *Crowdstrike Global Threat Report 2020*. White Paper. Sunnyvale, CA, 2020. URL: https://go.crowdstrike.com/rs/281-OBQ-266/images/Report2020CrowdStrikeGlobalThreatReport.pdf (visited on 03/25/2020).

[27] Mimecast. *The State of Email Security Report*. White Paper. London, UK: Mimecast, 2019. URL: https://www.mimecast.com/the-state-of-email-security-2019/?utm_medium=SEMPPC&utm_source=GooglePPC&utm_campaign=7013l000000TmCuAAK&utm_term=cyber%20security&gclid=Cj0KCQjwm9D0BRCMARIsAIfvfIYjY42W93p1B1fEHUKouH3oo0zoJ0VoHNjumDWylDlq4l1EFXJeQwcB#section2 (visited on 04/26/2020).

[28] Malwarebytes. *White Hat, Black Hat and the Emergence of the Gray Hat: The True Costs of Cybercrime*. White. Black Diamond, WA: Osterman Research, Inc., Aug. 2019. URL: https://go.malwarebytes.com/OstermanCostofCybercrimeQ3FY19_GLOBAL_Blog.html (visited on 05/25/2020).

[29] Brightstarlang Wanswett and Hemanta Kumar Kalita. "The Threat of Obfuscated Zero Day Polymorphic Malwares: An Analysis". In: *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*. ISSN: 2472-7555. Dec. 2015, pp. 1188–1193. DOI: 10.1109/CICN.2015.230.

[30] N. Innab, Eman Alomairy, and L. Alsheddi. "Hybrid System Between Anomaly Based Detection System and Honeypot to Detect Zero Day Attack". en. In: *undefined* (2018). URL: https://www.semanticscholar.org/paper/Hybrid-System-Between-Anomaly-Based-Detection-and-Innab-Alomairy/5ca4f26c185280593e6b5a287cbe809f53610077 (visited on 10/12/2021).

[31] Farag Azzedin, Husam Suwad, and Zaid Alyafeai. "Countermeasureing Zero Day Attacks: Asset-Based Approach". In: *2017 International Conference on High Performance Computing & Simulation (HPCS)* (2017). DOI: 10.1109/HPCS.2017.129.

[32] Ammar Almomani et al. *Phishing Dynamic Evolving Neural Fuzzy Framework for Online Detection Zero-day Phishing Email*. arXiv:1302.0629 [cs]. Feb. 2013. DOI: 10.48550/arXiv.1302.0629. URL: http://arxiv.org/abs/1302.0629 (visited on 01/23/2023).

[33] ESG. *ESG Master Survey Results: 2020 Technology Spending Intentions Survey*. en-us. Survey. Milford, MA, 2020. URL: https://www.esg-global.com/research/esg-master-survey-results-2020-technology-spending-intentions-survey (visited on 04/30/2020).

[34] Ratinder Kaur and Maninder Singh. "Efficient hybrid technique for detecting zero-day polymorphic worms". In: *2014 IEEE International Advance Computing Conference (IACC)*. ISSN: null. Feb. 2014, pp. 95–100. DOI: 10.1109/IAdCC.2014.6779301.

[35] Pengzhen Ren et al. *A Survey of Deep Active Learning*. arXiv:2009.00236 [cs, stat]. Dec. 2021. DOI: 10.48550/arXiv.2009.00236. URL: http://arxiv.org/abs/2009.00236 (visited on 01/14/2023).

[36] Webroot. *Webroot Threat Report*. White Paper. Broomfield, CO: OpenText, 2020. URL: https://mypage.webroot.com/rs/557-FSI-195/images/2020%20Webroot%20Threat%20Report_US_FINAL.pdf (visited on 04/27/2020).

[37] Harlan Carvey. "Chapter 6 - Malware Detection". en. In: *Windows Forensic Analysis Toolkit (Fourth Edition)*. Ed. by Harlan Carvey. Boston: Syngress, Jan. 2014, pp. 169–209. ISBN: 978-0-12-417157-2. DOI: 10.1016/B978-0-12-417157-2.00006-0. URL: http://www.sciencedirect.com/science/article/pii/B9780124171572000060 (visited on 05/02/2020).

[38] Michael Gregg. *Build Your Own Security Lab: A Field Guide for Network Testing*. en. Google-Books-ID: pE8b1sso7OUC. John Wiley & Sons, Aug. 2010. ISBN: 978-0-470-37947-9.

[39] John Aycock. *Computer Viruses and Malware*. en. Springer, Sept. 2006. ISBN: 978-0-387-34188-0.

[40] Elizabeth Seamans and Thomas Alexander. *Fast Virus Signature Matching on the GPU*. en. Library Catalog: developer.nvidia.com. 2007. URL: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-35-fast-virus-signature-matching-gpu (visited on 05/03/2020).

[41] Daniel Mellado. *IT Security Governance Innovations: Theory and Research: Theory and Research*. en. Google-Books-ID: R7WeBQAAQBAJ. IGI Global, Sept. 2012. ISBN: 978-1-4666-2084-1.

[42] Devendra Kumar Mahawer and A. Nagaraju. "Metamorphic malware detection using base malware identification approach". en. In: *Security and Communication Networks* 7.11 (2014). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.869, pp. 1719–1733. ISSN: 1939-0122. DOI: 10.1002/sec.869. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.869 (visited on 05/15/2020).

[43] Santanam Raghu, Sethumadhavan M, and Virendra Mohit. *Cyber Security, Cyber Crime and Cyber Forensics: Applications and Perspectives: Applications and Perspectives*. en. Google-Books-ID: A7Tr3ztALBAC. Idea Group Inc (IGI), Dec. 2010. ISBN: 978-1-60960-125-6.

[44] Shucheng Yu, Wenjing Lou, and Kui Ren. "Chapter 15 - Data Security in Cloud Computing". en. In: *Handbook on Securing Cyber-Physical Critical Infrastructure*. Ed. by Sajal K. Das, Krishna Kant, and Nan Zhang. Boston: Morgan Kaufmann, Jan. 2012, pp. 389–410. ISBN: 978-0-12-415815-3. DOI: 10.1016/B978-0-12-415815-3.00015-7. URL: http://www.sciencedirect.com/science/article/pii/B9780124158153000157 (visited on 05/03/2020).

[45] Shailesh Kumar Shivakumar. "5 - Securing Enterprise Web Application". en. In: *Architecting High Performing, Scalable and Available Enterprise Web Applications*. Ed. by Shailesh Kumar Shivakumar. Boston: Morgan Kaufmann, Jan. 2015, pp. 159–178. ISBN: 978-0-12-802258-0. DOI: 10.1016/B978-0-12-802258-0.00005-6. URL: http://www.sciencedirect.com/science/article/pii/B9780128022580000056 (visited on 05/03/2020).

[46] Essam Al Daoud, Iqbal H Jebril, and Belal Zaqaibeh. "Computer Virus Strategies and Detection Methods". en. In: *Int. J. Open Problems Compt. Math.* 1.2 (2008), p. 8.

[47] McAfee. *McAfee Network Security Platform 9.1.x*. en. Product Guide. Santa Clara, CA: McAfee, Aug. 2017. URL: https://docs.mcafee.com/bundle/network-security-platform-9.1.x-product-guide/page/GUID-D00D67EA-5EAE-4461-ACFC-A2B2A78C3E50.html.

[48] Felix Kreuk et al. "Deceiving End-to-End Deep Learning Malware Detectors using Adversarial Examples". In: *arXiv:1802.04528 [cs]* (Feb. 2018). arXiv: 1802.04528. URL: http://arxiv.org/abs/1802.04528 (visited on 01/04/2019).

[49] Aditya Rohan, Kanad Basu, and Ramesh Karri. "Can Monitoring System State + Counting Custom Instruction Sequences Aid Malware Detection?" In: *2019 IEEE 28th Asian Test Symposium (ATS)* (2019). DOI: 10.1109/ATS47505.2019.00007.

[50] Da Lin and Mark Stamp. "Hunting for undetectable metamorphic viruses". en. In: *J Comput Virol* 7.3 (Aug. 2011), pp. 201–214. ISSN: 1772-9904. DOI: 10.1007/s11416-010-0148-y. URL: https://doi.org/10.1007/s11416-010-0148-y (visited on 02/05/2020).

[51] Ilsun You and Kangbin Yim. "Malware Obfuscation Techniques: A Brief Survey". In: *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. ISSN: null. Nov. 2010, pp. 297–300. DOI: 10.1109/BWCCA.2010.85.

[52] Daniel C. Park, Haidar Khan, and Bulent Yener. "Generation & Evaluation of Adversarial Examples for Malware Obfuscation". In: *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)* (2019). DOI: 10.1109/ICMLA.2019.00210.

[53] Lukas Durfina and Dusan Kolar. "C source code obfuscator". In: *Kybernetika* 48.3 (2012), pp. 494–501.

[54] Peter Szor. *The Art of Computer Virus Research and Defense*. en. OCLC: ocm57382448. Upper Saddle River, NJ: Addison-Wesley, 2005. ISBN: 978-0-321-30454-4.

[55] B Jyothi Kumar et al. "Logistic regression for polymorphic malware detection using ANOVA F-test". In: *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*. ISSN: null. Mar. 2017, pp. 1–5. DOI: 10.1109/ICIIECS.2017.8275880.

[56] Annie H. Toderici and Mark Stamp. "Chi-squared distance and metamorphic virus detection". en. In: *J Comput Virol Hack Tech* 9.1 (Feb. 2013), pp. 1–14. ISSN: 2263-8733. DOI: 10.1007/s11416-012-0171-2. URL: https://doi.org/10.1007/s11416-012-0171-2 (visited on 01/07/2020).

[57] Xufang Li, Peter Kok Keong Loh, and Freddy Tan. "Mechanisms of Polymorphic and Metamorphic Viruses". In: *2011 European Intelligence and Security Informatics Conference* (2011). DOI: 10.1109/EISIC.2011.77.

[58] Annie Hii Toderici. "Chi-Squared Distance and Metamorphic Virus Detection". en. Master of Science. San Jose, CA, USA: San Jose State University, Jan. 2012. DOI: 10.31979/etd.j3nz-gjtr. URL: https://scholarworks.sjsu.edu/etd_theses/4177 (visited on 05/18/2020).

[59] Cristián Ignacio Vidal Barría et al. "Obfuscation procedure based in dead code insertion into crypter". In: *2016 6th International Conference on Computers Communications and Control (ICCCC)* (2016). DOI: 10.1109/ICCCC.2016.7496733.

[60] Kenneth Brezinski and Ken Ferens. "Metamorphic Malware and Obfuscation: A Survey of Techniques, Variants, and Generation Kits". In: *Security and Communication Networks* 2023.2 (Sept. 2023). Ed. by Konstantinos Rantos. Publisher: Hindawi, p. 8227751. ISSN: 1939-0114. DOI: 10.1155/2023/8227751. URL: https://doi.org/10.1155/2023/8227751.

[61] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. "Evolution of Computer Virus Concealment and Anti-Virus Techniques: A Short Survey". en. In: 7.6 (2010), p. 9.

[62] Mahim Patel. "Similarity Tests for Metamorphic Virus Detection". In: *Master's Projects* (Apr. 2011). DOI: https://doi.org/10.31979/etd.6j9f-9drn. URL: https://scholarworks.sjsu.edu/etd_projects/175.

[63] Wing Wong and Mark Stamp. "Hunting for metamorphic engines". en. In: *J Comput Virol* 2.3 (Nov. 2006), pp. 211–229. ISSN: 1772-9890, 1772-9904. DOI: 10.1007/s11416-006-0028-7. URL: http://link.springer.com/10.1007/s11416-006-0028-7 (visited on 05/12/2020).

[64] Mohamed R. Chouchane and Arun Lakhotia. "Using engine signature to detect metamorphic malware". In: *WORM '06*. 2006. DOI: 10.1145/1179542.1179558.

[65] Ran Jin et al. "Normalization towards Instruction Substitution Metamorphism Based on Standard Instruction Set". In: *2007 International Conference on Computational Intelligence and Security Workshops (CISW 2007)*. Dec. 2007, pp. 795–798. DOI: 10.1109/CISW.2007.4425615.

[66] Zhang Yu-jia and Pang Jian-min. "A New Compile-Time Obfuscation Scheme for Software Protection". In: *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* (2016). DOI: 10.1109/CyberC.2016.10.

[67] Pascal Junod et al. "Obfuscator-LLVM – Software Protection for the Masses". In: *2015 IEEE/ACM 1st International Workshop on Software Protection* (2015). DOI: 10.1109/SPRO.2015.10.

[68] Anne Kayem. *Information Security in Diverse Computing Environments*. en. Google-Books-ID: RZ5_BAAAQBAJ. IGI Global, June 2014. ISBN: 978-1-4666-6159-2.

[69] Sujandharan Venkatachalam and Mark Stamp. "Detecting Undetectable Metamorphic Viruses". en. In: Nevada, 2011, p. 7.

[70] Donabelle Baysa, Richard M. Low, and Mark Stamp. "Structural entropy and metamorphic malware". en. In: *J Comput Virol Hack Tech* 9.4 (Nov. 2013), pp. 179–192. ISSN: 2263-8733. DOI: 10.1007/s11416-013-0185-4. URL: https://doi.org/10.1007/s11416-013-0185-4 (visited on 03/20/2020).

[71] Ashu Sharma and S. K. Sahay. "Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey". en. In: *IJCA* 90.2 (Mar. 2014), pp. 7–11. ISSN: 09758887. DOI: 10.5120/15544-4098. URL: http://research.ijcaonline.org/volume90/number2/pxc3894098.pdf (visited on 05/18/2020).

[72] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. *Camouflage in Malware: from Encryption to Metamorphism*. en. Library Catalog: www.semanticscholar.org. 2012. URL: https://www.semanticscholar.org/paper/Camouflage-in-Malware%3A-from-Encryption-to-Rad-Masrom/c48621d1e598a1c2ed28496ca1a2da3a98d08ed5 (visited on 03/27/2020).

[73] Philippe Beaucamps. "Advanced Polymorphic Techniques". en. In: *International Journal of Computer and Information Engineering* 1.10 (2007), p. 12.

[74] Peter Szor and Peter Ferrie. *Hunting for Metamorphic*. White Paper. Cupertino, CA: Symantec, 2001. URL: https://crypto.stanford.edu/cs155old/cs155-spring09/papers/viruses.pdf (visited on 06/04/2020).

[75] D. Spinellis. "Reliable identification of bounded-length viruses is NP-complete". en. In: *IEEE Trans. Inform. Theory* 49.1 (Jan. 2003), pp. 280–284. ISSN: 0018-9448. DOI: 10.1109/TIT.2002.806137. URL: http://ieeexplore.ieee.org/document/1159781/ (visited on 06/04/2020).

[76] Jean-Marie Borello and Ludovic Mé. "Code obfuscation techniques for metamorphic viruses". en. In: *J Comput Virol* 4.3 (Aug. 2008), pp. 211–220. ISSN: 1772-9904. DOI: 10.1007/s11416-008-0084-2. URL: https://doi.org/10.1007/s11416-008-0084-2 (visited on 06/04/2020).

[77] Mark Stamp. *Information Security Principles and Practice*. 2nd Edition. John Wiley & Sons, Ltd, Apr. 2011.

[78] Milan Tuba, Shyam Akashe, and Amit Joshi. *Information and Communication Technology for Sustainable Development: Proceedings of ICT4SD 2018*. en. Google-Books-ID: Z0efDwAAQBAJ. Springer, June 2019. ISBN: 9789811371660.

[79] M.E. Saleh, A.B. Mohamed, and A.A. Nabi. "Eigenviruses for metamorphic virus recognition". In: *IET Information Security* 5.4 (Dec. 2011), pp. 191–198. ISSN: 1751-8717. DOI: 10.1049/iet-ifs.2010.0136.

[80] Rodelio G. Fiñones and Richard Fernandez. *Solving the metamorphic puzzle*. 2006. URL: https://www.virusbulletin.com/virusbulletin/2006/03/solving-metamorphic-puzzle (visited on 06/04/2020).

[81] Sudarshan Madenur Sridhara. "METAMORPHIC WORM THAT CARRIES ITS OWN MORPHING ENGINE". en. Master of Science. San Jose, CA, USA: San Jose State University, Apr. 2012. DOI: 10.31979/etd.unb6-nb8s. URL: https://scholarworks.sjsu.edu/etd_projects/240 (visited on 06/07/2020).

[82] Sayali Deshpande, Younghee Park, and Mark Stamp. "Eigenvalue analysis for metamorphic detection". en. In: *J Comput Virol Hack Tech* 10.1 (Feb. 2014), pp. 53–65. ISSN: 2263-8733. DOI: 10.1007/s11416-013-0193-4. URL: https://doi.org/10.1007/s11416-013-0193-4 (visited on 02/07/2020).

[83] Zahra Bazrafshan et al. *A survey on heuristic malware detection techniques*. en. Library Catalog: www.semanticscholar.org. 2013. URL: /paper/A-survey-on-heuristic-malware-detection-techniques-Bazrafshan-Hashemi/2e7caf022dcc7a64f897af1b661ba73e4f296ea0 (visited on 03/16/2020).

[84] Pavel V. Zbitskiy. "Code mutation techniques by means of formal grammars and automatons". en. In: *J Comput Virol* 5.3 (Aug. 2009), pp. 199–207. ISSN: 1772-9904. DOI: 10.1007/s11416-009-0121-9. URL: https://doi.org/10.1007/s11416-009-0121-9 (visited on 05/20/2020).

[85] Teja Tamboli, Thomas H. Austin, and Mark Stamp. "Metamorphic code generation from LLVM bytecode". en. In: *J Comput Virol Hack Tech* 10.3 (Aug. 2014), pp. 177–187. ISSN: 2263-8733. DOI: 10.1007/s11416-013-0194-3. URL: https://doi.org/10.1007/s11416-013-0194-3 (visited on 05/18/2020).

[86] Sudarshan Madenur Sridhara and Mark Stamp. "Metamorphic worm that carries its own morphing engine". en. In: *J Comput Virol Hack Tech* 9.2 (May 2013), pp. 49–58. ISSN: 2263-8733. DOI: 10.1007/s11416-012-0174-z. URL: https://doi.org/10.1007/s11416-012-0174-z (visited on 01/07/2020).

[87] David Harley, Robert Slade, and Urs Gattiker. *Viruses Revealed*. en. Google-Books-ID: UZI52Ohp8E0C. McGraw Hill Professional, Dec. 2002. ISBN: 978-0-07-222818-2.

[88] EC-Council. *Ethical Hacking and Countermeasures: Threats and Defense Mechanisms*. en. Google-Books-ID: Xr9sCgAAQBAJ. Nelson Education, Sept. 2009. ISBN: 978-1-111-78699-1.

[89] Hacked Team. *hackedteam/core-packer*. original-date: 2015-07-06T11:16:17Z. June 2020. URL: https://github.com/hackedteam/core-packer (visited on 06/13/2020).

[90] Jaime Lloret Mauri et al. *Security in Computing and Communications: Second International Symposium, SSCC 2014, Delhi, India, September 24-27, 2014. Proceedings*. en. Google-Books-ID: K7peBAAAQBAJ. Springer, Aug. 2014. ISBN: 978-3-662-44966-0.

[91] Eric Filiol. *Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the bradley virus*. en. report. INRIA, 2004, p. 10. URL: https://hal.inria.fr/inria-00070748 (visited on 06/11/2020).

[92] J.R.ziz Al-Enezi, M.F. Abbod, and S. Alsharhan. "Artificial Immune Systems – Models, Algorithms and Applications". en. In: *Artificial Immune Systems* (2010), p. 14.

[93] Dr Mafaz Mohsin Khalil Al-Anezi. "Generic Packing Detection Using Several Complexity Analysis for Accurate Malware Detection". en. In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 5.1 (2014). Number: 1 Publisher: The Science and Information (SAI) Organization Limited. ISSN: 2156-5570. DOI: 10.14569/IJACSA.2014.050102. URL: https://thesai.org/Publications/ViewPaper?Volume=5&Issue=1&Code=IJACSA&SerialNo=2 (visited on 05/26/2020).

[94] Xinjian Ma et al. "Using multi-features to reduce false positive in malware classification". In: *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*. ISSN: null. May 2016, pp. 361–365. DOI: 10.1109/ITNEC.2016.7560382.

[95] Allan Liska and Timothy Gallo. *Ransomware: Defending Against Digital Extortion*. en. Google-Books-ID: sYKRDQAAQBAJ. "O'Reilly Media, Inc.", Nov. 2016. ISBN: 978-1-4919-6783-6.

[96] Herbert Bos, Fabian Monrose, and Gregory Blanc. *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan,November 2-4, 2015. Proceedings*. en. Google-Books-ID: ACjUCgAAQBAJ. Springer, Oct. 2015. ISBN: 978-3-319-26362-5.

[97] Minh Hai Nguyen et al. "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning". en. In: *Computers & Security* 76 (July 2018), pp. 128–155. ISSN: 0167-4048. DOI: 10.1016/j.cose.2018.02.006. URL: http://www.sciencedirect.com/science/article/pii/S0167404818300889 (visited on 05/26/2020).

[98] Ivan Sorokin. "Comparing files using structural entropy". en. In: *J Comput Virol* 7.4 (June 2011), p. 259. ISSN: 1772-9904. DOI: 10.1007/s11416-011-0153-9. URL: https://doi.org/10.1007/s11416-011-0153-9 (visited on 03/20/2020).

[99] Felipe N. Ducau et al. "Automatic Malware Description via Attribute Tagging and Similarity Embedding". In: *arXiv:1905.06262 [cs, stat]* (Jan. 2020). arXiv: 1905.06262. URL: http://arxiv.org/abs/1905.06262 (visited on 10/12/2021).

[100] R.P. Lippmann et al. "Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation". In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. Vol. 2. Jan. 2000, 12–26 vol.2. DOI: 10.1109/DISCEX.2000.821506.

[101] Wenke Lee and Salvatore J. Stolfo. "A framework for constructing features and models for intrusion detection systems". In: *ACM Trans. Inf. Syst. Secur.* 3.4 (Nov. 2000), pp. 227–261. ISSN: 1094-9224. DOI: 10.1145/382912.382914. URL: https://doi.org/10.1145/382912.382914 (visited on 05/24/2020).

[102] Atilla Özgür and Hamit Erdem. *A review of KDD99 dataset usage in intrusion detection and machine learning between 2010 and 2015*. en. Tech. rep. e1954v1. ISSN: 2167-9843. PeerJ Inc., Apr. 2016. DOI: 10.7287/peerj.preprints.1954v1. URL: https://peerj.com/preprints/1954 (visited on 10/12/2021).

[103] Matthew V. Mahoney and Philip K. Chan. "An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection". en. In: *Recent Advances in Intrusion Detection*. Ed. by Gerhard Goos et al. Vol. 2820. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 220–237. ISBN: 978-3-540-40878-9 978-3-540-45248-5. DOI: 10.1007/978-3-540-45248-5_13. URL: http://link.springer.com/10.1007/978-3-540-45248-5_13 (visited on 05/14/2020).

[104] Mahbod Tavallaee et al. "A detailed analysis of the KDD CUP 99 data set". en. In: *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. Ottawa, ON, Canada: IEEE, July 2009, pp. 1–6. ISBN: 978-1-4244-3763-4. DOI: 10.1109/CISDA.2009.5356528. URL: http://ieeexplore.ieee.org/document/5356528/ (visited on 05/14/2020).

[105] John McHugh. "Testing Intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory". In: *ACM Trans. Inf. Syst. Secur.* 3.4 (Nov. 2000), pp. 262–294. ISSN: 1094-9224. DOI: 10.1145/382912.382923. URL: https://doi.org/10.1145/382912.382923 (visited on 05/24/2020).

[106] Nour Moustafa and Jill Slay. "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)". In: *2015 Military Communications and Information Systems Conference (MilCIS)*. Nov. 2015, pp. 1–6. DOI: 10.1109/MilCIS.2015.7348942.

[107] Wesam Bhaya and Mehdi Ebady Manaa. "A Proactive DDoS Attack Detection Approach Using Data Mining Cluster Analysis". en. In: *Journal of Next Generation Information Technology* 5.4 (Nov. 2014), p. 12.

[108] Ali Shiravi et al. "Toward developing a systematic approach to generate benchmark datasets for intrusion detection". en. In: *Computers & Security* 31.3 (May 2012), pp. 357–374. ISSN: 0167-4048. DOI: 10.1016/j.cose.2011.12.012. URL: http://www.sciencedirect.com/science/article/pii/S0167404811001672 (visited on 05/14/2020).

[109] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization". In: *ICISSP*. 2018. DOI: 10.5220/0006639801080116.

[110] Markus Ring et al. "A Survey of Network-based Intrusion Detection Data Sets". en. In: *Computers & Security* 86 (Sept. 2019). arXiv: 1903.02460, pp. 147–167. ISSN: 01674048. DOI: 10.1016/j.cose.2019.06.005. URL: http://arxiv.org/abs/1903.02460 (visited on 05/14/2020).

[111] L. Nataraj et al. "Malware images: visualization and automatic classification". In: *Proceedings of the 8th International Symposium on Visualization for Cyber Security*. VizSec '11. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, July 2011, pp. 1–7. ISBN: 978-1-4503-0679-9. DOI: 10.1145/2016904.2016908. URL: https://doi.org/10.1145/2016904.2016908 (visited on 05/13/2020).

[112] Daniel Gibert, Carles Mateu, and Jordi Planes. "An End-to-End Deep Learning Architecture for Classification of Malware's Binary Content". en. In: *Artificial Neural Networks and Machine Learning – ICANN 2018*. Ed. by Věra Kůrková et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 383–391. ISBN: 978-3-030-01424-7. DOI: 10.1007/978-3-030-01424-7_38.

[113] Niket Bhodia et al. "Transfer Learning for Image-Based Malware Classification". en. In: *arXiv:1903.11551 [cs, stat]* (Jan. 2019). arXiv: 1903.11551. URL: http://arxiv.org/abs/1903.11551 (visited on 05/04/2020).

[114] Trung Kien Tran, Hiroshi Sato, and Masao Kubo. "Image-Based Unknown Malware Classification with Few-Shot Learning Models". In: *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. Nov. 2019, pp. 401–407. DOI: 10.1109/CANDARW.2019.00075.

[115] Mahmoud Kalash et al. "Malware Classification with Deep Convolutional Neural Networks". In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. ISSN: 2157-4960. Feb. 2018, pp. 1–5. DOI: 10.1109/NTMS.2018.8328749.

[116] S. Akarsh et al. “A Detailed Investigation and Analysis of Deep Learning Architectures and Visualization Techniques for Malware Family Identification”. en. In: *Cybersecurity and Secure Information Systems: Challenges and Solutions in Smart Environments*. Ed. by Aboul Ella Hassanien and Mohamed Elhoseny. Advanced Sciences and Technologies for Security Applications. Cham: Springer International Publishing, 2019, pp. 241–286. ISBN: 978-3-030-16837-7. DOI: 10.1007/978-3-030-16837-7_12. URL: https://doi.org/10.1007/978-3-030-16837-7_12 (visited on 10/12/2021).

[117] S. Akarsh et al. “Deep Learning Framework and Visualization for Malware Classification”. In: *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*. ISSN: 2575-7288. Mar. 2019, pp. 1059–1063. DOI: 10.1109/ICACCS.2019.8728471.

[118] Fargana Abdullayeva. “Malware Detection in Cloud Computing using an Image Visualization Technique”. In: *2019 IEEE 13th International Conference on Application of Information and Communication Technologies (AICT)*. ISSN: 2472-8586. Oct. 2019, pp. 1–5. DOI: 10.1109/AICT47866.2019.8981727.

[119] Bao Ngoc Vi et al. “Adversarial Examples Against Image-based Malware Classification Systems”. In: *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*. ISSN: 2164-2508. Oct. 2019, pp. 1–5. DOI: 10.1109/KSE.2019.8919481.

[120] Wai Weng Lo, Xu Yang, and Yapeng Wang. “An Xception Convolutional Neural Network for Malware Classification with Transfer Learning”. In: *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. ISSN: 2157-4960. June 2019, pp. 1–5. DOI: 10.1109/NTMS.2019.8763852.

[121] Antonio Nappa, M. Zubair Rafique, and Juan Caballero. “Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting”. en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by David Hutchison et al. Vol. 7967. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–20. ISBN: 978-3-642-39234-4 978-3-642-39235-1. DOI: 10.1007/978-3-642-39235-1_1. URL: http://link.springer.com/10.1007/978-3-642-39235-1_1 (visited on 05/13/2020).

[122] Platon Kotzias et al. “Certified PUP: Abuse in Authenticode Code Signing”. In: *CCS ’15*. 2015. DOI: 10.1145/2810103.2813665.

[123] Marcos Sebastián et al. “AVclass: A Tool for Massive Malware Labeling”. en. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Fabian Monrose et al. Vol. 9854. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 230–253. ISBN: 978-3-319-45718-5 978-3-319-45719-2. DOI: 10.1007/978-3-319-45719-2_11. URL: http://link.springer.com/10.1007/978-3-319-45719-2_11 (visited on 05/24/2020).

[124] Daniel Arp et al. *DREBIN: Efficient and Explainable Detection ofAndroid Malware in Your Pocket*. en. Technical Report. Georg-August Institute of Computer Science, 2013, p. 16.

[125] Michael Spreitzenbarth et al. “Mobile-sandbox: having a deeper look into android applications”. en. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*. Coimbra, Portugal: ACM Press, 2013, p. 1808. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480701. URL: http://dl.acm.org/citation.cfm?doid=2480362.2480701 (visited on 05/24/2020).

[126] Arash Habibi Lashkari et al. “Towards a Network-Based Framework for Android Malware Detection and Characterization”. In: *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. Aug. 2017, pp. 233–23309. DOI: 10.1109/PST.2017.00035.

[127] Muhammad Murtaz et al. “A framework for Android Malware detection and classification”. In: *2018 IEEE 5th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*. Nov. 2018, pp. 1–5. DOI: 10.1109/ICETAS.2018.8629270.

[128] Huy Kang Kim. *IoT network intrusion dataset*. en. Publisher: IEEE. Sept. 2019. URL: https://ieee-dataport.org/open-access/iot-network-intrusion-dataset (visited on 05/24/2020).

[129] Cyril Goutte. *Advances in Artificial Intelligence*. en. Google-Books-ID: oLzhDwAAQBAJ. Springer Nature, 1987. ISBN: 978-3-030-47358-7.

[130] Imtiaz Ullah and Qusay H. Mahmoud. “A Scheme for Generating a Dataset for Anomalous Activity Detection in IoT Networks”. en. In: *Advances in Artificial Intelligence*. Ed. by Cyril Goutte and Xiaodan Zhu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 508–520. ISBN: 978-3-030-47358-7. DOI: 10.1007/978-3-030-47358-7_52.

[131] urwithajit9. *ClaMP*. original-date: 2016-04-01T07:30:33Z. Jan. 2020. URL: https://github.com/urwithajit9/ClaMP (visited on 03/20/2020).

[132] Hyrum S. Anderson and Phil Roth. “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models”. In: *arXiv:1804.04637 [cs]* (Apr. 2018). arXiv: 1804.04637. URL: http://arxiv.org/abs/1804.04637 (visited on 03/20/2020).

[133] endgameinc. *ember*. original-date: 2018-04-11T17:48:15Z. May 2020. URL: https://github.com/endgameinc/ember (visited on 05/24/2020).

[134] Gideon Creech and Jiankun Hu. “A Semantic Approach to Host-Based Intrusion Detection Systems Using Contiguousand Discontiguous System Call Patterns”. In: *IEEE Transactions on Computers* 63.4 (Apr. 2014). Conference Name: IEEE Transactions on Computers, pp. 807–819. ISSN: 1557-9956. DOI: 10.1109/TC.2013.13.

[135] Gideon Creech and Jiankun Hu. “Generation of a new IDS test dataset: Time to retire the KDD collection”. In: *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. ISSN: 1558-2612. Apr. 2013, pp. 4487–4492. DOI: 10.1109/WCNC.2013.6555301.

[136] Gideon Creech. “Developing a high-accuracy cross platformHost-Based Intrusion Detection System capableof reliably detecting zero-day attacks”. PhD thesis. Australian Defence Force Academy: University of South Wales, 2014. URL: http://unsworks.unsw.edu.au/fapi/datastream/unsworks:11913/SOURCE02?view=true (visited on 05/14/2020).

[137] CERT. *CERT Insider Threat Center*. White Paper. Pittsburgh, PA: Carnegie Mellon University Software Engineering Institute, 2017. URL: https://resources.sei.cmu.edu/asset_files/Brochure/2017_015_001_452233.pdf (visited on 05/25/2020).

[138] Joshua Glasser and Brian Lindauer. "Bridging the Gap: A Pragmatic Approach to Generating Insider Threat Data". en. In: *2013 IEEE Security and Privacy Workshops*. San Francisco, CA: IEEE, May 2013, pp. 98–104. ISBN: 978-1-4799-0458-7. DOI: 10.1109/SPW.2013.37. URL: http://ieeexplore.ieee.org/document/6565236/ (visited on 05/25/2020).

[139] tisf. *ytisf/theZoo*. original-date: 2014-01-09T18:55:35Z. May 2020. URL: https://github.com/ytisf/theZoo (visited on 05/02/2020).

[140] Jusop Choi et al. "AMVG: Adaptive Malware Variant Generation Framework Using Machine Learning". In: *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*. ISSN: 2473-3105. Dec. 2019, pp. 246–24609. DOI: 10.1109/PRDC47002.2019.00055.

[141] Rakesh M. Verma and David J. Marchette. *Cybersecurity Analytics*. en. Google-Books-ID: _3zADwAAQBAJ. CRC Press, Nov. 2019. ISBN: 978-1-00-072789-0.

[142] Partha Sarathi Bhattacharjee, Abul Kashim Md Fujail, and Shahin Ara Begum. "A Comparison of Intrusion Detection by K-Means and Fuzzy C-Means Clustering Algorithm Over the NSL-KDD Dataset". In: *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. ISSN: 2473-943X. Dec. 2017, pp. 1–6. DOI: 10.1109/ICCIC.2017.8524401.

[143] Bhupendra Ingre and Anamika Yadav. "Performance analysis of NSL-KDD dataset using ANN". In: *2015 International Conference on Signal Processing and Communication Engineering Systems*. Jan. 2015, pp. 92–96. DOI: 10.1109/SPACES.2015.7058223.

[144] Lukman Hakim, Rahilla Fatma, and Novriandi. "Influence Analysis of Feature Selection to Network Intrusion Detection System Performance Using NSL-KDD Dataset". In: *2019 International Conference on Computer Science, Information Technology, and Electrical Engineering (ICOMITEE)*. Oct. 2019, pp. 217–220. DOI: 10.1109/ICOMITEE.2019.8920961.

[145] Nilesh Kunhare and Ritu Tiwari. "Study of the Attributes using Four Class Labels on KDD99 and NSL-KDD Datasets with Machine Learning Techniques". In: *2018 8th International Conference on Communication Systems and Network Technologies (CSNT)*. ISSN: 2329-7182. Nov. 2018, pp. 127–131. DOI: 10.1109/CSNT.2018.8820244.

[146] Gaurav Meena and Ravi Raj Choudhary. "A review paper on IDS classification using KDD 99 and NSL KDD dataset in WEKA". In: *2017 International Conference on Computer, Communications and Electronics (Comptelix)*. July 2017, pp. 553–558. DOI: 10.1109/COMPIX.2017.8004032.

[147] Rajesh Thomas and Deepa Pavithran. "A Survey of Intrusion Detection Models based on NSL-KDD Data Set". In: *2018 Fifth HCT Information Technology Trends (ITT)*. Nov. 2018, pp. 286–291. DOI: 10.1109/CTIT.2018.8649498.

[148] Chongzhen Zhang et al. "A Deep Learning Approach for Network Intrusion Detection Based on NSL-KDD Dataset". In: *2019 IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. ISSN: 2163-5056. Oct. 2019, pp. 41–45. DOI: 10.1109/ICASID.2019.8925239.

[149] Hafsa BENADDI, Khalil IBRAHIMI, and Abderrahim BENSLIMANE. "Improving the Intrusion Detection System for NSL-KDD Dataset based on PCA-Fuzzy Clustering-KNN". In: *2018 6th International Conference on Wireless Networks and Mobile Communications (WINCOM)*. Oct. 2018, pp. 1–6. DOI: 10.1109/WINCOM.2018.8629718.

[150] Kunal Singh and K. James Mathai. "Performance Comparison of Intrusion Detection System Between Deep Belief Network (DBN)Algorithm and State Preserving Extreme Learning Machine (SPELM) Algorithm". In: *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. Feb. 2019, pp. 1–7. DOI: 10.1109/ICECCT.2019.8869492.

[151] Luis Alfredo Álvarez Almeida and Juan Carlos Martinez Santos. "Evaluating Features Selection on NSL-KDD Data-Set to Train a Support Vector Machine-Based Intrusion Detection System". In: *2019 IEEE Colombian Conference on Applications in Computational Intelligence (ColCACI)*. June 2019, pp. 1–5. DOI: 10.1109/ColCACI.2019.8781803.

[152] Nerijus Paulauskas and Juozas Auskalnis. "Analysis of data pre-processing influence on intrusion detection using NSL-KDD dataset". In: *2017 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. Apr. 2017, pp. 1–5. DOI: 10.1109/eStream.2017.7950325.

[153] Ahmad Riza'ain Yusof et al. "Adaptive feature selection for denial of services (DoS) attack". In: *2017 IEEE Conference on Application, Information and Network Security (AINS)*. Nov. 2017, pp. 81–84. DOI: 10.1109/AINS.2017.8270429.

[154] Sireesha Rodda and Uma Shankar Rao Erothi. "Class imbalance problem in the Network Intrusion Detection Systems". In: *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*. Mar. 2016, pp. 2685–2688. DOI: 10.1109/ICEEOT.2016.7755181.

[155] Shivani Samdani and Sanyam Shukla. "A novel technique for converting nominal attributes to numeric attributes for intrusion detection". In: *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. July 2017, pp. 1–5. DOI: 10.1109/ICCCNT.2017.8204171.

[156] M. A. Jabbar and Shirina Samreen. "Intelligent network intrusion detection using alternating decision trees". In: *2016 International Conference on Circuits, Controls, Communications and Computing (I4C)*. Oct. 2016, pp. 1–6. DOI: 10.1109/CIMCA.2016.8053265.

[157] Zhifan Ye and Yuanlong Yu. "Network intrusion classification based on extreme learning machine". In: *2015 IEEE International Conference on Information and Automation*. Aug. 2015, pp. 1642–1647. DOI: 10.1109/ICInfA.2015.7279549.

[158] Fekadu Yihunie, Eman Abdelfattah, and Amish Regmi. "Applying Machine Learning to Anomaly-Based Intrusion Detection Systems". In: *2019 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. ISSN: 2642-8873. May 2019, pp. 1–5. DOI: 10.1109/LISAT.2019.8817340.

[159] T. Sree Kala and A. Christy. "An Intrusion Detection System using Opposition based Particle Swarm Optimization Algorithm and PNN". In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. Feb. 2019, pp. 184–188. DOI: 10.1109/COMITCon.2019.8862237.

[160] Zhang Xiaofeng and Hao Xiaohong. "Research on intrusion detection based on improved combination of K-means and multi-level SVM". In: *2017 IEEE 17th International Conference on Communication Technology (ICCT)*. ISSN: 2576-7828. Oct. 2017, pp. 2042–2045. DOI: 10.1109/ICCT.2017.8359987.

[161] Ayşe Gül and Eşref Adalı. "A feature selection algorithm for IDS". In: *2017 International Conference on Computer Science and Engineering (UBMK)*. Oct. 2017, pp. 816–820. DOI: 10.1109/UBMK.2017.8093538.

[162] Zhaomin Chen et al. "Autoencoder-based network anomaly detection". In: *2018 Wireless Telecommunications Symposium (WTS)*. Apr. 2018, pp. 1–5. DOI: 10.1109/WTS.2018.8363930.

[163] Ju-ho Woo, Joo-Yeop Song, and Young-June Choi. "Performance Enhancement of Deep Neural Network Using Feature Selection and Preprocessing for Intrusion Detection". In: *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. Feb. 2019, pp. 415–417. DOI: 10.1109/ICAIIC.2019.8668995.

[164] I.I. Mikhail et al. "Neural Nets to Detect Abnormal Traffic in Communication Networks". In: *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*. Oct. 2019, pp. 1–3. DOI: 10.1109/FarEastCon.2019.8933969.

[165] Elkhadir Zyad, Chougdali Khalid, and Benattou Mohammed. "An effective network intrusion detection based on truncated mean LDA". In: *2017 International Conference on Electrical and Information Technologies (ICEIT)*. Nov. 2017, pp. 1–5. DOI: 10.1109/EITech.2017.8255298.

[166] Jianlei Gao et al. "A Novel Intrusion Detection System based on Extreme Machine Learning and Multi-Voting Technology". In: *2019 Chinese Control Conference (CCC)*. ISSN: 1934-1768. July 2019, pp. 8909–8914. DOI: 10.23919/ChiCC.2019.8865258.

[167] Muna AL-Hawawreh, Nour Moustafa, and Elena Sitnikova. "Identification of malicious activities in industrial internet of things based on deep learning models". en. In: *Journal of Information Security and Applications* 41 (Aug. 2018), pp. 1–11. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2018.05.002. URL: http://www.sciencedirect.com/science/article/pii/S2214212617306002 (visited on 03/20/2020).

[168] Manjula C Belavagi and Balachandra Muniyal. "Game theoretic approach towards intrusion detection". In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. Vol. 1. Aug. 2016, pp. 1–5. DOI: 10.1109/INVENTIVE.2016.7823193.

[169] Sheraz Naseer et al. "Enhanced Network Anomaly Detection Based on Deep Neural Networks". In: *IEEE Access* 6 (2018). Conference Name: IEEE Access, pp. 48231–48246. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2863036.

[170] Santosh Kumar Sahu, Sauravranjan Sarangi, and Sanjaya Kumar Jena. "A detail analysis on intrusion detection datasets". In: *2014 IEEE International Advance Computing Conference (IACC)*. Feb. 2014, pp. 1348–1353. DOI: 10.1109/IAdCC.2014.6779523.

[171] Ripon Patgiri et al. "An Investigation on Intrusion Detection System Using Machine Learning". In: *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. Nov. 2018, pp. 1684–1691. DOI: 10.1109/SSCI.2018.8628676.

[172] Peilun Wu, Hui Guo, and Richard Buckland. "A Transfer Learning Approach for Network Intrusion Detection". In: *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*. Mar. 2019, pp. 281–285. DOI: 10.1109/ICBDA.2019.8713213.

[173] Shuqin Dong and Bin Zhang. "SVDD-based Network Traffic Anomaly Detection Method with High Robustness". In: *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*. Dec. 2019, pp. 1522–1526. DOI: 10.1109/ICCC47050.2019.9064205.

[174] Jonghoon Lee et al. "Cyber Threat Detection Based on Artificial Neural Networks Using Event Profiles". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 165607–165626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2953095.

[175] Mondher Essid and Farah Jemili. "Combining intrusion detection datasets using MapReduce". In: *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2016, pp. 004724–004728. DOI: 10.1109/SMC.2016.7844977.

[176] Zongren Li and Guanghui Yan. "A Spark Platform-Based Intrusion Detection System by Combining MSMOTE and Improved Adaboost Algorithms". In: *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*. ISSN: 2327-0594. Nov. 2018, pp. 1046–1049. DOI: 10.1109/ICSESS.2018.8663723.

[177] Khalil Ibrahimi and Mostafa Ouaddane. "Management of intrusion detection systems based-KDD99: Analysis with LDA and PCA". In: *2017 International Conference on Wireless Networks and Mobile Communications (WINCOM)*. Nov. 2017, pp. 1–6. DOI: 10.1109/WINCOM.2017.8238171.

[178] Preeti Singh and Amrish Tiwari. "An Efficient Approach for Intrusion Detection in Reduced Features of KDD99 Using ID3 and Classification with KNNGA". In: *2015 Second International Conference on Advances in Computing and Communication Engineering*. May 2015, pp. 445–452. DOI: 10.1109/ICACCE.2015.49.

[179] Jianguo Jiang et al. "A Novel Multi-classification Intrusion Detection Model Based on Relevance Vector Machine". In: *2015 11th International Conference on Computational Intelligence and Security (CIS)*. Dec. 2015, pp. 303–307. DOI: 10.1109/CIS.2015.81.

[180] Harshal A. Karande and Shyam S. Gupta. "Ontology based intrusion detection system for web application security". In: *2015 International Conference on Communication Networks (ICCN)*. Nov. 2015, pp. 228–232. DOI: 10.1109/ICCN.2015.44.

[181] Trupti Chandak, Chaitanya Ghorpade, and Sanyam Shukla. "Effective Analysis of Feature Selection Algorithms for Network based Intrusion Detection System". In: *2019 IEEE Bombay Section Signature Conference (IBSSC)*. July 2019, pp. 1–5. DOI: 10.1109/IBSSC47189.2019.8973103.

[182] Ritumbhra Uikey and Manasi Gyanchandani. "Survey on Classification Techniques Applied to Intrusion Detection System and its Comparative Analysis". In: *2019 International Conference on Communication and Electronics Systems (ICCES)*. July 2019, pp. 1451–1456. DOI: 10.1109/ICCES45898.2019.9002129.

[183] Noe Elisa, Longzhi Yang, and Nitin Naik. "Dendritic Cell Algorithm with Optimised Parameters Using Genetic Algorithm". In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. July 2018, pp. 1–8. DOI: 10.1109/CEC.2018.8477932.

[184] Abdulla Amin Aburomman and Mamun Bin Ibne Reaz. "Ensemble of binary SVM classifiers based on PCA and LDA feature extraction for intrusion detection". In: *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. Oct. 2016, pp. 636–640. DOI: 10.1109/IMCEC.2016.7867287.

[185] Yogita Danane and Thaksen Parvat. "Intrusion detection system using fuzzy genetic algorithm". In: *2015 International Conference on Pervasive Computing (ICPC)*. Jan. 2015, pp. 1–5. DOI: 10.1109/PERVASIVE.2015.7086963.

[186] Therese Bjerkestrand, Dimitris Tsaptsinos, and Eckhard Pfluegel. "An evaluation of feature selection and reduction algorithms for network IDS data". In: *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*. June 2015, pp. 1–2. DOI: 10.1109/CyberSA.2015.7166129.

[187] Tahir Mehmood and Helmi B Md Rais. "Machine learning algorithms in context of intrusion detection". In: *2016 3rd International Conference on Computer and Information Sciences (ICCOINS)*. Aug. 2016, pp. 369–373. DOI: 10.1109/ICCOINS.2016.7783243.

[188] Riaz Ullah Khan et al. "An Improved Convolutional Neural Network Model for Intrusion Detection in Networks". In: *2019 Cybersecurity and Cyberforensics Conference (CCC)*. May 2019, pp. 74–77. DOI: 10.1109/CCC.2019.000-6.

[189] Jingping Song, Zhiliang Zhu, and Chris Price. "A new evidence accumulation method with hierarchical clustering". In: *2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. July 2016, pp. 122–126. DOI: 10.1109/ICCCBDA.2016.7529545.

[190] Haiting Cui. "Research on Eliminating Abnormal Big Data based on PSO-SVM". In: *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. ISSN: 2381-0947. Oct. 2018, pp. 2460–2463. DOI: 10.1109/IAEAC.2018.8577474.

[191] Abhinaya Nagisetty and Govind P. Gupta. "Framework for Detection of Malicious Activities in IoT Networks using Keras Deep Learning Library". In: *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. Mar. 2019, pp. 633–637. DOI: 10.1109/ICCMC.2019.8819688.

[192] Du Shao-Bo. "Intrusion Feature Selection Method Based on Neighborhood Distance". In: *2017 International Conference on Computer Systems, Electronics and Control (ICCSEC)*. Dec. 2017, pp. 748–751. DOI: 10.1109/ICCSEC.2017.8446849.

[193] Saqr Mohammed Almansob and Santosh Shivajirao Lomte. "Addressing challenges for intrusion detection system using naive Bayes and PCA algorithm". In: *2017 2nd International Conference for Convergence in Technology (I2CT)*. Apr. 2017, pp. 565–568. DOI: 10.1109/I2CT.2017.8226193.

[194] Arijit Chandra, Sunil Kumar Khatri, and Rajbala Simon. "Filter-based Attribute Selection Approach for Intrusion Detection using k-Means Clustering and Sequential Minimal Optimization Techniq". In: *2019 Amity International Conference on Artificial Intelligence (AICAI)*. Feb. 2019, pp. 740–745. DOI: 10.1109/AICAI.2019.8701373.

[195] Yang Jia, Meng Wang, and Yagang Wang. "Network intrusion detection algorithm based on deep neural network". In: *IET Information Security* 13.1 (2019). Conference Name: IET Information Security, pp. 48–53. ISSN: 1751-8717. DOI: 10.1049/iet-ifs.2018.5258.

[196] Tahir Mehmood and Helmi B Md Rais. "SVM for network anomaly detection using ACO feature subset". In: *2015 International Symposium on Mathematical Sciences and Computing Research (iSMSC)*. May 2015, pp. 121–126. DOI: 10.1109/ISMSC.2015.7594039.

[197] Xuanqiang Zhao, Guoying Wang, and Zhixing Li. "Unsupervised network anomaly detection based on abnormality weights and subspace clustering". In: *2016 Sixth International Conference on Information Science and Technology (ICIST)*. May 2016, pp. 482–486. DOI: 10.1109/ICIST.2016.7483462.

[198] Ammar Alazab et al. "Using response action with intelligent intrusion detection and prevention system against web application malware". In: *Information Management & Computer Security* 22.5 (Jan. 2014). Publisher: Emerald Group Publishing Limited, pp. 431–449. ISSN: 0968-5227. DOI: 10.1108/IMCS-02-2013-0007. URL: https://doi.org/10.1108/IMCS-02-2013-0007 (visited on 05/14/2020).

[199] Solane Duque and Mohd. Nizam bin Omar. "Using Data Mining Algorithms for Developing a Model for Intrusion Detection System (IDS)". en. In: *Procedia Computer Science*. Complex Adaptive Systems San Jose, CA November 2-4, 2015 61 (Jan. 2015), pp. 46–51. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.09.145. URL: http://www.sciencedirect.com/science/article/pii/S1877050915029750 (visited on 05/14/2020).

[200] Fouzi Harrou et al. "A Method to Detect DOS and DDOS Attacks based on Generalized Likelihood Ratio Test". In: *2018 International Conference on Applied Smart Systems (ICASS)*. Nov. 2018, pp. 1–6. DOI: 10.1109/ICASS.2018.8652030.

[201] Fouzi Harrou et al. "Detecting cyber-attacks using a CRPS-based monitoring approach". In: *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. Nov. 2018, pp. 618–622. DOI: 10.1109/SSCI.2018.8628797.

[202] Benamar Bouyeddou et al. "An Effective Network Intrusion Detection Using Hellinger Distance-Based Monitoring Mechanism". In: *2018 International Conference on Applied Smart Systems (ICASS)*. Nov. 2018, pp. 1–6. DOI: 10.1109/ICASS.2018.8652008.

[203] Benamar Bouyeddou et al. “Detection of smurf flooding attacks using Kullback-Leibler-based scheme”. In: *2018 4th International Conference on Computer and Technology Applications (ICCTA)*. May 2018, pp. 11–15. DOI: 10.1109/CATA.2018.8398647.

[204] Guo Fan and Yu Min. “Automatic Attack Scenario Construction by Mining Meta-alert Sequences”. In: *2009 Second Pacific-Asia Conference on Web Mining and Web-based Application*. June 2009, pp. 149–153. DOI: 10.1109/WMWA.2009.13.

[205] Guo Fan, Ye JiHua, and Yu Min. “Design and implementation of a distributed IDS alert aggregation model”. In: *2009 4th International Conference on Computer Science Education*. July 2009, pp. 975–980. DOI: 10.1109/ICCSE.2009.5228172.

[206] Hamed Salehi, Hossein Shirazi, and Reza Askari Moghadam. “Increasing Overall Network Security by Integrating Signature-Based NIDS with Packet Filtering Firewall”. In: *2009 International Joint Conference on Artificial Intelligence*. Apr. 2009, pp. 357–362. DOI: 10.1109/JCAI.2009.12.

[207] Richard Lippmann et al. “The 1999 DARPA off-line intrusion detection evaluation”. In: *Comput. Netw.* 34.4 (Oct. 2000), pp. 579–595. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(00)00139-0. URL: https://doi.org/10.1016/S1389-1286(00)00139-0 (visited on 05/14/2020).

[208] Nour Moustafa and Jill Slay. “The Significant Features of the UNSW-NB15 and the KDD99 Data Sets for Network Intrusion Detection Systems”. In: *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. Nov. 2015, pp. 25–31. DOI: 10.1109/BADGERS.2015.014.

[209] Dishan Jing and Hai-Bao Chen. “SVM Based Network Intrusion Detection for the UNSW-NB15 Dataset”. In: *2019 IEEE 13th International Conference on ASIC (ASICON)*. ISSN: 2162-755X. Oct. 2019, pp. 1–4. DOI: 10.1109/ASICON47005.2019.8983598.

[210] Anwar Husain et al. “Development of an Efficient Network Intrusion Detection Model Using Extreme Gradient Boosting (XGBoost) on the UNSW-NB15 Dataset”. In: *2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. ISSN: 2641-5542. Dec. 2019, pp. 1–7. DOI: 10.1109/ISSPIT47144.2019.9001867.

[211] Liu Zhiqiang et al. “Modeling Network Intrusion Detection System Using Feed-Forward Neural Network Using UNSW-NB15 Dataset”. In: *2019 IEEE 7th International Conference on Smart Energy Grid Engineering (SEGE)*. ISSN: 2575-2693. Aug. 2019, pp. 299–303. DOI: 10.1109/SEGE.2019.8859773.

[212] Tharmini Janarthanan and Shahrzad Zargari. “Feature selection in UNSW-NB15 and KDDCUP’99 datasets”. In: *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*. ISSN: 2163-5145. June 2017, pp. 1881–1886. DOI: 10.1109/ISIE.2017.8001537.

[213] Abhishek Divekar et al. “Benchmarking datasets for Anomaly-based Network Intrusion Detection: KDD CUP 99 alternatives”. In: *2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS)*. Oct. 2018, pp. 1–8. DOI: 10.1109/CCCS.2018.8586840.

[214] Charles Wheelus, Elias Bou-Harb, and Xingquan Zhu. "Tackling Class Imbalance in Cyber Security Datasets". In: *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. July 2018, pp. 229–232. DOI: 10.1109/IRI.2018.00041.

[215] Malek Al-Zewairi, Sufyan Almajali, and Arafat Awajan. "Experimental Evaluation of a Multilayer Feed-Forward Artificial Neural Network Classifier for Network Intrusion Detection System". In: *2017 International Conference on New Trends in Computing Sciences (ICTCS)*. Oct. 2017, pp. 167–172. DOI: 10.1109/ICTCS.2017.29.

[216] Kamalakanta Sethi et al. "Deep Reinforcement Learning based Intrusion Detection System for Cloud Infrastructure". In: *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. ISSN: 2155-2509. Jan. 2020, pp. 1–6. DOI: 10.1109/COMSNETS48256.2020.9027452.

[217] Nour Moustafa et al. "Collaborative anomaly detection framework for handling big data of cloud computing". In: *2017 Military Communications and Information Systems Conference (MilCIS)*. Nov. 2017, pp. 1–6. DOI: 10.1109/MilCIS.2017.8190421.

[218] Sana Siddiqui, Muhammad Salman Khan, and Ken Ferens. "Multiscale Hebbian neural network for cyber threat detection". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. ISSN: 2161-4407. May 2017, pp. 1427–1434. DOI: 10.1109/IJCNN.2017.7966020.

[219] M. A. Mithun Aravind and V.K.G. Kalaiselvi. "Design of an intrusion detection system based on distance feature using ensemble classifier". In: *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*. Mar. 2017, pp. 1–6. DOI: 10.1109/ICSCN.2017.8085661.

[220] SU Yang. "Research on Network Behavior Anomaly Analysis Based on Bidirectional LSTM". In: *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. Mar. 2019, pp. 798–802. DOI: 10.1109/ITNEC.2019.8729475.

[221] Gcinizwe Dlamini, Rufina Galieva, and Muhammad Fahim. "A Lightweight Deep Autoencoder-Based Approach for Unsupervised Anomaly Detection". In: *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*. ISSN: 2161-5330. Nov. 2019, pp. 1–5. DOI: 10.1109/AICCSA47632.2019.9035217.

[222] Meliboev Azizjon, Alikhanov Jumabek, and Wooseong Kim. "1D CNN based network intrusion detection with normalization on imbalanced data". In: *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. Feb. 2020, pp. 218–224. DOI: 10.1109/ICAIIC48513.2020.9064976.

[223] Umair Ahmad et al. "Analysis of Classification Techniques for Intrusion Detection". In: *2019 International Conference on Innovative Computing (ICIC)*. Nov. 2019, pp. 1–6. DOI: 10.1109/ICIC48496.2019.8966675.

[224] Bayu Adhi Tama, Marco Comuzzi, and Kyung-Hyune Rhee. "TSE-IDS: A Two-Stage Classifier Ensemble for Intelligent Anomaly-Based Intrusion Detection System". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 94497–94507. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2928048.

[225] Olayemi Oladimeji Olasehinde, Olanrewaju Victor Johnson, and Olufunke Catherine Olayemi. "Evaluation Of Selected Meta Learning Algorithms For The Prediction Improvement Of Network Intrusion Detection System". In: *2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS)*. Mar. 2020, pp. 1–7. DOI: 10.1109/ICMCECS47690.2020.240893.

[226] Haitao He et al. "A Novel Multimodal-Sequential Approach Based on Multi-View Features for Network Intrusion Detection". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 183207–183221. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2959131.

[227] Muhammad Hilmi Kamarudin et al. "A LogitBoost-Based Algorithm for Detecting Known and Unknown Web Attacks". In: *IEEE Access* 5 (2017). Conference Name: IEEE Access, pp. 26190–26200. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2766844.

[228] Daniel Gibert et al. "Using convolutional neural networks for classification of malware represented as images". en. In: *J Comput Virol Hack Tech* 15.1 (Mar. 2019), pp. 15–28. ISSN: 2263-8733. DOI: 10.1007/s11416-018-0323-0. URL: https://doi.org/10.1007/s11416-018-0323-0 (visited on 03/21/2020).

[229] Vikash Raja Samuel Selvin. "Malware Scores Based on Image Processing". en. Master of Science. San Jose, CA, USA: San Jose State University, May 2017. DOI: 10.31979/etd.347x-pf32. URL: https://scholarworks.sjsu.edu/etd_projects/546 (visited on 05/04/2020).

[230] Aziz Makandar and Anita Patrot. "Malware class recognition using image processing techniques". In: *2017 International Conference on Data Management, Analytics and Innovation (ICDMAI)*. Feb. 2017, pp. 76–80. DOI: 10.1109/ICDMAI.2017.8073489.

[231] Aziz Makandar and Anita Patrot. "Detection and Retrieval of Malware Using Classification". In: *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*. Aug. 2017, pp. 1–5. DOI: 10.1109/ICCUBEA.2017.8463798.

[232] William R. Claycomb et al. "Identifying indicators of insider threats: Insider IT sabotage". In: *2013 47th International Carnahan Conference on Security Technology (ICCST)*. ISSN: 2153-0742. Oct. 2013, pp. 1–5. DOI: 10.1109/CCST.2013.6922038.

[233] Andrew P. Moore et al. "Balancing Organizational Incentives to Counter Insider Threat". In: *2018 IEEE Security and Privacy Workshops (SPW)*. May 2018, pp. 237–246. DOI: 10.1109/SPW.2018.00039.

[234] David A. Mundie, Sam Perl, and Carly L. Huth. "Toward an Ontology for Insider Threat Research: Varieties of Insider Threat Definitions". In: *2013 Third Workshop on Socio-Technical Aspects in Security and Trust*. ISSN: 2325-1697. June 2013, pp. 26–36. DOI: 10.1109/STAST.2013.14.

[235] Obinna Igbe and Tarek Saadawi. "Insider Threat Detection using an Artificial Immune system Algorithm". In: *2018 9th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. Nov. 2018, pp. 297–302. DOI: 10.1109/UEMCON.2018.8796583.

[236] Lori Flynn et al. "Best practices against insider threats for all nations". In: *2012 Third Worldwide Cybersecurity Summit (WCS)*. Oct. 2012, pp. 1–8. DOI: 10.1109/WCS.2012.6780874.

[237] William R Claycomb and Alex Nicoll. "Insider Threats to Cloud Computing: Directions for New Research Challenges". In: *2012 IEEE 36th Annual Computer Software and Applications Conference*. ISSN: 0730-3157. July 2012, pp. 387–394. DOI: 10.1109/COMPSAC.2012.113.

[238] Maryam Aldairi, Leila Karimi, and James Joshi. "A Trust Aware Unsupervised Learning Approach for Insider Threat Detection". In: *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*. July 2019, pp. 89–98. DOI: 10.1109/IRI.2019.00027.

[239] Liu Liu et al. "Insider Threat Identification Using the Simultaneous Neural Learning of Multi-Source Logs". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 183162–183176. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2957055.

[240] Fanzhi Meng et al. "Deep Learning Based Attribute Classification Insider Threat Detection for Data Security". In: *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. June 2018, pp. 576–581. DOI: 10.1109/DSC.2018.00092.

[241] Lingli Lin et al. "Insider Threat Detection Based on Deep Belief Network Feature Representation". In: *2017 International Conference on Green Informatics (ICGI)*. Aug. 2017, pp. 54–59. DOI: 10.1109/ICGI.2017.37.

[242] Ahmed Saaudi et al. "Insider Threats Detection Using CNN-LSTM Model". In: *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. Dec. 2018, pp. 94–99. DOI: 10.1109/CSCI46756.2018.00025.

[243] Jiarong Wang et al. "Embedding Learning with Heterogeneous Event Sequence for Insider Threat Detection". In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. ISSN: 2375-0197. Nov. 2019, pp. 947–954. DOI: 10.1109/ICTAI.2019.00134.

[244] Frank L. Greitzer et al. "Combating the Insider Cyber Threat". In: *IEEE Security Privacy* 6.1 (Jan. 2008). Conference Name: IEEE Security Privacy, pp. 61–64. ISSN: 1558-4046. DOI: 10.1109/MSP.2008.8.

[245] Philip A. Legg. "Visualizing the insider threat: challenges and tools for identifying malicious user activity". In: *2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*. Oct. 2015, pp. 1–7. DOI: 10.1109/VIZSEC.2015.7312772.

[246] Adam James Hall et al. "Predicting Malicious Insider Threat Scenarios Using Organizational Data and a Heterogeneous Stack-Classifier". In: *2018 IEEE International Conference on Big Data (Big Data)*. Dec. 2018, pp. 5034–5039. DOI: 10.1109/BigData.2018.8621922.

[247] Jianguo Jiang et al. "Warder: Online Insider Threat Detection System Using Multi-Feature Modeling and Graph-Based Correlation". In: *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*. ISSN: 2155-7586. Nov. 2019, pp. 1–6. DOI: 10.1109/MILCOM47813.2019.9020931.

[248] Aodi Liu, Xuehui Du, and Na Wang. "Recognition of Access Control Role Based on Convolutional Neural Network". In: *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. Dec. 2018, pp. 2069–2074. DOI: 10.1109/CompComm.2018.8780610.

[249] David Noever. "Classifier Suites for Insider Threat Detection". In: *arXiv:1901.10948 [cs, stat]* (Jan. 2019). arXiv: 1901.10948. URL: http://arxiv.org/abs/1901.10948 (visited on 05/13/2020).

[250] Chin-Wei Chen et al. "Malware Family Classification using Active Learning by Learning". In: *2020 22nd International Conference on Advanced Communication Technology (ICACT)*. ISSN: 1738-9445. Feb. 2020, pp. 590–595. DOI: 10.23919/ICACT48636.2020.9061419.

[251] Ahmad Karim, Rosli Salleh, and Muhammad Khurram Khan. "SMARTbot: A Behavioral Analysis Framework Augmented with Machine Learning to Identify Mobile Botnet Applications". In: *PLoS One* 11.3 (Mar. 2016). ISSN: 1932-6203. DOI: 10.1371/journal.pone.0150077. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4792466/ (visited on 07/19/2020).

[252] Xiao Zhang et al. "ACE – An Anomaly Contribution Explainer for Cyber-Security Applications". In: *2019 IEEE International Conference on Big Data (Big Data)*. Dec. 2019, pp. 1991–2000. DOI: 10.1109/BigData47090.2019.9005989.

[253] Paolo Buono and Pietro Carella. "Towards Secure Mobile Learning. Visual Discovery of Malware Patterns in Android Apps". In: *2019 23rd International Conference Information Visualisation (IV)*. ISSN: 2375-0138. July 2019, pp. 364–369. DOI: 10.1109/IV.2019.00068.

[254] Anastasia Skovoroda and Dennis Gamayunov. "Automated Static Analysis and Classification of Android Malware using Permission and API Calls Models". In: *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. Aug. 2017, pp. 243–24309. DOI: 10.1109/PST.2017.00036.

[255] Ambra Demontis et al. "Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks". In: *arXiv:1809.02861 [cs, stat]* (June 2019). arXiv: 1809.02861. URL: http://arxiv.org/abs/1809.02861 (visited on 10/12/2021).

[256] Royi Ronen et al. "Microsoft Malware Classification Challenge". In: *arXiv:1802.10135 [cs]* (Feb. 2018). arXiv: 1802.10135. URL: http://arxiv.org/abs/1802.10135 (visited on 03/20/2020).

[257] Sukriti Bhattacharya et al. "ITect: Scalable Information Theoretic Similarity for Malware Detection". In: *arXiv:1609.02404 [cs]* (Sept. 2016). arXiv: 1609.02404. URL: http://arxiv.org/abs/1609.02404 (visited on 01/04/2019).

[258] Jake Drew, Tyler Moore, and Michael Hahsler. "Polymorphic Malware Detection Using Sequence Classification Methods". In: *2016 IEEE Security and Privacy Workshops (SPW)*. May 2016, pp. 81–87. DOI: 10.1109/SPW.2016.30.

[259] Mansour Ahmadi et al. "Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification". In: *arXiv:1511.04317 [cs]* (Mar. 2016). arXiv: 1511.04317. URL: http://arxiv.org/abs/1511.04317 (visited on 12/22/2021).

[260] Daniel Gibert, Carles Mateu, and Jordi Planes. "A Hierarchical Convolutional Neural Network for Malware Classification". In: *2019 International Joint Conference on Neural Networks (IJCNN)*. ISSN: 2161-4407. July 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8852469.

[261] Mehadi Hassen and Philip K. Chan. "Scalable Function Call Graph-based Malware Classification". In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. New York, NY, USA: Association for Computing Machinery, Mar. 2017, pp. 239–248. ISBN: 978-1-4503-4523-1. DOI: 10.1145/3029806.3029824. URL: https://doi.org/10.1145/3029806.3029824 (visited on 10/12/2021).

[262] Daniel Gibert et al. "Classification of Malware by Using Structural Entropy on Convolutional Neural Networks". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 7759–7764. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16133 (visited on 03/20/2020).

[263] Daniel Gibert, Carles Mateu, and Jordi Planes. "A Hierarchical Convolutional Neural Network for Malware Classification". In: *2019 International Joint Conference on Neural Networks (IJCNN)*. ISSN: 2161-4393. July 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8852469.

[264] Temesguen Messay Kebede et al. "Classification of Malware programs using autoencoders based deep learning architecture and its application to the microsoft malware Classification challenge (BIG 2015) dataset". In: *2017 IEEE National Aerospace and Electronics Conference (NAECON)*. ISSN: 2379-2027. June 2017, pp. 70–75. DOI: 10.1109/NAECON.2017.8268747.

[265] Temesguen Messay-Kebede, Barath Narayanan Narayanan, and Ouboti Djaneye-Boundjou. "Combination of Traditional and Deep Learning based Architectures to Overcome Class Imbalance and its Application to Malware Classification". In: *NAECON 2018 - IEEE National Aerospace and Electronics Conference*. ISSN: 2379-2027. July 2018, pp. 73–77. DOI: 10.1109/NAECON.2018.8556722.

[266] Barath Narayanan Narayanan, Ouboti Djaneye-Boundjou, and Temesguen M. Kebede. "Performance analysis of machine learning and pattern recognition algorithms for Malware classification". In: *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*. ISSN: 2379-2027. July 2016, pp. 338–342. DOI: 10.1109/NAECON.2016.7856826.

[267] Mamta Kumari, George Hsieh, and Christopher A. Okonkwo. "Deep Learning Approach to Malware Multi-class Classification Using Image Processing Techniques". In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. Dec. 2017, pp. 13–18. DOI: 10.1109/CSCI.2017.3.

[268] Haidar Safa, Mohamed Nassar, and Wael Al Rahal Al Orabi. "Benchmarking Convolutional and Recurrent Neural Networks for Malware Classification". In: *2019 15th International Wireless Communications Mobile Computing Conference (IWCMC)*. ISSN: 2376-6506. June 2019, pp. 561–566. DOI: 10.1109/IWCMC.2019.8766515.

[269] Venkata Salini Priyamvada Davuluru, Barath Narayanan Narayanan, and Eric J. Balster. "Convolutional Neural Networks as Classification Tools and Feature Extractors for Distinguishing Malware Programs". In: *2019 IEEE National Aerospace and Electronics Conference (NAECON)*. ISSN: 2379-2027. July 2019, pp. 273–278. DOI: 10.1109/NAECON46414.2019.9058025.

[270] Evgeny Burnaev and Dmitry Smolyakov. "One-Class SVM with Privileged Information and Its Application to Malware Detection". In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. ISSN: 2375-9259. Dec. 2016, pp. 273–280. DOI: 10.1109/ICDMW.2016.0046.

[271] Chang Hoon Kim, Espoir K. Kabanga, and Sin-Jae Kang. "Classifying malware using convolutional gated neural network". In: *2018 20th International Conference on Advanced Communication Technology (ICACT)*. Feb. 2018, pp. 40–44. DOI: 10.23919/ICACT.2018.8323640.

[272] Mouhammd Alkasassbeh and Samail Al-Daleen. *Classification of malware based on file content and characteristics*. en. Project. 2018, p. 12.

[273] S. Abijah Roseline and S Geetha. "An Efficient Malware Detection System using Hybrid Feature Selection Methods". en. In: *IJEAT* 9.1S3 (Dec. 2019), pp. 224–228. ISSN: 2249-8958. DOI: 10.35940/ijeat.A1043.1291S319. URL: https://www.ijeat.org/wp-content/uploads/papers/v9i1s3/A10431291S319.pdf (visited on 05/04/2020).

[274] Emmanuel Gbenga Dada et al. "Performance Evaluation of Machine Learning Algorithms for Detection and Prevention of Malware Attacks". en. In: *IOSR-Journal of Computer Engineering* 21.3 (June 2019), pp. 18–27.

[275] Subhojeet Pramanik and Hemanth Teja. "EMBER - Analysis of Malware Dataset Using Convolutional Neural Networks". In: *2019 Third International Conference on Inventive Systems and Control (ICISC)*. Jan. 2019, pp. 286–291. DOI: 10.1109/ICISC44355.2019.9036424.

[276] Yoshihiro Oyama, Takumi Miyashita, and Hirotaka Kokubo. "Identifying Useful Features for Malware Detection in the Ember Dataset". In: *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. Nov. 2019, pp. 360–366. DOI: 10.1109/CANDARW.2019.00069.

[277] Andre T. Nguyen, Edward Raff, and Aaron Sant-Miller. "Would a File by Any Other Name Seem as Malicious?" In: *2019 IEEE International Conference on Big Data (Big Data)*. Dec. 2019, pp. 1322–1331. DOI: 10.1109/BigData47090.2019.9006132.

[278] Chris Eagle. *The IDA Pro Book, 2nd Edition: The Unofficial Guide to the World's Most Popular Disassembler*. en. Google-Books-ID: 3nPAM3AZ1foC. No Starch Press, 2011. ISBN: 978-1-59327-289-0.

[279] Peyman Khodamoradi et al. "Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms". In: *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*. Oct. 2015, pp. 1–6. DOI: 10.1109/CADS.2015.7377792.

[280] Necmettin Çarkacı and İbrahim Soğukpınar. "Frequency based metamorphic malware detection". In: *2016 24th Signal Processing and Communication Application Conference (SIU)*. May 2016, pp. 421–424. DOI: 10.1109/SIU.2016.7495767.

[281] Sanjay K. Sahay and Ashu Sharma. "Grouping the Executables to Detect Malwares with High Accuracy". en. In: *Procedia Computer Science*. 1st International Conference on Information Security & Privacy 2015 78 (Jan. 2016), pp. 667–674. ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.02.115. URL: http://www.sciencedirect.com/science/article/pii/S1877050916001174 (visited on 02/07/2020).

[282] Arzu Gorgulu Kakisim et al. “Analysis and Evaluation of Dynamic Feature-Based Malware Detection Methods”. en. In: *Innovative Security Solutions for Information Technology and Communications: 11th International Conference, SecITC 2018, Bucharest, Romania, November 8–9, 2018, Revised Selected Papers*. Google-Books-ID: IciGDwAAQBAJ. Springer, Feb. 2019. ISBN: 978-3-030-12942-2.

[283] Bhatnagar Vishal. *Data Mining and Analysis in the Engineering Field*. en. Google-Books-ID: 0RWXBQAAQBAJ. IGI Global, May 2014. ISBN: 978-1-4666-6087-8.

[284] Maleh Yassine. *Security and Privacy Management, Techniques, and Protocols*. en. Google-Books-ID: OD1RDwAAQBAJ. IGI Global, Apr. 2018. ISBN: 978-1-5225-5584-1.

[285] Alireza Khalilian et al. “G3MD: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families”. en. In: *Expert Systems with Applications* 112 (Dec. 2018), pp. 15–33. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2018.06.012. URL: http://www.sciencedirect.com/science/article/pii/S0957417418303580 (visited on 02/05/2020).

[286] Wing Wong. “Analysis and Detection of Metamorphic Computer Viruses”. en. Master of Science. San Jose, CA, USA: San Jose State University, Jan. 2006. DOI: 10.31979/etd.rnm3-sdfc. URL: https://scholarworks.sjsu.edu/etd_projects/153 (visited on 05/18/2020).

[287] Vinod P. Nair et al. “MEDUSA: MEtamorphic malware dynamic analysis usingsignature from API”. en. In: *Proceedings of the 3rd international conference on Security of information and networks - SIN '10*. Taganrog, Rostov-on-Don, Russian Federation: ACM Press, 2010, p. 263. ISBN: 978-1-4503-0234-0. DOI: 10.1145/1854099.1854152. URL: http://portal.acm.org/citation.cfm?doid=1854099.1854152 (visited on 05/18/2020).

[288] Mohamad Fadli Zolkipli and Aman Jantan. “A Framework for Malware Detection Using Combination Technique and Signature Generation”. In: *2010 Second International Conference on Computer Research and Development*. May 2010, pp. 196–199. DOI: 10.1109/ICCRD.2010.25.

[289] Shiv Kumar Agarwal and Vishal Shrivastava. *An Opcode Statistical Analysis for Metamorphic Malware*. en. Library Catalog: www.semanticscholar.org. 2013. URL: https://www.semanticscholar.org/paper/An-Opcode-Statistical-Analysis-for-Metamorphic-Agarwal-Shrivastava/2c5f5c82b4d814ec1327e97404e79945bde7a354 (visited on 05/12/2020).

[290] Manoj Shelar D. and Srinivasa Rao. “Malicious Threats Detection of Executable File”. en. In: *IJITEE* 9.3 (Jan. 2020), pp. 3257–3262. ISSN: 2278-3075. DOI: 10.35940/ijitee.C8918.019320. URL: http://www.ijitee.org/wp-content/uploads/papers/v9i3/C8918019320.pdf (visited on 05/12/2020).

[291] Essam Al Daoud. “Metamorphic Viruses Detection Using Artificial Immune System”. In: *2009 International Conference on Communication Software and Networks*. Feb. 2009, pp. 168–172. DOI: 10.1109/ICCSN.2009.145.

[292] Radhouane Chouchane et al. “Detecting machine-morphed malware variants via engine attribution”. en. In: *J Comput Virol Hack Tech* 9.3 (Aug. 2013), pp. 137–157. ISSN: 2263-8733. DOI: 10.1007/s11416-013-0183-6. URL: https://doi.org/10.1007/s11416-013-0183-6 (visited on 05/12/2020).

[293] Ankur Singh Bist and Anuj Sharma. "Analysis of Computer Virus Using Feature Fusion". In: *2016 Second International Conference on Computational Intelligence Communication Technology (CICT)*. Feb. 2016, pp. 609–614. DOI: 10.1109/CICT.2016.127.

[294] Reza Mirzazadeh, Mohammad Hossein Moattar, and Majid Vafaei Jahan. "Metamorphic malware detection using Linear Discriminant Analysis and Graph Similarity". In: *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*. Oct. 2015, pp. 61–66. DOI: 10.1109/ICCKE.2015.7365862.

[295] Jithu Raphel and Vinod P. "Pruned feature space for metamorphic malware detection using Markov Blanket". In: *2015 Eighth International Conference on Contemporary Computing (IC3)*. Aug. 2015, pp. 377–382. DOI: 10.1109/IC3.2015.7346710.

[296] Jikku Kuriakose and P. Vinod. "Ranked linear discriminant analysis features for metamorphic malware detection". In: *2014 IEEE International Advance Computing Conference (IACC)*. Feb. 2014, pp. 112–117. DOI: 10.1109/IAdCC.2014.6779304.

[297] Ashwini Venkatesan. "Code Obfuscation and Virus Detection". en. Master of Science. San Jose, CA, USA: San Jose State University, Jan. 2008. DOI: 10.31979/etd.ez5v-x8jc. URL: https://scholarworks.sjsu.edu/etd_projects/116 (visited on 05/12/2020).

[298] Mark Stamp. *Introduction to Machine Learning with Applications in Information Security*. en. Google-Books-ID: CuM2DwAAQBAJ. CRC Press, Sept. 2017. ISBN: 978-1-351-81806-3.

[299] Srilatha Attaluri, Scott McGhee, and Mark Stamp. "Profile hidden Markov models and metamorphic virus detection". en. In: *J Comput Virol* 5.2 (May 2009), pp. 151–169. ISSN: 1772-9904. DOI: 10.1007/s11416-008-0105-1. URL: https://doi.org/10.1007/s11416-008-0105-1 (visited on 05/15/2020).

[300] Neha Runwal, Richard M. Low, and Mark Stamp. "Opcode graph similarity and metamorphic detection". en. In: *J Comput Virol* 8.1 (May 2012), pp. 37–52. ISSN: 1772-9904. DOI: 10.1007/s11416-012-0160-5. URL: https://doi.org/10.1007/s11416-012-0160-5 (visited on 01/07/2020).

[301] Michael A Davis, Sean Bodmer, and Aaron LeMasters. *Hacking exposed malware & rootkits: malware & rootkits security secrets & solutions*. en. OCLC: 465403255. New York: McGraw Hill, 2010. ISBN: 978-0-07-159119-5. URL: http://site.ebrary.com/id/10343415 (visited on 08/06/2021).

[302] Abhishek Singh and Ali Islam. *An Encounter with Trojan Nap*. en. Feb. 2013. URL: https://www.fireeye.com/blog/threat-research/2013/02/an-encounter-with-trojan-nap.html (visited on 12/05/2020).

[303] Abhishek Singh and Yasir Khalid. *Don't Click the Left Mouse Button: Introducing Trojan UpClicker*. en. Dec. 2012. URL: https://www.fireeye.com/blog/threat-research/2012/12/dont-click-the-left-mouse-button-trojan-upclicker.html (visited on 12/05/2020).

[304] Boldizsár Bencsáth et al. *Duqu: A Stuxnet-like malware found in the wild*. Technical Report. Budapest: Budapest University of Technology and Economics, 2011. URL: https://www.crysys.hu/publications/files/bencsathPBF11duqu.pdf (visited on 07/18/2020).

[305] Vitaly Kamluk. *The Mystery of Duqu: Part Six (The Command and Control servers)*. APT Reports. kaspersky, 2011. URL: https://securelist.com/the-mystery-of-duqu-part-six-the-command-and-control-servers-36/31863/ (visited on 07/18/2020).

[306] Abhishek Singh and Zheng Bu. *Hot knives Through Butter: Evading File-based Sandboxes*. en. White Paper. Milpitas, CA: Fireye, 2014, p. 27.

[307] Tal Garfinkel et al. "Compatibility is not transparency: VMM detection myths and realities". In: *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. HOTOS'07. USA: USENIX Association, May 2007, pp. 1–6. (Visited on 08/06/2021).

[308] Parvez Faruki et al. "Evaluation of Android Anti-malware Techniques against Dalvik Bytecode Obfuscation". In: *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. ISSN: 2324-9013. Sept. 2014, pp. 414–421. DOI: 10.1109/TrustCom.2014.54.

[309] Dhilung Kirat and Giovanni Vigna. "MalGene: Automatic Extraction of Malware Analysis Evasion Signature". In: *CCS '15*. 2015. DOI: 10.1145/2810103.2813642.

[310] Thanasis Petsas et al. "Rage against the virtual machine: hindering dynamic analysis of Android malware". In: *EuroSec '14*. 2014. DOI: 10.1145/2592791.2592796.

[311] Parvez Faruki et al. "DroidAnalyst: Synergic App Framework for Static and Dynamic App Analysis". en. In: *Recent Advances in Computational Intelligence in Defense and Security*. Ed. by Rami Abielmona et al. Studies in Computational Intelligence. Cham: Springer International Publishing, 2016, pp. 519–552. ISBN: 978-3-319-26450-9. DOI: 10.1007/978-3-319-26450-9_20. URL: https://doi.org/10.1007/978-3-319-26450-9_20 (visited on 01/08/2020).

[312] androguard. *androguard/androguard*. original-date: 2014-09-12T08:48:56Z. July 2020. URL: https://github.com/androguard/androguard (visited on 07/19/2020).

[313] Patrik Lantz. *pjlantz/droidbox*. original-date: 2014-08-25T12:11:18Z. July 2020. URL: https://github.com/pjlantz/droidbox (visited on 07/19/2020).

[314] Martina Lindorfer et al. "ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors". In: *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. Sept. 2014, pp. 3–17. DOI: 10.1109/BADGERS.2014.7.

[315] Seymour Bosworth and M. E. Kabay. *Computer Security Handbook*. en. Google-Books-ID: rCx5OfSFUPkC. John Wiley & Sons, Oct. 2002. ISBN: 978-0-471-26975-5.

[316] Vikas Sihag, Manu Vardhan, and Pradeep Singh. "BLADE: Robust malware detection against obfuscation in android". en. In: *Forensic Science International: Digital Investigation* 38 (Sept. 2021), p. 301176. ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2021.301176. URL: https://www.sciencedirect.com/science/article/pii/S2666281721000846 (visited on 04/27/2023).

[317] Hoda El Merabet and Abderrahmane Hajraoui. "A Survey of Malware Detection Techniques based on Machine Learning". In: *International Journal of Advanced Computer Science and Applications* 10 (Jan. 2019). DOI: 10.14569/IJACSA.2019.0100148.

[318] Alireza Souri and Rahil Hosseini. "A state-of-the-art survey of malware detection approaches using data mining techniques". In: *Human-centric Computing and Information Sciences* 8.1 (Jan. 2018), p. 3. ISSN: 2192-1962. DOI: 10.1186/s13673-018-0125-x. URL: https://doi.org/10.1186/s13673-018-0125-x (visited on 12/22/2021).

[319] N. Idika and A. Mathur. "A Survey of Malware Detection Techniques". en. In: *undefined* (2007). URL: https://www.semanticscholar.org/paper/A-Survey-of-Malware-Detection-Techniques-Idika-Mathur/888cb0ac77e16ec98c8e2a5af79e567d8a43dcd1 (visited on 12/22/2021).

[320] Ömer Aslan and Refik Samet. "A Comprehensive Review on Malware Detection Approaches". In: *IEEE Access* (2020). DOI: 10.1109/ACCESS.2019.2963724.

[321] E.M. Rudd et al. "A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions". English. In: *IEEE Communications Surveys and Tutorials* 19.2 (2017), pp. 1145–1172. ISSN: 1553-877X. DOI: 10.1109/COMST.2016.2636078.

[322] Daniel Gibert, Carles Mateu, and Jordi Planes. "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges". en. In: *Journal of Network and Computer Applications* 153 (Mar. 2020), p. 102526. ISSN: 10848045. DOI: 10.1016/j.jnca.2019.102526. URL: https://linkinghub.elsevier.com/retrieve/pii/S1084804519303868 (visited on 05/02/2020).

[323] David M. Chess and Steve R. White. "Un undectable computer virus". In: *Proceedings of Virus Bulletin*. Vol. 5. 2000. URL: https://cryptohub.nl/zines/vxheavens/lib/adc06.html (visited on 06/04/2020).

[324] Masoud Narouei et al. "DLLMiner: structural mining for malware detection". en. In: *Security and Communication Networks* 8.18 (2015), pp. 3311–3322. ISSN: 1939-0122. DOI: 10.1002/sec.1255. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1255 (visited on 01/08/2020).

[325] Matthew G. Schultz et al. "Data mining methods for detection of new malicious executables". In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001* (2001). DOI: 10.1109/SECPRI.2001.924286.

[326] S. P. Choudhary and Miss Deepti Vidyarthi. "A Simple Method for Detection of Metamorphic Malware using Dynamic Analysis and Text Mining". en. In: *Procedia Computer Science*. Eleventh International Conference on Communication Networks, ICCN 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Data Mining and Warehousing, ICDMW 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Image and Signal Processing, ICISP 2015, August 21-23, 2015, Bangalore, India 54 (Jan. 2015), pp. 265–270. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.06.031. URL: https://www.sciencedirect.com/science/article/pii/S1877050915013551 (visited on 12/20/2021).

[327] Hisham Shehata Galal, Yousef Bassyouni Mahdy, and Mohammed Ali Atiea. "Behavior-based features model for malware detection". en. In: *J Comput Virol Hack Tech* 12.2 (May 2016), pp. 59–67. ISSN: 2263-8733. DOI: 10.1007/s11416-015-0244-0. URL: https://doi.org/10.1007/s11416-015-0244-0 (visited on 03/20/2020).

[328] Zahra Salehi, Ashkan Sami, and Mahboobe Ghiasi. "MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values". en. In: *Engineering Applications of Artificial Intelligence* 59 (Mar. 2017), pp. 93–102. ISSN: 0952-1976. DOI: 10.1016/j.engappai.2016.12.016. URL: http://www.sciencedirect.com/science/article/pii/S0952197616302512 (visited on 03/20/2020).

[329] Paolo Milani Comparetti et al. "Identifying Dormant Functionality in Malware Programs". In: *2010 IEEE Symposium on Security and Privacy*. ISSN: 2375-1207. May 2010, pp. 61–76. DOI: 10.1109/SP.2010.12.

[330] Ahmad Firdaus et al. "Bio-inspired computational paradigm for feature investigation and malware detection: interactive analytics". en. In: *Multimed Tools Appl* 77.14 (July 2018), pp. 17519–17555. ISSN: 1573-7721. DOI: 10.1007/s11042-017-4586-0. URL: https://doi.org/10.1007/s11042-017-4586-0 (visited on 12/20/2021).

[331] Arzu Gorgulu Kakisim, Mert Nar, and Ibrahim Sogukpinar. "Metamorphic malware identification using engine-specific patterns based on co-opcode graphs". en. In: *Computer Standards & Interfaces* 71 (Aug. 2020), p. 103443. ISSN: 0920-5489. DOI: 10.1016/j.csi.2020.103443. URL: https://www.sciencedirect.com/science/article/pii/S0920548919302685 (visited on 02/01/2022).

[332] Usukhbayar Baldangombo, Nyamjav Jambaljav, and Shi-Jinn Horng. "A Static Malware Detection System Using Data Mining Methods". In: *arXiv:1308.2831 [cs]* (Aug. 2013). arXiv: 1308.2831. URL: http://arxiv.org/abs/1308.2831 (visited on 12/20/2021).

[333] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. en. Google-Books-ID: FQC8EPYy834C. No Starch Press, Feb. 2012. ISBN: 978-1-59327-430-6.

[334] Muhammad Ejaz Ahmed, Surya Nepal, and Hyoungshick Kim. "MEDUSA: Malware Detection Using Statistical Analysis of System's Behavior". In: *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. Oct. 2018, pp. 272–278. DOI: 10.1109/CIC.2018.00044.

[335] Jithin Pavithran, Milan Patnaik, and Chester Rebeiro. "{D-TIME}: Distributed Threadless Independent Malware Execution for Runtime Obfuscation". en. In: 2019. URL: https://www.usenix.org/conference/woot19/presentation/pavithran (visited on 03/28/2022).

[336] Shay Zamir, Yanki Margalit, and Dany Margalit. "Method for detecting unwanted executables". US20060015940A1. Jan. 2006. URL: https://patents.google.com/patent/US20060015940/en (visited on 04/13/2022).

[337] Red Teaming Experiments. *Persisting in svchost.exe with a Service DLL*. 2022. URL: https://www.ired.team/offensive-security/persistence/persisting-in-svchost.exe-with-a-service-dll-servicemain (visited on 03/28/2022).

[338] Mamoun Alazab, Sitalakshmi Venkataraman, and Paul Watters. "Towards Understanding Malware Behaviour by the Extraction of API Calls". In: *2010 Second Cybercrime and Trustworthy Computing Workshop*. ISSN: null. July 2010, pp. 52–59. DOI: 10.1109/CTC.2010.8.

[339] Şerif Bahtiyar, Mehmet Barış Yaman, and Can Yılmaz Altınığne. "A multi-dimensional machine learning approach to predict advanced malware". en. In: *Computer Networks* 160 (Sept. 2019), pp. 118–129. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2019.06.015. URL: http://www.sciencedirect.com/science/article/pii/S138912861831082X (visited on 02/12/2020).

[340] Yuxin Ding et al. "A malware detection method based on family behavior graph". en. In: *Computers & Security* 73 (Mar. 2018), pp. 73–86. ISSN: 0167-4048. DOI: 10.1016/j.cose.2017.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0167404817302146 (visited on 12/20/2021).

[341] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur. "Malware Characterization Using Windows API Call Sequences". en. In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Claude Carlet, M. Anwar Hasan, and Vishal Saraswat. Cham: Springer International Publishing, 2016, pp. 271–280. ISBN: 978-3-319-49445-6. DOI: 10.1007/978-3-319-49445-6_15.

[342] Sanchit Gupta et al. "Malware Characterization Using WindowsAPI Call Sequences". en. In: *JCSM* 7.4 (2018), pp. 363–378. ISSN: 2245-1439. DOI: 10.13052/jcsm2245-1439.741. URL: http://www.riverpublishers.com/journal_read_html_article.php?j=JCSM/7/4/1 (visited on 04/18/2020).

[343] Mohamed Belaoued and Smaine Mazouzi. "Statistical Study of Imported APIs by PE Type Malware". In: *2014 International Conference on Advanced Networking Distributed Systems and Applications*. June 2014, pp. 82–86. DOI: 10.1109/INDS.2014.22.

[344] P. Vinod et al. *MEDUSA: MEtamorphic malware dynamic analysis usingsignature from API*. Journal Abbreviation: SIN'10 - Proceedings of the 3rd International Conference of Security of Information and Networks Pages: 269 Publication Title: SIN'10 - Proceedings of the 3rd International Conference of Security of Information and Networks. Jan. 2010. DOI: 10.1145/1854099.1854152.

[345] Aleksandr Matrosov et al. *Stuxnet Under the Microscope*. en. Tech. rep. Revision 1.1. ESET LLC, 2010, p. 72. URL: https://www.esetnod32.ru/company/viruslab/analytics/doc/Stuxnet_Under_the_Microscope.pdf.

[346] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. "Intrusion detection using sequences of system calls". In: *J. Comput. Secur.* 6.3 (Aug. 1998), pp. 151–180. ISSN: 0926-227X.

[347] Yanfang Ye et al. "An intelligent PE-malware detection system based on association mining". en. In: *J Comput Virol* 4.4 (Nov. 2008), pp. 323–334. ISSN: 1772-9904. DOI: 10.1007/s11416-008-0082-4. URL: https://doi.org/10.1007/s11416-008-0082-4 (visited on 01/08/2020).

[348] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. "A Novel Approach to Detect Malware Based on API Call Sequence Analysis". en. In: *International Journal of Distributed Sensor Networks* 11.6 (June 2015). Publisher: SAGE Publications, p. 659101. ISSN: 1550-1329. DOI: 10.1155/2015/659101. URL: https://doi.org/10.1155/2015/659101 (visited on 03/29/2022).

[349] Aya Hellal and Lotfi Ben Romdhane. "Minimal contrast frequent pattern mining for malware detection". en. In: *Computers & Security* 62 (Sept. 2016), pp. 19–32. ISSN: 0167-4048. DOI: 10.1016/j.cose.2016.06.004. URL: http://www.sciencedirect.com/science/article/pii/S0167404816300694 (visited on 01/08/2020).

[350] Weijie Han et al. "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics". en. In: *Computers & Security* 83 (June 2019), pp. 208–233. ISSN: 0167-4048. DOI: 10.1016/j.cose.2019.02.007. URL: http://www.sciencedirect.com/science/article/pii/S016740481831246X (visited on 03/21/2020).

[351] Dolly Uppal et al. "Malware detection and classification based on extraction of API sequences". In: *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. ISSN: null. Sept. 2014, pp. 2337–2342. DOI: 10.1109/ICACCI.2014.6968547.

[352] Ulrich Bayer et al. "Scalable, Behavior-Based Malware Clustering". en. In: (2009), p. 18.

[353] Abdurrahman Pektaş and Tankut Acarman. "Classification of malware families based on runtime behaviors". en. In: *Journal of Information Security and Applications* 37 (Dec. 2017), pp. 91–100. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2017.10.005. URL: http://www.sciencedirect.com/science/article/pii/S2214212617301643 (visited on 03/21/2020).

[354] Matilda Rhode et al. "LAB to SOC: Robust Features for Dynamic Malware Detection". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Industry Track*. ISSN: null. June 2019, pp. 13–16. DOI: 10.1109/DSN-Industry.2019.00010.

[355] Igor Santos et al. "OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection". en. In: *International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions*. Ed. by Álvaro Herrero et al. Advances in Intelligent Systems and Computing. Berlin, Heidelberg: Springer, 2013, pp. 271–280. ISBN: 978-3-642-33018-6. DOI: 10.1007/978-3-642-33018-6_28.

[356] P.V. Shijo and A. Salim. "Integrated Static and Dynamic Analysis for Malware Detection". In: *Procedia Computer Science* 46 (Dec. 2015), pp. 804–811. DOI: 10.1016/j.procs.2015.02.149.

[357] Kevadia Kaushal, Prashant Swadas, and Nilesh Prajapati. "Metamorphic Malware Detection Using Statistical Analysis". en. In: 2.3 (2012), p. 5.

[358] Madhu K. Shankarapani et al. "Malware detection using assembly and API call sequences". en. In: *J Comput Virol* 7.2 (May 2011), pp. 107–119. ISSN: 1772-9904. DOI: 10.1007/s11416-010-0141-5. URL: https://doi.org/10.1007/s11416-010-0141-5 (visited on 12/20/2021).

[359] Faraz Ahmed et al. "Using spatio-temporal information in API calls with machine learning algorithms for malware detection". In: *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*. AISec '09. New York, NY, USA: Association for Computing Machinery, Nov. 2009, pp. 55–62. ISBN: 978-1-60558-781-3. DOI: 10.1145/1654988.1655003. URL: https://doi.org/10.1145/1654988.1655003 (visited on 12/20/2021).

[360] Cheng Wang et al. "Malware Detection Based on Suspicious Behavior Identification". In: *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science - Volume 02*. ETCS '09. USA: IEEE Computer Society, Mar. 2009, pp. 198–202. ISBN: 978-0-7695-3557-9. DOI: 10.1109/ETCS.2009.306. URL: https://doi.org/10.1109/ETCS.2009.306 (visited on 02/12/2020).

[361] Ashkan Sami et al. "Malware detection based on mining API calls". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, Mar. 2010, pp. 1020–1025. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774303. URL: https://doi.org/10.1145/1774088.1774303 (visited on 03/19/2020).

[362] Yi-Dong Shen, Zhong Zhang, and Qiang Yang. "Objective-oriented utility-based association mining". In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* Dec. 2002, pp. 426–433. DOI: 10.1109/ICDM.2002.1183938.

[363] Yanfang Ye et al. "IMDS: intelligent malware detection system". In: *KDD '07*. 2007. DOI: 10.1145/1281192.1281308.

[364] Yanfang Ye et al. "An intelligent PE-malware detection system based on association mining". en. In: *J Comput Virol* 4.4 (Nov. 2008), pp. 323–334. ISSN: 1772-9904. DOI: 10.1007/s11416-008-0082-4. URL: https://doi.org/10.1007/s11416-008-0082-4 (visited on 03/20/2020).

[365] Yanfang Ye et al. "Hierarchical associative classifier (HAC) for malware detection from the large and imbalanced gray list". en. In: *J Intell Inf Syst* 35.1 (Aug. 2010), pp. 1–20. ISSN: 1573-7675. DOI: 10.1007/s10844-009-0086-7. URL: https://doi.org/10.1007/s10844-009-0086-7 (visited on 01/08/2020).

[366] Arkan A. G. Al-Hamodi, Song Lu, and Y. Alsalhi. "AN ENHANCED FREQUENT PATTERN GROWTH BASED ON MAP REDUCE FOR MINING ASSOCIATION RULES". In: (2016). DOI: 10.5121/IJDKP.2016.6202.

[367] Yuxin Ding et al. "A fast malware detection algorithm based on objective-oriented association mining". en. In: *Computers & Security* 39 (Nov. 2013), pp. 315–324. ISSN: 0167-4048. DOI: 10.1016/j.cose.2013.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0167404813001259 (visited on 03/20/2020).

[368] Anna L. Buczak and Erhan Guven. "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection". In: *IEEE Communications Surveys Tutorials* 18.2 (2016). Conference Name: IEEE Communications Surveys Tutorials, pp. 1153–1176. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2494502.

[369] G.J. Tesauro, J.O. Kephart, and G.B. Sorkin. "Neural networks for computer virus recognition". In: *IEEE Expert* 11.4 (Aug. 1996). Conference Name: IEEE Expert, pp. 5–6. ISSN: 2374-9407. DOI: 10.1109/64.511768.

[370] Andrew Walenstein, Daniel J. Hefner, and Jeffery Wichers. "Header information in malware families and impact on automated classifiers". In: *2010 5th International Conference on Malicious and Unwanted Software*. Oct. 2010, pp. 15–22. DOI: 10.1109/MALWARE.2010.5665799.

[371] J. Zico Kolter and Marcus A. Maloof. "Learning to Detect and Classify Malicious Executables in the Wild". In: *J. Mach. Learn. Res.* 7 (Dec. 2006), pp. 2721–2744. ISSN: 1532-4435.

[372] Chandrasekar Ravi and R. Manoharan. "Malware Detection using Windows Api Sequence and Machine Learning". In: (2012). DOI: 10.5120/6194-8715.

[373] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. "Detecting Self-mutating Malware Using Control-Flow Graph Matching". en. In: *Detection of Intrusions and Malware & Vulnerability Assessment*. Ed. by Roland Büschkes and Pavel Laskov. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 129–143. ISBN: 978-3-540-36017-9. DOI: 10.1007/11790754_8.

[374] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. "Control Flow Graphs as Malware Signatures". en. In: (May 2007), p. 7.

[375] Kyoochang Jeong and Heejo Lee. "Code Graph for Malware Detection". In: *2008 International Conference on Information Networking*. ISSN: 2332-5658. Jan. 2008, pp. 1–5. DOI: 10.1109/ICOIN.2008.4472801.

[376] Mojtaba Eskandari and Hooman Raesi. "Frequent sub-graph mining for intelligent malware detection". en. In: *Security and Communication Networks* 7.11 (2014), pp. 1872–1886. ISSN: 1939-0122. DOI: 10.1002/sec.902. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.902 (visited on 02/07/2020).

[377] Parvez Faruki et al. "Mining control flow graph as API call-grams to detect portable executable malware". In: *Proceedings of the Fifth International Conference on Security of Information and Networks*. SIN '12. Jaipur, India: Association for Computing Machinery, Oct. 2012, pp. 130–137. ISBN: 978-1-4503-1668-2. DOI: 10.1145/2388576.2388594. URL: https://doi.org/10.1145/2388576.2388594 (visited on 03/19/2020).

[378] Kristina Blokhin, Josh Saxe, and David Mentis. "Malware Similarity Identification Using Call Graph Based System Call Subsequence Features". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. ISSN: 2332-5666. July 2013, pp. 6–10. DOI: 10.1109/ICDCSW.2013.55.

[379] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. "Mining specifications of malicious behavior". In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, Sept. 2007, pp. 5–14. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287628. URL: https://doi.org/10.1145/1287624.1287628 (visited on 12/20/2021).

[380] Roger Ming Hieng Ting and J. Bailey. "Mining Minimal Contrast Subgraph Patterns". In: *SDM*. 2006. DOI: 10.1137/1.9781611972764.76.

[381] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. "Malware detection with quantitative data flow graphs". In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ASIA CCS '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 271–282. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590319. URL: https://doi.org/10.1145/2590296.2590319 (visited on 12/20/2021).

[382] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. "Detecting metamorphic malwares using code graphs". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, Mar. 2010, pp. 1970–1977. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774505. URL: https://doi.org/10.1145/1774088.1774505 (visited on 03/15/2020).

[383] Vishakha Mehra, Vinesh Jain, and Dolly Uppal. "DaCoMM: Detection and Classification of Metamorphic Malware". In: *2015 Fifth International Conference on Communication Systems and Network Technologies*. Apr. 2015, pp. 668–673. DOI: 10.1109/CSNT.2015.62.

[384] Ahmed Abusnaina et al. "Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. ISSN: 2575-8411. July 2019, pp. 1296–1305. DOI: 10.1109/ICDCS.2019.00130.

[385] K.V. Vinayaka and C.D. Jaidhar. "Android Malware Detection using Function Call Graph with Graph Convolutional Networks". In: *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*. May 2021, pp. 279–287. DOI: 10.1109/ICSCCC51823.2021.9478141.

[386] Pengbin Feng et al. "Android Malware Detection Based on Call Graph via Graph Neural Network". In: *2020 International Conference on Networking and Network Applications (NaNA)*. Dec. 2020, pp. 368–374. DOI: 10.1109/NaNA51271.2020.00069.

[387] Rahim Taheri et al. "Similarity-based Android malware detection using Hamming distance of static binary features". en. In: *Future Generation Computer Systems* 105 (Apr. 2020), pp. 230–247. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.11.034. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19315122 (visited on 02/01/2022).

[388] Mojtaba Eskandari and Sattar Hashemi. "A graph mining approach for detecting unknown malwares". en. In: *Journal of Visual Languages & Computing* 23.3 (June 2012), pp. 154–162. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2012.02.002. URL: http://www.sciencedirect.com/science/article/pii/S1045926X12000146 (visited on 01/08/2020).

[389] ElMouatez Billah Karbab and Mourad Debbabi. "MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports". en. In: *Digital Investigation* 28 (Apr. 2019), S77–S87. ISSN: 1742-2876. DOI: 10.1016/j.diin.2019.01.017. URL: https://www.sciencedirect.com/science/article/pii/S1742287619300271 (visited on 12/20/2021).

[390] Ferhat Ozgur Catak et al. "Deep learning based Sequential model for malware analysis using Windows exe API Calls". en. In: *PeerJ Comput. Sci.* 6 (July 2020). Publisher: PeerJ Inc., e285. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.285. URL: https://peerj.com/articles/cs-285 (visited on 01/25/2022).

[391] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. arXiv:1301.3781 [cs]. Sept. 2013. DOI: 10.48550/arXiv.1301.3781. URL: http://arxiv.org/abs/1301.3781 (visited on 02/07/2023).

[392] Eslam Amer and Ivan Zelinka. "A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence". en. In: *Computers & Security* 92 (May 2020), p. 101760. ISSN: 0167-4048. DOI: 10.1016/j.cose.2020.101760. URL: https://www.sciencedirect.com/science/article/pii/S0167404820300444 (visited on 12/20/2021).

[393] Eslam Amer, Shaker El-Sappagh, and Jong Wan Hu. "Contextual Identification of Windows Malware through Semantic Interpretation of API Call Sequence". en. In: *Applied Sciences* 10.21 (Oct. 2020), p. 7673. ISSN: 2076-3417. DOI: 10.3390/app10217673. URL: https://www.mdpi.com/2076-3417/10/21/7673 (visited on 04/08/2022).

[394] Jungho Kang et al. "Long short-term memory-based Malware classification method for information security". en. In: *Computers & Electrical Engineering* 77 (July 2019), pp. 366–375. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2019.06.014. URL: https://www.sciencedirect.com/science/article/pii/S0045790618328167 (visited on 12/20/2021).

[395] Ferhat Ozgur Catak and Ahmet Faruk Yazı. "A Benchmark API Call Dataset for Windows PE Malware Classification". In: *arXiv:1905.01999 [cs]* (Feb. 2021). arXiv: 1905.01999. URL: http://arxiv.org/abs/1905.01999 (visited on 04/20/2022).

[396] Ahmet Faruk Yazi, Ferhat Özgür Çatak, and Ensar Gül. "Classification of Methamorphic Malware with Deep Learning(LSTM)". In: *2019 27th Signal Processing and Communications Applications Conference (SIU)*. ISSN: 2165-0608. Apr. 2019, pp. 1–4. DOI: 10.1109/SIU.2019.8806571.

[397] Serena McDonnell et al. "CyberBERT: A Deep Dynamic-State Session-Based Recommender System for Cyber Threat Recognition". In: *2021 IEEE Aerospace Conference (50100)*. ISSN: 1095-323X. Mar. 2021, pp. 1–12. DOI: 10.1109/AERO50100.2021.9438286.

[398] Salih Yesir and İbrahim Soğukpinar. "Malware Detection and Classification Using fastText and BERT". In: *2021 9th International Symposium on Digital Forensics and Security (ISDFS)*. June 2021, pp. 1–6. DOI: 10.1109/ISDFS52919.2021.9486377.

[399] Armand Joulin et al. "FastText.zip: Compressing text classification models". In: *arXiv:1612.03651 [cs]* (Dec. 2016). arXiv: 1612.03651. URL: http://arxiv.org/abs/1612.03651 (visited on 04/29/2022).

[400] Alexey Strokach et al. "Fast and Flexible Protein Design Using Deep Graph Neural Networks". en. In: *Cell Systems* 11.4 (Oct. 2020), 402–411.e4. ISSN: 2405-4712. DOI: 10.1016/j.cels.2020.08.016. URL: https://www.sciencedirect.com/science/article/pii/S2405471220303276 (visited on 04/22/2022).

[401] Jiawei Zhou et al. "Automating Botnet Detection with Graph Neural Networks". In: *arXiv:2003.06344 [cs, stat]* (Mar. 2020). arXiv: 2003.06344. URL: http://arxiv.org/abs/2003.06344 (visited on 04/22/2022).

[402] Chunyan Diao et al. "A Novel Spatial-Temporal Multi-Scale Alignment Graph Neural Network Security Model for Vehicles Prediction". In: *IEEE Transactions on Intelligent Transportation Systems* (2022). Conference Name: IEEE Transactions on Intelligent Transportation Systems, pp. 1–11. ISSN: 1558-0016. DOI: 10.1109/TITS.2022.3140229.

[403] M. Gori, G. Monfardini, and F. Scarselli. "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. ISSN: 2161-4407. July 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

[404] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1 (Jan. 2009). Conference Name: IEEE Transactions on Neural Networks, pp. 61–80. ISSN: 1941-0093. DOI: 10.1109/TNN.2008.2005605.

[405] David Duvenaud et al. "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *arXiv:1509.09292 [cs, stat]* (Nov. 2015). arXiv: 1509.09292. URL: http://arxiv.org/abs/1509.09292 (visited on 02/08/2022).

[406] Yakang Hua, Yuanzheng Du, and Dongzhi He. "Classifying Packed Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network". In: *2020 International Conference on Computer Engineering and Application (ICCEA)*. Mar. 2020, pp. 254–258. DOI: 10.1109/ICCEA50009.2020.00062.

[407] 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)*. ISSN: 1530-0889. June 2019, pp. 52–63. DOI: 10.1109/DSN.2019.00020.

[408] Kay Liu et al. *BOND: Benchmarking Unsupervised Outlier Node Detection on Static Attributed Graphs*. arXiv:2206.10071 [cs]. Oct. 2022. DOI: 10.48550/arXiv.2206.10071. URL: http://arxiv.org/abs/2206.10071 (visited on 12/05/2022).

[409] M. Sakurada and T. Yairi. "Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction". In: *MLSDA'14*. 2014. DOI: 10.1145/2689746.2689747.

[410] Sambaran Bandyopadhyay, Lokesh N, and M. N. Murty. *Outlier Aware Network Embedding for Attributed Networks*. en. arXiv:1811.07609 [cs]. Nov. 2018. URL: http://arxiv.org/abs/1811.07609 (visited on 11/07/2022).

[411] Xuexiong Luo et al. "ComGA: Community-Aware Attributed Graph Anomaly Detection". In: *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. WSDM '22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 657–665. ISBN: 978-1-4503-9132-0. DOI: 10.1145/3488560.3498389. URL: https://doi.org/10.1145/3488560.3498389 (visited on 04/28/2022).

[412] S. Bandyopadhyay et al. "Outlier Resistant Unsupervised Deep Architectures for Attributed Network Embedding". In: *WSDM* (2020). DOI: 10.1145/3336191.3371788.

[413] Kaize Ding et al. "Deep Anomaly Detection on Attributed Networks". In: *Proceedings of the 2019 SIAM International Conference on Data Mining (SDM)*. Proceedings. Society for Industrial and Applied Mathematics, May 2019, pp. 594–602. DOI: 10.1137/1.9781611975673.67. URL: https://epubs.siam.org/doi/10.1137/1.9781611975673.67 (visited on 11/22/2022).

[414] Haoyi Fan, Fengbin Zhang, and Zuoyong Li. "Anomalydae: Dual Autoencoder for Anomaly Detection on Attributed Networks". In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ISSN: 2379-190X. May 2020, pp. 5685–5689. DOI: 10.1109/ICASSP40776.2020.9053387.

[415] Xu Yuan et al. “Higher-order Structure Based Anomaly Detection on Attributed Networks”. In: *2021 IEEE International Conference on Big Data (Big Data)*. Dec. 2021, pp. 2691–2700. DOI: 10.1109/BigData52589.2021.9671990.

[416] Kaize Ding et al. “Inductive Anomaly Detection on Attributed Networks”. en. In: vol. 2. ISSN: 1045-0823. July 2020, pp. 1288–1294. DOI: 10.24963/ijcai.2020/179. URL: https://www.ijcai.org/proceedings/2020/179 (visited on 04/28/2022).

[417] Lei Cai et al. “Structural Temporal Graph Neural Networks for Anomaly Detection in Dynamic Graphs”. In: *arXiv:2005.07427 [cs, stat]* (May 2020). arXiv: 2005.07427. URL: http://arxiv.org/abs/2005.07427 (visited on 04/12/2022).

[418] Xiaoxiao Ma et al. “A Comprehensive Survey on Graph Anomaly Detection with Deep Learning”. In: *IEEE Trans. Knowl. Data Eng.* (2021). arXiv: 2106.07178, pp. 1–1. ISSN: 1041-4347, 1558-2191, 2326-3865. DOI: 10.1109/TKDE.2021.3118815. URL: http://arxiv.org/abs/2106.07178 (visited on 04/28/2022).

[419] Rex Ying et al. *GNNExplainer: Generating Explanations for Graph Neural Networks*. arXiv:1903.03894 [cs, stat]. Nov. 2019. DOI: 10.48550/arXiv.1903.03894. URL: http://arxiv.org/abs/1903.03894 (visited on 11/22/2022).

[420] Qiang Huang et al. *GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks*. arXiv:2001.06216 [cs, stat]. Sept. 2020. DOI: 10.48550/arXiv.2001.06216. URL: http://arxiv.org/abs/2001.06216 (visited on 11/27/2022).

[421] Tetsu Kasanishi, Xueting Wang, and Toshihiko Yamasaki. “Edge-Level Explanations for Graph Neural Networks by Extending Explainability Methods for Convolutional Neural Networks”. In: *2021 IEEE International Symposium on Multimedia (ISM)*. Nov. 2021, pp. 249–252. DOI: 10.1109/ISM52913.2021.00049.

[422] Masaru Todoriki et al. “Semi-Automatic Reliable Explanations for Prediction in Graphs”. In: *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. Jan. 2021, pp. 0311–0320. DOI: 10.1109/CCWC51732.2021.9375922.

[423] Scott Lundberg and Su-In Lee. *A Unified Approach to Interpreting Model Predictions*. arXiv:1705.07874 [cs, stat]. Nov. 2017. DOI: 10.48550/arXiv.1705.07874. URL: http://arxiv.org/abs/1705.07874 (visited on 12/22/2022).

[424] John Sutcliffe et al. “Harold Edwin Hurst: the Nile and Egypt, past and future”. In: *Hydrological Sciences Journal* 61.9 (July 2016). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/02626667.2015.1 pp. 1557–1570. ISSN: 0262-6667. DOI: 10.1080/02626667.2015.1019508. URL: https://doi.org/10.1080/02626667.2015.1019508 (visited on 01/17/2023).

[425] Jongsuk R. Lee, Sang-Kug Ye, and Hae-Duck J. Jeong. “Detecting Anomaly Teletraffic Using Stochastic Self-Similarity Based on Hadoop”. In: *2013 16th International Conference on Network-Based Information Systems*. ISSN: 2157-0426. Sept. 2013, pp. 282–287. DOI: 10.1109/NBiS.2013.43.

[426] Aaron D. Likens et al. “Better than DFA? A Bayesian Method for Estimating the Hurst Exponent in Behavioral Sciences”. eng. In: *ArXiv* (Jan. 2023), arXiv:2301.11262v1. ISSN: 2331-8422.

[427] S. J. Yu et al. “Hurst Parameter Based Anomaly Detection for Intrusion Detection System”. In: *2016 IEEE International Conference on Computer and Information Technology (CIT)*. Dec. 2016, pp. 234–240. DOI: 10.1109/CIT.2016.98.

[428] W.E. Leland et al. “On the self-similar nature of Ethernet traffic (extended version)”. In: *IEEE/ACM Transactions on Networking* 2.1 (Feb. 1994). Conference Name: IEEE/ACM Transactions on Networking, pp. 1–15. ISSN: 1558-2566. DOI: 10.1109/90.282603.

[429] Ernande F. Melo and H. M. de Oliveira. *An Overview of Self-Similar Traffic: Its Implications in the Network Design*. arXiv:2005.02858 [cs]. May 2020. DOI: 10.48550/arXiv.2005.02858. URL: http://arxiv.org/abs/2005.02858 (visited on 01/17/2023).

[430] Hui He et al. *Early warning of active worms based on multi-similarity*. Journal Abbreviation: Gastroenterology Pages: 3883 Vol. 6 Publication Title: Gastroenterology. Sept. 2005. ISBN: 978-0-7803-9091-1. DOI: 10.1109/ICMLC.2005.1527616.

[431] Xunyi Ren, Ruchuan Wang, and Haiyan Wang. “Wavelet analysis method for detection of DDoS attack based on self-similar”. In: *Frontiers of Electrical and Electronic Engineering in China* 2 (Jan. 2007), pp. 73–77. DOI: 10.1007/s11460-007-0013-z.

[432] Liang Fu Lu et al. “An Improved Wavelet Analysis Method for Detecting DDoS Attacks”. In: *2010 Fourth International Conference on Network and System Security*. Sept. 2010, pp. 318–322. DOI: 10.1109/NSS.2010.23.

[433] Hae-Duck J. Jeong et al. “Anomalous Traffic Detection and Self-Similarity Analysis in the Environment of ATMSim”. en. In: *Cryptography* 1.3 (Dec. 2017). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, p. 24. ISSN: 2410-387X. DOI: 10.3390/cryptography1030024. URL: https://www.mdpi.com/2410-387X/1/3/24 (visited on 01/17/2023).

[434] Jordan Frecon et al. “Non-linear regression for bivariate self-similarity identification — application to anomaly detection in Internet traffic based on a joint scaling analysis of packet and byte counts”. In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ISSN: 2379-190X. Mar. 2016, pp. 4184–4188. DOI: 10.1109/ICASSP.2016.7472465.

[435] Fahad Mira, Antony Brown, and Wei Huang. “Novel malware detection methods by using LCS and LCSS”. In: *2016 22nd International Conference on Automation and Computing (ICAC)*. Sept. 2016, pp. 554–559. DOI: 10.1109/IConAC.2016.7604978.

[436] F. Mira, W. Huang, and A. Brown. “Improving malware detection time by using RLE and N-gram”. In: *2017 23rd International Conference on Automation and Computing (ICAC)*. Sept. 2017, pp. 1–5. DOI: 10.23919/IConAC.2017.8082001.

[437] Yunan Zhang et al. “Using Multi-features and Ensemble Learning Method for Imbalanced Malware Classification”. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. ISSN: 2324-9013. Aug. 2016, pp. 965–973. DOI: 10.1109/TrustCom.2016.0163.

[438] Kenneth Brezinski and Ken Ferens. “Incorporating Topological Complexity into a Multilayer Perceptron”. In: *Transactions on Computational Science & Computational Intelligence*. Advances in Artificial Intelligence and Applied Cognitive Computing. Mar. 2022.

[439] Wentao Chang et al. “Characterizing botnets-as-a-service”. In: *Proceedings of the 2014 ACM conference on SIGCOMM*. SIGCOMM ’14. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 585–586. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2631464. URL: https://doi.org/10.1145/2619239.2631464 (visited on 02/19/2021).

[440] Jeffrey Pennington, Richard Socher, and Christopher Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162 (visited on 02/07/2023).

[441] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. July 2020. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165 (visited on 02/07/2023).

[442] Alec Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018. URL: https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5(visited on 02/07/2023).

[443] Kai Wang et al. “Relational Graph Attention Network for Aspect-based Sentiment Analysis”. In: *arXiv:2004.12362 [cs]* (Apr. 2020). arXiv: 2004.12362. URL: http://arxiv.org/abs/2004.12362 (visited on 11/23/2021).

[444] Xuan Guo and Jianlong Wang. “Modeling of the fractal-like adsorption systems based on the diffusion limited aggregation model”. en. In: *Journal of Molecular Liquids* 324 (Feb. 2021), p. 114692. ISSN: 0167-7322. DOI: 10.1016/j.molliq.2020.114692. URL: https://www.sciencedirect.com/science/article/pii/S0167732220369348 (visited on 08/16/2021).

[445] Ryan A. Rossi and Nesreen K. Ahmed. “The network data repository with interactive graph analytics and visualization”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI’15. Austin, Texas: AAAI Press, Jan. 2015, pp. 4292–4293. ISBN: 978-0-262-51129-2. (Visited on 12/27/2022).

[446] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. *Multi-scale Attributed Node Embedding*. arXiv:1909.13021 [cs, stat]. Mar. 2021. DOI: 10.48550/arXiv.1909.13021. URL: http://arxiv.org/abs/1909.13021 (visited on 12/27/2022).

[447] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (visited on 11/23/2021).

[448] Kaiming He et al. *Deep Residual Learning for Image Recognition*. arXiv:1512.03385 [cs]. Dec. 2015. DOI: 10.48550/arXiv.1512.03385. URL: http://arxiv.org/abs/1512.03385 (visited on 12/16/2022).

[449] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. arXiv:1602.04938 [cs, stat]. Aug. 2016. DOI: 10.48550/arXiv.1602.04938. URL: http://arxiv.org/abs/1602.04938 (visited on 01/02/2023).

[450] Hwanjun Song et al. *How does Early Stopping Help Generalization against Label Noise?* arXiv:1911.08059 [cs, stat]. Sept. 2020. DOI: 10.48550/arXiv.1911.08059. URL: http://arxiv.org/abs/1911.08059 (visited on 01/31/2023).

[451] Rich Caruana, Steve Lawrence, and C. Giles. "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping". In: *Advances in Neural Information Processing Systems*. Vol. 13. MIT Press, 2000. URL: https://proceedings.neurips.cc/paper/2000/hash/059fdcd96baeb75112f09fa1dcc740cc-Abstract.html (visited on 01/31/2023).

[452] Xue-Wen Chen and Xiaotong Lin. "Big Data Deep Learning: Challenges and Perspectives". In: *IEEE Access* 2 (2014). Conference Name: IEEE Access, pp. 514–525. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2325029.

[453] Douglas Heaven. "Why deep-learning AIs are so easy to fool". en. In: *Nature* 574.7777 (Oct. 2019). Bandiera_abtest: a Cg_type: News Feature Number: 7777 Publisher: Nature Publishing Group Subject_term: Computer science, Information technology, pp. 163–166. DOI: 10.1038/d41586-019-03013-5. URL: https://www.nature.com/articles/d41586-019-03013-5 (visited on 02/02/2023).

[454] John D. Kelleher. *Deep Learning*. en. Google-Books-ID: b06qDwAAQBAJ. MIT Press, Sept. 2019. ISBN: 978-0-262-53755-1.

[455] Jun-Mo Jo. "Effectiveness of Normalization Pre-Processing of Big Data to the Machine Learning Performance". eng. In: *The Journal of the Korea institute of electronic communication sciences* 14.3 (2019). Publisher: Korea Institute of Electronic Communication Science, pp. 547–552. ISSN: 1975-8170. DOI: 10.13067/JKIECS.2019.14.3.547. URL: https://koreascience.kr/article/JAKO201924763903550.page (visited on 01/09/2023).

[456] Pedro Ferreira, Duc C. Le, and Nur Zincir-Heywood. "Exploring Feature Normalization and Temporal Information for Machine Learning Based Insider Threat Detection". In: *2019 15th International Conference on Network and Service Management (CNSM)*. ISSN: 2165-963X. Oct. 2019, pp. 1–7. DOI: 10.23919/CNSM46954.2019.9012708.

[457] Kübra Yıldırım et al. "A YARA-based approach for detecting cyber security attack types". In: *FIRAT UNIVERSITY JOURNAL OF EXPERIMENTAL AND COMPUTATIONAL ENGINEERING* 2 (Jan. 2023), pp. 55–68. DOI: 10.5505/fujece.2023.09709.

[458] Qin Si et al. "Malware Detection Using Automated Generation of Yara Rules on Dynamic Features". en. In: *Science of Cyber Security*. Ed. by Chunhua Su, Kouichi Sakurai, and Feng Liu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 315–330. ISBN: 978-3-031-17551-0. DOI: 10.1007/978-3-031-17551-0_21.

[459] Mohamed Belaoued et al. "Combined dynamic multi-feature and rule-based behavior for accurate malware detection". In: *International Journal of Distributed Sensor Networks* 15 (Nov. 2019). DOI: 10.1177/1550147719889907.

[460] Kenneth Brezinski. *Evaluating the Complexity and Robustness of Speech Utterances using Length and Variance Fractal Dimensions*. Course Project. Department of Electrical and Computer Engineering, University of Manitoba, Winnipeg, 2019.

[461] Bhavna Soman et al. *Firenze: Model Evaluation Using Weak Signals*. arXiv:2207.00827 [cs]. July 2022. DOI: 10.48550/arXiv.2207.00827. URL: http://arxiv.org/abs/2207.00827 (visited on 01/03/2024).

[462] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *AISTATS*. 2010.

[463] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (visited on 06/28/2019).

[464] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Jan. 2017). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 12/01/2021).

[465] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *arXiv:1711.05101 [cs, math]* (Jan. 2019). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101 (visited on 12/01/2021).