12-10-2021

Detecting malware in memory with memory object relationships

DeMarcus M. Thomas Sr.
Mississippi State University, demarcus.thomas601@gmail.com

Detecting malware in memory with memory object relationships

By

DeMarcus M. Thomas Sr.

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2021

Name: DeMarcus M. Thomas Sr.

Date of Degree: December 10, 2021

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: John A. Hamilton Jr.

Title of Study: Detecting malware in memory with memory object relationships

Pages of Study: 145

Candidate for Degree of Doctor of Philosophy

Malware is a growing concern that not only affects large businesses but the basic consumer as well. As a result, there is a need to develop tools that can identify the malicious activities of malware authors. A useful technique to achieve this is memory forensics. Memory forensics is the study of volatile data and its structures in Random Access Memory (RAM). It can be utilized to pinpoint what actions have occurred on a computer system.

This dissertation utilizes memory forensics to extract relationships between objects and supervised machine learning as a novel method for identifying malicious processes in a system memory dump. In this work, the Object Association Extractor (OAE) was created to extract objects in a memory dump and label the relationships as a graph of nodes and edges. With OAE, we extracted processes from 13,882 memory images that contained malware from the repository VirusShare and 91 memory images created with benign software from the package management software Chocolatey. The final dataset contained 267,824 processes.

Two feature sets were created from the processes dataset and used to train classifiers based on four classification algorithms. These classifiers were evaluated against the ZeroR method using accuracy and recall as the evaluation metrics. The experiments showed that both sets of features used to build classifiers were able to beat the ZeroR method for the Decision Tree and Random Forest algorithms. The Random Forest classifier achieved the highest performance by reaching a recall score of almost 97%.

Key words: malware, memory forensics, Rekall, community detection, machine learning

DEDICATION

To my Lord and Savior Jesus Christ, I thank you for your grace in walking this journey. There were many instances where I questioned if this process was worth it. However, over time I realized that I was placed in this season and role for a reason, and it was my job to be faithful during this time.

To Allie, I thank God for you each day. There is no way that I could have reached this milestone without your love, support, and encouragement. In my final days before the defense, you allowed me to put my full attention into the work and have my most profound adoration for your sacrifice. Thank you, My Love, for all that you do.

To my three sons, you drive me in so much of what I do. Each day I strive to be a father that you can be proud of. I want you to know that you can accomplish anything you desire by the grace of God and hard work. Junior and Crew thank you so much for the questions each day of, "Daddy, did you make some progress today?" I always wanted to give you a good report when I came home.

I appreciate the laughs, advice, and support my boy Chris Lanclos has provided. We have known each other for over a decade and have become brothers through this process. I look forward to you crossing this finish line also.

To my immediate family, I wouldn’t be who I am today with you. Terris, thanks for paving the way as my big brother. It is not easy being the first to blaze a trail, and I appreciate your loving example. TaMiah, thanks for helping me to talk through issues over the years. You have been a

valuable confidant, and I thank you for that. Titus, I also wanted to work to be a positive example for you. Thanks for pushing me to be better.

To my parents, thank you both for your selfless attitudes. You always made things about supporting those around you and strove to provide the best for your kids. You planted the seeds and created an environment for us all to grow.

To the people named in this section and the countless others that have shaped me over the years, this accomplishment is for you.

DMT

ACKNOWLEDGEMENTS

Many have played a role in my academic journey, and I would like to acknowledge a portion of them now. First, to Dr. Constance Bland, you gave me an opportunity when I barely had an idea of who I was. Thank you for taking a chance on me and pushing me where needed. You are truly missed.

To Dr. Ray Vaughn, thanks for giving me the spark that cybersecurity was a possible career path. To Dr. Edward Allen, thank you for recruiting an unproven kid from a small university. Dr. David Dampier, your mentoring and advice were invaluable to my academic development. You gave me opportunities at the FTC and DASI, and I am much better for it.

To my defacto fifth and sixth committee members, Dae and Puntitra "Tan" Glendowne, I have no way to repay the number of hours we spent in discussion. Thank you!

Finally, to my major professor Dr. John Hamilton, I can’t thank you enough for the wisdom you have contributed over the years and the autonomy you provided that allowed me to grow in my various roles at CCI. I did not always know the correct path to take, but you would patiently allow me to figure things out or provide the necessary nudge. Your leadership is greatly appreciated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Malware is a growing concern that not only affects large businesses but the basic consumer as well. Malicious activities are incentivized, and actors develop methods to profit from design, software, and user errors. Because of these constantly growing threats, there is always a need for effective solutions for identifying and mitigating risks. Malware researchers and incident responders have taken the task of joining the arms race against malicious attackers. They have employed a number of techniques including static [1] [2] [3] and dynamic analysis [4],[5] and have utilized automated systems to discern the intent of malware in a controlled environment [6],[7]. The struggle between malware authors and the security community is constantly changing. Each new defensive or investigative discovery prompts attackers to find new methods to subvert systems. Techniques are needed to make the job of concealing activities more difficult for malware authors. Memory forensics exploits a fundamental flaw malware authors face when attempting to conceal their behavior.

1.1 Memory Forensics

Memory forensics is the study of volatile data and its structures from a memory dump [8]. This technique provides insight into what actions have taken place on a system using physical memory. Walters and Petroni Jr. [9], [10], presented the first open-source framework used to collect information from physical memory on Windows systems. The authors also discussed integrating the collection of memory into the digital forensics process with minimum intrusiveness. Since its inception from string and grep analysis, there has been extensive work performed in understanding the components of physical memory [11],[12],[13]. In the early stages of the field, professionals saw these techniques as a luxury when hard disk forensics did not yield results. As the field has matured, it has become a core technique for discovering and understanding what has occurred on a computer system. Memory forensics has become an effective technique for digital forensics investigations that focus on cybercrime, incident response, and malware analysis [14],[10]. It gives context for what user accounts were active on a system, offers information on the actions taken, allows for the extraction of passwords, and can be used to extract specific files from a memory dump [15],[16]. These capabilities have drawn much interest in the research community and industry. This has led to a demand for training to learn and apply these techniques. [8], [17], [18]. In addition to the information described above, an analyst can also determine active and hidden processes, open files, information on network activities, and potentially find malware that is active on a system [19]. There are several tools available to perform investigations and research, but two main open-source tools have gained strong support: Volatility [20], and Rekall [21]. Volatility is a command-line tool with plugins that allow for the inspection of data structures in memory. Rekall branched from Volatility and has much of the same functionality but focuses on using the tool as a library. When this research was started, Volatility focused on collecting information from data structures in memory and saving that data to output files. Rekall provides an interface to collect data from its plugins and interact with data programmatically. In this work, Rekall is the tool of choice because of the programmatic functionality. It is used to automate the data extraction process.

1.2 Malware in Memory

Malware often aims to perform its tasks while also remaining concealed as long as possible. For malware to achieve its desired actions, it must exist in memory, which makes it more difficult to stay hidden. This has been called the rootkit paradox [22]. Malware can manifest itself in many ways, from a dropper that spawns other processes and ensures persistence through registry keys to samples that use hooking to perform keylogging to loading malicious kernel drivers with complete access over a system. With each of these manifestations, the presence of malware in some way affects some other components in memory. The internals of random access memory (RAM) on a system consists of many components that rely on one another to function correctly. Malware authors have studied how these components interact with one another and have developed methods to exploit the intended design of systems. This research aims to find a new way of representing the layout of objects in system memory. Specific to this work, objects in memory reference data structures used by the Windows operating system to perform tasks. Relationships between objects refer to the notion that a given object contains, creates, deletes, and can modify another object. With these definitions, this work views the problem of representing the layout of a system memory image as a graph of object relationships. Each object will be displayed as a node, and the relationships between the nodes are shown as edges. As with any graph, whether it be the study of molecules in chemistry, computer networks, and social networks, the more connected nodes are within a graph, the deeper the relationships with one another. Value is seen in this approach because it is used to identify relationships that were difficult to discover using manual analysis. The focus of this work is using memory forensics to determine the relationships of objects within memory images.

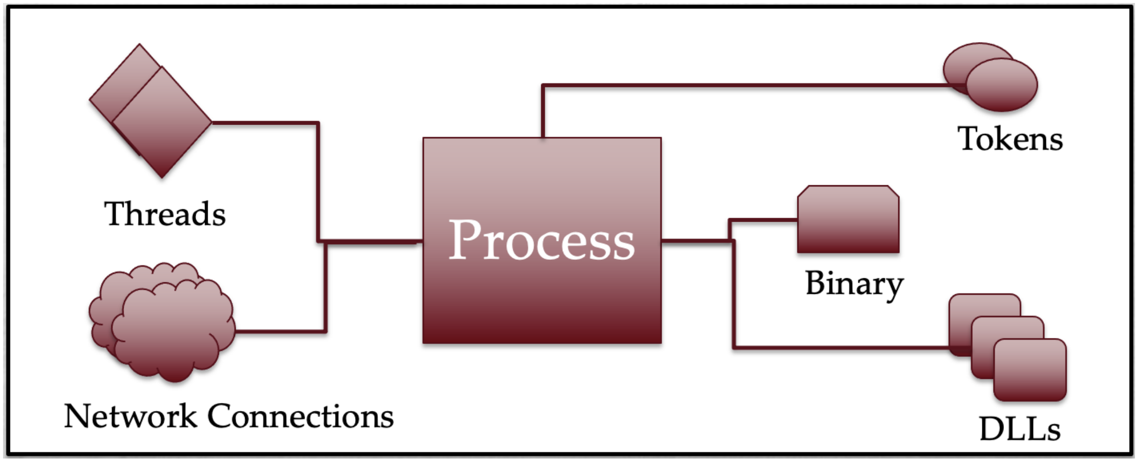


Figure 1.1 Process Object Relationships

Figure 1.1 shows a process that is started from a portable executable (PE) file. Once started, the address space for a process will contain dynamic-link libraries (DLLs) and tokens. DLLs are portable PE files used on Windows systems to provide access to common functions. If a DLL provides a given function, a process will import that DLL into its address space to access it. Tokens are objects that describe the accounts, groups, and privileges that are associated with a process. Tokens determine the types of resources that a process can access. In Figure 1.1 the process also creates threads, network connections, etc. A thread is a basic unit of a process that can be allocated processor time. Each process must have at least one thread that is executing. A network connection is established to communicate with other systems via a communication channel. Objects in memory have high connectivity to other objects and use these relationships to function properly. This research aims to prove that understanding these relationships and knowing how objects should be connected can effectively identify malicious activity.

1.3 Malicious Path Data

An assumption for malware is that if it has multiple components, it will have mechanisms that allow for synchronization and communication. This aspect of the research focuses on leveraging the path data of each memory object graph to determine if there are uncommon paths in a graph that malware uses. Malware authors can utilize different methods for performing malicious actions on an information system. In these experiments, empirical data is used to study the most common paths between malicious processes versus the most uncommon paths. Our intuition is that paths that are less common between objects could be informative for identifying malware in memory. To help identify relationships of interest, the information retrieval technique term frequency-inverse document frequency (TF-IDF) will be used. Hubballi, Biswas, and Nandi [23] used this method in previous work to identify combinations of opcode strings and API calls to identify malicious activity.

1.4 Machine Learning

Once the grouping of related memory objects has been collected from memory images, the characteristics of the graphs and the objects within the graphs can be extracted. These characteristics will then be used to train machine learning classifiers and determine if the connections of objects in memory can be used to find malicious processes. In the machine learning workflow, we ensure that standard techniques for building supervised machine learning models are followed: splitting data into training, validation, and test sets, ensuring the datasets are balanced based on class, and only using the test set for the final predictions.

1.5 Hypothesis

The hypothesis for this research is: *Memory object graph topology attributes and object characteristics from operating system memory images are a predictive factor for identifying processes containing malicious code.* To establish this hypothesis, a model will be trained and will show how accurately malicious or benign processes can be classified. In addition, the work will aim to maximize the metric recall. Recall measures how well a classifier can determine the positive class, in our case, identifying malicious processes.

As a baseline for comparison, the Zero Rule classifier will be used [24]. The Zero Rule classifier counts the number of instances of each class value, either malicious or benign, and determines the most frequent class value. All data points are then predicted to be this majority class. A successful classifier should be able to outperform the Zero Rule method. The research is currently structured around the Windows 7 operating system as a proof of concept. Once evidence to support our hypothesis is found, additional operating systems can be explored to apply our method to other contexts. There is also an extensive amount of malware available which targets this operating system. This availability of malware helps in creating a robust dataset for testing. In the future, a similar study will be conducted on Windows 10. The success of this research would logically lead to generalizing the demonstrated techniques to other operating systems. Operating systems may have similar functionality, but the underlying structures in memory can be different. Because of these differences, malware exhibits different behaviors based on the implementation details in the target OS. Therefore, if the research methodology work is proven for Windows 7, similar work taking account for specific OS implementation is required for other operating systems. Therefore, while we believe this research will be widely applicable to other operating systems, we stress that comparable work with different operating systems and operating system versions must be tested independently of these results.

1.6 Research Questions

Based on the hypothesis above, research questions were developed to steer the direction of this work.

1.6.1 Which memory objects are related to processes in random access memory?

This work focuses on studying the characteristics of processes, the paths between processes in a graph, and the relationships of processes. Processes interact with different objects in memory. A clear understanding of how these relationships are normally expressed will assist in the identification of malicious anomalies. Using Cuckoo Sandbox [6] and diffing, malicious processes can be identified. Diffing entails executing a malware sample and examining the changes that are detected on that system. Any new or modified objects are considered malicious because the malware sample was responsible for these changes to the system. These labels will be used in the classification of processes as either malicious or benign.

1.6.2 What are common memory object relationships that malware tends to manipulate and/or establish in memory?

To answer this question, the datasets collected will be analyzed to reveal how malware with different types of behaviors manifests itself regarding memory object graphs. The hypothesis is that the relationships of malicious processes to other memory objects will look different from normal process relationships. These differences in relationships will be exploited to train classifiers. To prove the difference in object relationships between malicious and benign processes, Windows 7 memory images will be studied and utilized for training machine learning classifiers.

1.6.3 Do malware samples that have multiple processes have distinct path characteristics?

As a malware sample executes, there is assumed to be orchestration between processes as they perform their specified actions. As this question is answered, the aim is to identify the mechanisms commonly used by malware to communicate. This will be done by looking at the paths in our memory object graphs where the source and destination nodes are malicious processes. With this data, relationship paths can be studied to determine if any approaches are common to malware vs. benign software. For uncommon malicious paths, these can be used as features for our machine learning classifiers.

1.6.4 Do benign and malicious process paths have distinguishing characteristics?

With the path data collected, it can be used to identify differences in the paths that malicious processes use vs benign processes. To distinguish between types of paths, term frequency-inverse document frequency (TF-IDF) [25]) will be applied to our paths dataset. This technique is geared toward devaluing prevalent paths and highlighting (giving larger weighted values) less common paths and could indicate malicious activity.

1.6.5 Which features best support the goal of maximizing accuracy while minimizing false-negative rate in identifying malicious graphs?

In this stage, the feature selection process will be performed. Redundant and noisy features will be removed to improve classification accuracy and reduce the false-negative rate.

1.7 Research Benefits

The results of this work will be helpful to forensic analysts that have to identify malicious activity and often have to rely upon vast amounts of background knowledge and manual analysis. This will be simplified by automating the types of relationships that should be investigated. In addition, this research will support current investigations by making them more effective and lays the groundwork for the development of new tools in the future.

CHAPTER II

RELATED WORK

This section will introduce related research to the proposed topic. The body of knowledge is reviewed to understand what has been done in studying malware concerning random access memory and machine learning.

Thomas et al. [26] describe some of the basic procedures for extracting objects from memory. Some of the procedures included listing running processes, loaded DLLs, and extracting process memory from running processes. There has been much work that outlines steps and procedures for manually identifying malware in memory [27],[28][29]. These often suggest sequential steps that an analyst will follow, such as those outlined in SANS FOR526 [17]:

1. Identify rogue processes
2. Analyze process DLLs and handles
3. Review network artifacts
4. Look for evidence of code injection
5. Check for signs of a rootkit
6. Dump suspicious processes and drivers

This process is effective but efficiently performing these steps and building the background knowledge to do so correctly is difficult. Research has been conducted that aims to automate the detection process. For example, to identify userland keyloggers Case et. al. [30] created HookTracer and hooktracer_messagehooks to provide users other than expert investigators with this capability. Antivirus software can also be used to scan memory images, particularly with the advent of VirusTotal [31], which allows a file to be scanned by over 70 public malware detection engines. This method has shown some success, but it is not scalable and cannot be depended on alone. The difficulty with depending on antivirus companies is that they are dependent on keeping their analysis engines up-to-date and are consistently identifying infections from a reactionary perspective [32]. Studies have shown that, depending on the antivirus software, detection rates vary drastically [33] [34]. Memory forensics can also provide support for analysts that are attempting to identify ransomware. In work by Bajpai and Enbody [35] the cryptographic keys of malware families were found in system memory images. The use of machine learning to detect malware in memory is an area of research that has shown much promise. There are a variety of methods that have been used in the attempt to identify malicious behavior. Mosli et al. [36] had the goal of using features from registry keys, DLLs imported, and API calls to train machine learning classifiers. They created a processing environment using an Ubuntu system and a target Windows 7 system. Windows 7 was run on a bare-metal system to avoid the use of anti-VM techniques by malware samples. The Ubuntu system was responsible for resetting the target system each time a sample was executed. Artifacts were extracted using the memory forensics framework Volatility. The authors' used malware samples from VirusShare [37] and VX-Heaven to train classifiers with varying degrees of success. A Stochastic Gradient Descent model was created with registry activity to yield 96% accuracy, with DLLs an accuracy of 90.5% was achieved with random forest, and Stochastic Gradient Descent provided 93% accuracy with API function calls.

Glendowne et al. [38] used information collected from Windows 7 memory images generated by Cuckoo Sandbox to identify characteristics for DLL injection. After executing the malware, the authors developed heuristics for finding DLL injection from the resulting data. The following is a list of characteristics that were considered in the research: the process that was the target for injection, the number of injections that a particular sample performed, the times at which DLLs were loaded into multiple processes, the average load and initialization position for malicious vs. benign DLLs, common base addresses used when loading DLLs, imported and exported function counts, the directory that DLLs were loaded from, and determining if a DLL was used as a Component Object Model (COM) client or server.

Lashkari et. al. [39] created a tool based on the Volatility framework called VolMemLyzer. It was designed to extract the most critical characterization features from its memory dumps. The work tested a dataset of 1900 (1127 benign and 733 malicious) samples and aimed to classify a memory image as malicious or benign. The features used were collected from the following categories: processes, dynamic link libraries, handles, loaded modules, code injections, connections, sockets, services, drivers, and callbacks. The authors used four different machine learning classifiers with the Decision Tree and Random Forest classifiers performing the best with a True Positive Rate of 0.93 and a False Positive Rate of 0.066.

In work conducted by Bozkir et. al [40] the author leveraged Sysinternals utility ProcDump [41] to create memory dumps of a targeted process. The authors used the Windows Sandbox environment included in the 1903 Windows 10 operating system version to perform this collection.

The authors collected these dumps from ten classes of malware and used computer vision techniques to train machine learning models to classify processes as malicious or benign.

Mosli et. al [42] used handles from memory images and term frequency-inverse document frequency (TF-IDF) as a means for classifying processes as malicious or benign. The idea is that using TF-IDF would recognize handles that were most important for identifying malware. Duan et al. [43] developed a method of identifying malicious vs benign processes based on the DLLs that the processes contained. The authors used a hidden Naive Bayes (HNB) classifying algorithm and collected data from VirusTotal and memory images using Volatility for their training and evaluation sets. The authors also used data mining and frequent pattern sets to understand the functionality of the malware. This was done by developing clusters for a training set, tagging/labeling these clusters, and then using these clusters to provide insight into the abilities of the unknown samples. Aghaeikheirabady et al. [44] used features collected from _EPROCESS structures, the Virtual Address Descriptor (VAD) tree, mapped files, and registry keys as input to classify malware. In total, the authors used 130 features and collected the data from about 350 samples from Virussign [45] and VX-Heaven. Benign samples were gathered from a Windows 7 operating system. The authors did not discuss the method used to execute their samples or the type of tool used to extract information from memory. The features collected were used in the following machine learning algorithms:

- Sequential Minimal Optimization (SMO)
- Random Forests (RF)
- Decision Trees (DT)

- Naive-Bayes
- Instance-Based (IBK)

Naive-Bayes yielded the best accuracy and false positive rate at 0.989 and 0.016, respectively. As a component of a more extensive system, the authors of [46] use characteristics of API hooks as a feature for detecting botnets in memory. Srivastava and Jones [47] focused their research on detecting code inject in processes. They performed this analysis by reviewing the relationship between the Virtual Address Descriptor (VAD) tree and the processes stack. Their idea was that inconsistencies in the data returned from both objects would be a sign of code injection. In the work, the authors created a Volatility plugin that automated the process of checking these objects. The authors proved their approach by performing two case studies using the methods described. This work will continue to explore how malware represents itself in memory and develop techniques using machine learning to automate the detection process.

Cohen and Nissim [48] aimed to use memory forensics and features extracted from memory dumps to detect ransomware using machine learning. The detection was performed in the cloud on systems running on VMware ESXi. The features used were collected using the Volatility framework, and the operating system used was Windows Server 2012 R2. A subset of the original features was collected in the paper after selecting and ranking the most important features. Cohen and Nissim [48] ended with twenty-three features they felt provided a quality representation of a system state. The focus of these features was on core system elements such as processes, threads, services, etc. The authors used Cerber, TeslaCrypt, Vipasana, Chimera, and HiddenTear as the samples in their experiments to test their work. Multiple scenarios were tested in this work, represented in multiple experiments. The authors wanted to train classifiers to identify known states on a running server in experiment one: anomalous activity detection, distinguishing between benign and infected states and detecting specific states. In experiment two, the goal was to have classifiers select an unknown infected state, and experiment three targeted the classification of both unknown benign and unknown infected states. Experiment four had the goal of training classifiers to identify remote access trojans (RATs) and experiment five looked to discover unknown states on an email server. Experiments 1-4 were run on an Internet Information Services (IIS) server. The RATs used in the experiments were Babylon, Comet, DarkComet, Pandora, and SpyGate. The authors used nine commonly used machine learning classification algorithms that are listed below:

- J48
- Random Forest
- Naive Bayes
- Bayesian Networks
- Logistic Regression
- LogitBoost
- Sequential Minimal Optimization
- Bagging
- AdaBoost

In the first subset of experiments, where the first task detects an anomalous state, the second task is to distinguish between benign and infected states. The third task is the detection of specific known states the Random Forest classifier achieved perfect results in the three sub-experiments: True Positive Rate (TPR) = 1, False Positive Rate (FPR) = 0, Area Under the Curve (AUC) = 1, and F-measure = 1. True Positive Rate measures the number of correctly marked predictions as the true class (Example: Malware was predicted as malware). The False Positive Rate specifies the rate at which samples that were not positive were predicted as positive. (Example: Benign samples labeled as malware.) Area of the Curve is a common metric to display the performance of a binary classifier based on a comparison between the TPR and FPR. It shows how well a model can classify a sample compared to random guessing. F-measure is a weighted average between the metrics precision and recall. Precision measures the ability for a classifier not to label as positive a sample that is negative. Recall measures the ability for a classifier not to label as negative a sample that is positive. The second experiment evaluated the detection of unknown infected states on a virtual server. The best detection results were achieved by the Random Forest classifier: TPR = 0.9575, FPR = 0, AUC = 1, and F-measure = 0.975. In the third experiment, they evaluated unknown infected states as well as unknown benign states. The results showed that the best detection results were achieved by the Random Forest classifier: TPR = 0.922, FPR = 0.052, AUC = 0.966, and F-measure = 0.910. The fourth experiment detected the capability of detecting RATs. The LogitBoost, AdaBoostM1, and J48 classifiers provided perfect results in this experiment. Experiment five looked to examine detecting both ransomware and RATs on an email server. From the results the Random Forest classifier achieved the best detection results: TPR = 0.925, FPR = 0.058, AUC = 0.966, F-measure = 0.935. The focus of their work was not to identify a particular element on the system as malicious or benign but to take a comprehensive look at memory and determine if the system was detected to be in an infected state. In future work, the authors are interested in extending this work to the labeling of compromised machines as a whole.

Kumara and Jaidhar [49] use memory forensics and machine learning in their work, but they use memory forensics to extract binaries from memory and then apply machine learning with the information they have used from virtual machine introspection. It seems as though the authors apply n-gram analysis techniques from static analysis as the major input into their machine learning feature vectors.

Much work has focused on effectively finding malware in memory and employing machine learning methods for identifying malicious activity. However, to our knowledge, no work has focused on using memory forensics to study the object relationships within a memory dump and view it as a graph problem for identifying malware in memory. Furthermore, no work uses the combination of studying the characteristics of objects in a graph and the graph's structure as features to machine learning classifiers for identifying malicious vs. benign processes. Viewing the problem from this perspective provides the benefits of identifying a malicious process and leveraging the graph structure to pinpoint related abnormal objects of interest. The remainder of this work outlines the procedures used to do this and the results collected.

CHAPTER III

METHODOLOGY

This section will describe the steps taken to conduct this research. This includes data generation, Object Association Extractor (OAE) tool development, data extraction, feature development, and classifier evaluation.

3.1 System Design

The system used to generate infected memory images is based on the dynamic malware analysis sandbox called Cuckoo Sandbox [6] and a system developed by Miller et. al [50] here at MSU. This system allows malware to be automatically submitted to a clean virtual machine, executes and monitors the samples for a set time, captures the memory of the running VM, and then the VM is reset to a clean state snapshot. This process is repeated for each sample submitted. This system uses samples collected from the online malware repositories VirusShare [37], and VirusTotal [31]. The diagram in Figure 3.1 displays the Cuckoo architecture being used, which was designed by Glendowne in [51]. Three blade servers were used with slightly different hardware configurations. Two of the servers had twenty physical cores, while one server had sixteen cores. Both types were equipped with 128 GiB of RAM. According to work from [50], a Cuckoo host is a VM on a blade server running ESXi 5.5.0. This VM runs on Centos 7 OS and contains 20 additional virtual machines running Windows 7 using QEMU [52]. This "VM inside of a VM" setup was performed as an added layer of security to handle malware samples that attempt to escape a virtualized environment. QEMU was selected as the second layer of virtualization due to its speed and stability (See Miller et. al [50]) for the comparison experiment between VMware and QEMU. Each Windows 7 VM was configured with 512 Mb of RAM, one CPU core, Adobe Reader 11, Python2.7 installed, and had Windows firewall and User Account Control (UAC) disabled. Each of the VMs did not have Internet access but were connected to an isolated network with INetSim [53] running. INetSim was created to spoof various Internet services such as DNS, HTTP, and SMTP. As samples are passed to the Windows 7 VMs, they each run for 125 seconds (This value was also empirically tested in Miller et al. [50]). After this execution period, the VM is paused, and the memory images for the samples are collected and saved to a NoSQL CouchDB [54] database in JSON [55] format for later analysis.

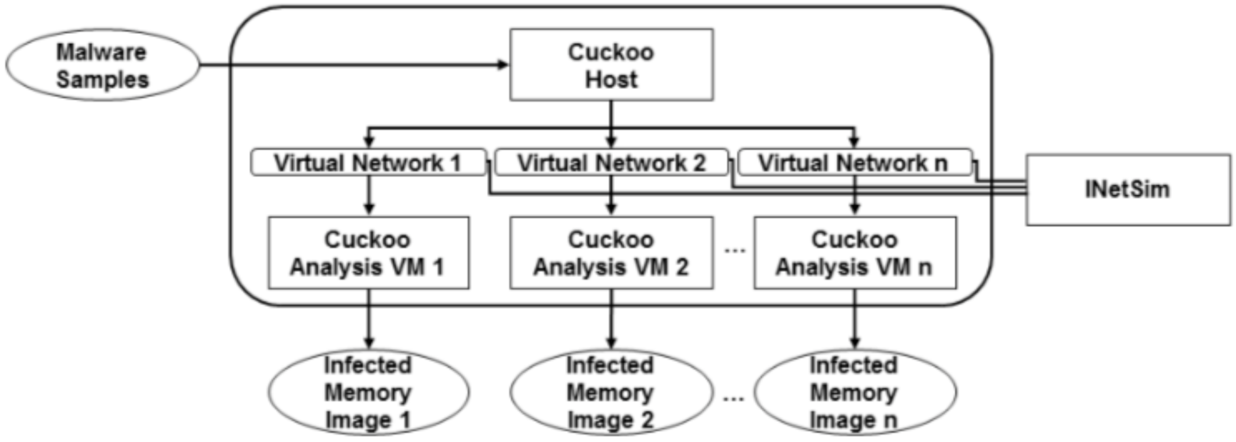


Figure 3.1 Cuckoo Sandbox System Diagram

3.2 Memory Object Mappings

The memory object mapping described in this work provides insight into how objects in memory fundamentally interact with one another. This work uses processes as the focus for identifying malicious activity. Processes are among the most critical objects due to their high level of connectivity to other objects in a memory object graph. To understand these interactions, the material provided in Ligh et. al. [56] and Russinovich [57] were used. Russinovich [57] provided context for what standard functionality looks like, and Ligh et al. [56] highlights how these object interactions are abused. The relationships reviewed are instances where an object creates, deletes, and can modify another object. A mapping of these relationships was created in a Microsoft Visio file, and then this information was used to develop the extraction tool to pull this data from a memory image in graph form.

3.3 OAE Development

A memory object graph tool called Object Association Extractor (OAE) has been created to identify and connect objects that are related to one another. Memory images created by the data generation system from Section 3.4 will be used to create memory object graphs. OAE is written in Python 3 and built on top of a memory forensics framework called Rekall [21]. Rekall was used to extract the characteristics of objects found in memory images. In addition to this, a network analysis package called NetworkX [58] was used to take information from Rekall and store it in graph form. When OAE processes a memory image, nodes and edges are generated as various object extraction modules are executed. Each module was responsible for populating different types of nodes to a graph. Related objects were then connected to others in that graph. The objects included in our work are listed in Table 3.1. A full write-up that describes the interactions between objects can be found in Appendix A. The current development is focused on userland objects of the Windows 7 operating system. Future work will include objects that run in kernel mode and apply similar methods to these objects.

The following section describes how OAE was used to extract data from RAM memory images.

3.4 Data Generation

For these experiments, memory images were filtered to a dataset containing at least two active malicious processes after running each malware sample through our malware execution system. The processes were labeled as malicious by using reports from Cuckoo Sandbox. The Cuckoo analysis system looks for indicators that are considered malicious and tracks other processes that are impacted by any malicious activity. To confirm that processes are active, we used the Rekall memory forensics framework. Rekall contains a plugin called psscan that scans a memory image for the data structure representing a process. This is called an _EPROCESS structure. This data structure has a member called ExitTime, which denotes when a process has stopped. The Cuckoo analysis report was used for each memory image to collect all malicious processes in the image. From this point, the ExitTime for each process was checked to ensure it was active. Once a memory image was confirmed to have two malicious processes, it was included in our dataset. The dataset contains 13,822 memory images. All malware samples used came from the VirusShare malware repository. This repository provides zip files of malware that can be downloaded for users with a verified account. Before using any samples from each VirusShare zip, they are run through a filtering process. First, the samples are extracted and checked to ensure they are valid portable

Table 3.1 Memory Object Descriptions

Object	**Description**
Atoms	Strings that can be shared between processes in the same session
Binary	Portable Executable file used to start a process
Callbacks	Notification routines used to signal events
Desktop	An object that represents the displayable screen surface and contains user objects such as windows, menus, and buttons
Device	An object created by drivers that handle I/O request packets
Driver	Represents the image of a loaded kernel-mode driver and contains addresses of the driver's input/output control handler functions
Event Hook	Notification used to signal UI-related events
I/O Request Packet	Structures used by devices and drivers to communicate with one another
Message Hook	Object used by applications to monitor and potentially record user activity
Module	An executable file that could include DLLs and kernel modules
Mutant	An object that represents mutual exclusion and is typically used for synchronization purposes or to control access to particular resources
Privilege	The permission to perform a specific task
Process	A container that allows threads to execute within a private virtual address space and maintains open handles to other objects
Registry Key	An instance of an open registry key that contains information about the key's values and data
Service	Program that executes in the background
Session	Outermost container for the GUI landscape
Network Connection	Established or attempted connections via TCP or UDP
System Service Descriptor Table	Table of pointers to API calls
Thread	An object that represents a scheduled execution entity within a process and its associated CPU context
Timer	An object used for synchronization and notification
Token	Stores security context information (such as security identifiers for processes and threads.)
Window Station	A security boundary for processes and desktops, which also contains a clipboard and atom tables

executable (PE) files. This check is performed by using the 'file' [59] UNIX command and the pefile [60] python library. Any file that is not a PE file is deleted. The next check uses the AVclass2 [61] project to identify any potentially unwanted programs (PUPs). PUPs are samples that fall into the category of being a nuisance, in general, and are not the target of this work. These samples are often referred to as adware. AVclass2 uses scans from VirusTotal to identify the families associated with a sample and if they are considered to be PUPs. Any PE file that is labeled as a PUP is deleted. Samples that make it through this filtering process are then submitted to our Cuckoo sandbox environment to be executed and have their memory images stored for data extraction. AVclass2 [61] is also used to provide a description of the types of malware samples included in our dataset. A wordcloud was created to display the most common malware families included in the dataset. This is displayed in Figure 3.2. The complete table of each malware family included in the dataset is listed in Appendix B. These malware samples are used to create the memory images that OAE uses.

3.4.1 Benign Memory Images Dataset

To increase the robustness of the memory images dataset, the Chocolatey [62] software manager was used to create memory images containing a variety of legitimate 3rd-party processes. Chocolatey itself is one of the leading software installation tools and integrates with many top open-source software automation and configuration platforms. A virtual machine was set up with a Powershell script that would download 20 binaries from Chocolatey, execute them, and then captures memory from the Windows 7 VM. This VM was configured with 6 GiB of RAM and two CPU cores. This method generated 91 memory images containing approximately 1,800 legitimate

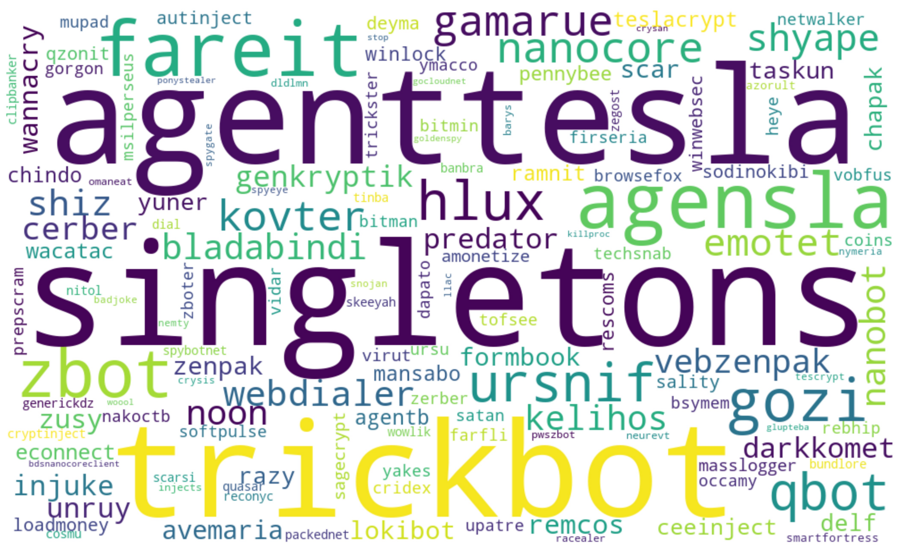


Figure 3.2 Malware Family Dataset Wordcloud

processes. These memory images were combined with those containing malware. Samples that are processed through Cuckoo produce an analysis report that will be used to label malicious processes in each memory image. This provides a form of ground truth for identifying malicious processes. All the processes from the Chocolatey processing will be labeled as benign. These labels offer the ability to annotate our datasets as they go through the process of feature engineering and model creation in Section 3.5.

3.4.2 Data Extraction

In this section, the data extracted from the memory images created will be discussed. This will include providing details on the process data being used and object relationship data. For each memory image created, the graph relationships are extracted. The extracted data from a memory image is called a memory object graph and will be collected using the OAE tool referenced in Section 3.3. Information about individual objects and relationships between objects in a memory image is extracted as a graph and saved to a NoSQL CouchDB [54] database in JSON [55] format for later analysis.

3.4.3 Process Data

For the process dataset, features are extracted primarily from information associated with _EProcess structures for each process in a memory image. Other data sources include plugins from Rekall that check for manipulation of objects by malware. From this, we separate the process specific information into its own dataset and save it into a MongoDB database for processing. The raw features in the dataset are provided below, with a description of each.

- *priv_info_list* collects a list of dictionaries that hold information about the privileges used by the current process. These can be indicative of the types of actions that a process is attempting to perform and if the privileges were loaded in a malicious manner.
- *handle_count_dict* is a dictionary of counts that keeps track of the types of handles that the process maintained.
- *handle_count_dict* is a dictionary of counts that keeps track of the types of handles that the process maintained.
- *token_info_list* contains a list of all of the token objects used by this process.

- *dll_info_list* contains a list of the dynamically linked libraries (DLLs) used by the current process.
- *pid* and *session_id* are the Process Identifier (pid) for the process and the session it belonged to.
- *ppid* is the parent process identifier.
- *phys_offset* is the physical offset in a memory image where an _EProcess structure is located.
- *create_time* is the time at which the process was started. The time is saved as an integer Unix epoch time.
- *exited* is a boolean value that identifies if the process is still active or has exited.
- *exited* is a boolean value that identifies if the process is still active or has exited.
- *type* is the object type.This is a type provided by OAE. For example, although each process is defined as an _EProcess structure, the type would simply be "process".
- *path* shows the file location to the binary that executes and starts a process.
- *launched_from_temp* returns a boolean value that identifies if the current process launched from some type of Windows temp directory.

The following features all return boolean values. They are used to identify if a process could have used Direct Kernel Object Manipulation (DKOM) to hide its presence from standard analysis tools that check for processes in a particular way. This information is collected by a plugin from

Rekall called psxview (process cross view). Its purpose is to show the alternative listings of a process within an operating system and compare them to identify any malicious discrepancies.

- *is_in_Sessions* - This feature checks the status of the SessionProcessLinks member of an _EProcess structure. This value associates all processes with a particular user's logon session.

- *is_in_CSRSS* - Client/Server runtime subsystem is used to create almost every process or thread on a system. The handle table for this process can be used to identify processes on a system.

- *is_in_PspCidTable* - This is a special table in kernel memory that stores references to all active processes and thread objects.

- *is_in_PsActiveProcessHead* - PsActiveProcessHead is a member of the kernel debugger data block (_KDDEBUGGER_DATA64) and is used to point to the head of a doubly-linked list of _EProcess structures.

- *is_in_Thrdproc* - Each process must have at least one active thread, so scanning for _EThread objects in memory and mapping the owning process is a way to cross-reference all processes.

- *is_in_PSScan* - Scan a memory image using the psscan plugin to find tags associated with _EProcess structures.

These features are used in the feature engineering and classifier evaluation stages to determine if they are predictive of malicious vs. benign processes.

3.4.4 Malicious Path Data

An important aspect of understanding malware is identifying its components and studying how these components potentially interact with one another in memory. Since each memory image in our dataset has at least two malicious processes, it is assumed that the malware components have some form of communication or synchronization. This would mean there are relationships between these objects used to perform their activities. The OAE tool extracts and graphs these relationships and provides information on the types of relationships that exist between objects. As an example, there are many parent-child relationships for processes in a given memory image. OAE will make each process a node in its generated graph and label this type of relationship with a connecting edge. Therefore, in each memory image, all paths between malicious processes will include the name of each object and a label that describes how the objects are connected. An example is shown below with Figure 3.3.

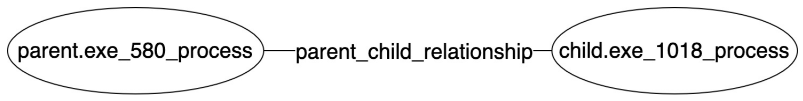


Figure 3.3 Path Example

The path example also highlights the labeling method used for objects in memory object graphs. The first segment shows the name of the object found in memory(parent.exe). Next, an identifier is used to label an object in memory. For processes, the process identifier (PID) is used. The physical offset is used for most other objects since this is a unique value for each object type. Finally, the object is given a label for its type in the graph. This example uses processes for each object in our path. The figure also illustrates the type of relationship by labeling the edge between the objects. Other objects follow a similar structure to the example outlined. From this data, additional tools were written that identify all the malicious processes in a memory image using the Cuckoo reports discussed previously. From these processes, all combinations of malicious processes for a memory image are found. For each pair, all paths between the two nodes are collected. This is performed by using NetworkX [58] and the function all_simple_paths. This function takes a source and target node from a graph and then generates all non-cyclical paths. This data will be used to create features based on the malicious paths collected. Transforming these malicious paths to be used with machine learning classifiers is discussed more in the next section.

3.4.5 Path Normalization

From the paths that are generated in the steps discussed previously, each path between malicious processes would have information that is specific to a given memory image. E.g. each process node in a malicious path would have a PID from its given memory image. The next step for collecting insight from our data is to transform the path data to a normalized format. From this format, the information that is specific to a memory image would be removed. This would leave only the object type for a node in the path and the relationship of that node to its neighbor in the path. An example of this normalization process is shown in Figure 3.4.

The paths normalized in this manner enable the review of data from all paths in the memory images dataset. The normalized paths can be used to determine the types of patterns that occur in the graphs generated from the memory images. We believe that normalized paths that occur frequently are generated from regular system activity. Malware can aim to exploit details of the

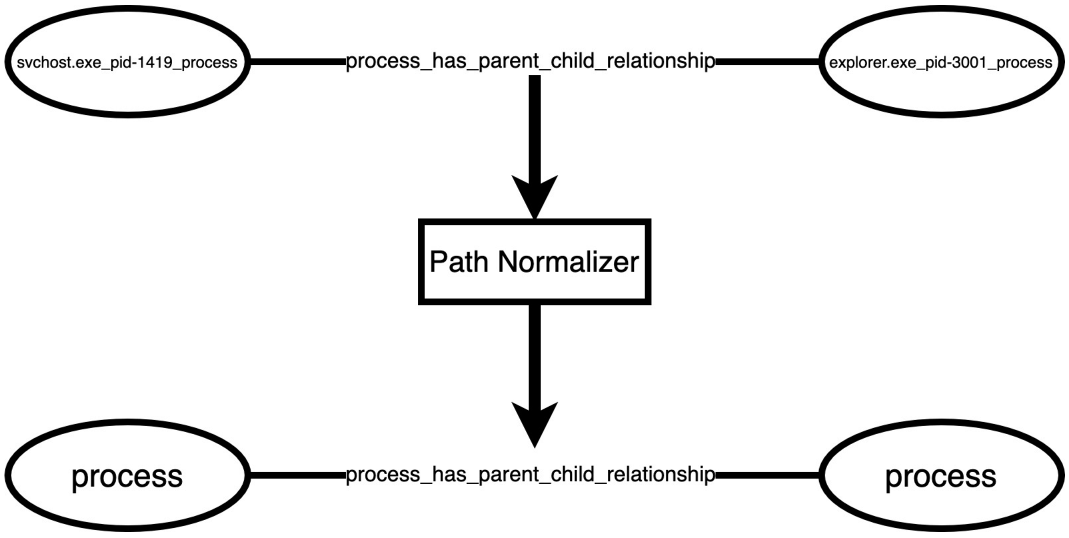

svchost.exe_pid-1419_process
process_has_parent_child_relationship
explorer.exe_pid-3001_process
Path Normalizer
process
process_has_parent_child_relationship
process

Figure 3.4 Path Normalization

operating system to achieve its tasks. Often it is the edge cases of these details that malware authors have discovered to use. Because of this, the focus of our analysis will be on less common paths in our normalized path dataset.

3.4.6 Path Analysis

The normalized paths of each memory image are used to determine if there are uncommon paths that have been generated. Malware authors can utilize different methods for performing malicious actions on an information system. In these experiments, empirical data is used to study the most common paths between malicious processes versus the most uncommon paths. Our intuition is that paths that are less common between objects could inform the machine learning model development process. We use Inverse Document Frequency (IDF) to analyze how a normalized path is represented based on the entire normalized path dataset.

IDF is commonly used as an information retrieval technique to find the overall importance of a word in an entire corpus. Its main benefit is devaluing common words or strings and giving a higher score to less common words of strings. An example of this is shown below:

- Document 1: "The school basketball team is great."
- Document 2: "The school newspaper is showing improvement."
- Document 3: "The theater club is great."

From the documents above we will generate IDF scores for each of the words. For this task we use the scikit-learn class TFidfVectorizer [63]. First lets start with the equation that TFidfVectorizer employs:

$$idf(t) = ln\frac{1+n}{1+df(t)} + 1 \tag{3.1}$$

n is the total number of documents in the document set, *df(t)* is the number of documents in the document set that contain the term *t*, and *ln* stands for natural log which is used by TFidfVectorizer to generate the IDF scores.

The larger the value returned, the more significant the word is in the dataset. A table of the idf values for this example is provided below:

From Table 3.2, it can be seen that common words such as "the" and "is" are weighted lower due to the frequency of the word occurring in the dataset. This same logic will be applied to the normalized path dataset.

The normalized path data is saved in our MongoDB features database. The normalized paths are collected and considered to be part of a single document for each memory image. All the

Table 3.2 TF-IDF Example

Word	IDF Value
basketball	1.69314718
club	1.69314718
improvement	1.69314718
newspaper	1.69314718
showing	1.69314718
team	1.69314718
theater	1.69314718
great	1.28768207
school	1.28768207
the	1
is	1

normalized paths from the memory image dataset make up the normalized path corpus. IDF looks at each path in the entire corpus and determines how frequently it occurs. Paths that are less common across all documents in the corpus will have a higher IDF score. The IDF table in Appendix C lists the top 150 normalized paths based on IDF score. These values were calculated using the TFidfVectorizer from Scikit-Learn. Additional parameters of note when generating these scores include setting a max_df of 0.85 and a min_df of 0.01. The max_df sets an upper bound on the percentage of documents a word can be found in. min_df sets a minimum threshold percentage for how many documents a word must be present to be included. These paths are used as features for our classifiers.

3.5 Feature Development

In our feature development process, the data collected and current trends of malware authors were studied for leads on creating additional features. These features were developed from two main sources. The first data source was information pulled directly from process objects in memory utilizing the attributes of a process node. The underlying data structure for a process, the _EProcess structure, will be used to collect the characteristics of the object. The second dataset uses path information between malicious processes discussed in the previous section.

3.5.1 Feature Selection

The feature selection techniques followed in this work go through two steps before training, validating, and testing our final models. The steps are outlined in Table 3.3 below:

Table 3.3 Feature Selection Techniques

Method	**Explanation**
Variation	The change in feature values across samples are key in identifying patterns for a model. Features that have no or low variation are not useful in training a model. Features with a variance of zero are dropped.
Stepwise Selection	The next procedure uses models to remove features and test their importance based on a metric such as recall. The features that have the most impact on the selected metric will be used while others are dropped. The specific type of stepwise selection used was Recursive Feature Elimination (RFE) [64]. The goal of RFE is to select features by recursively considering smaller sets of features. The model is trained with the initial set of features, and then the feature importance is calculated. The least important features are pruned, and the process continues until the best number of features are selected.

The feature selection methods chosen are fast and straightforward techniques that are widely accepted [64]. In Chapter V, the effectiveness of this approach is discussed.

3.6 Classifier Evaluation and Tuning

The objective of this work is to distinguish between malicious and benign processes in an automated manner. Machine learning classifiers use the datasets created above for this purpose. To perform the necessary machine learning work, Scikit-learn [65] was used. This Python library provides consistent, simple, and efficient tools for data analysis and model development. Three types of algorithms were selected: linear, non-linear, and an ensemble classifier. Based on these requirements, the following classifiers will be used:

- Logistic Regression (Linear): This algorithm works well with large datasets, and it is not computationally expensive. It was selected because models it produces can be trained quickly and have a track record of being easy to interpret.
- Decision Tree (Non-Linear): This is a fast algorithm that can work well with datasets with many features. It is an algorithm that does not require data normalization or scaling. A major benefit of using Decision Trees is their performance and the intuitive nature of understanding how a model makes its choices.
- Random Forests (Non-Linear Ensemble): Has broad versatility in the types of problems and data it can use. This algorithm also does not require normalization or scaling, which helps to simplify the data preparation process. Random Forest models often perform exceptionally well because of the ensemble approach that it takes. It overcomes one of the possible shortcomings of the Decision Tree algorithm, which is overfitting the training data, by using several different Decision Trees to improve accuracy. Alternatively, Random Forest models often take longer to build than Decision Trees.

- Gradient Boosting (Non-Linear Ensemble): This algorithm also has versatility and is commonly used as winning solutions for machine learning competitions such as Kaggle [66]. It is also an ensemble method similar to Random Forest. A significant difference is how trees are built. Random Forest builds all of its trees independently of one another. With Gradient Boosting, each new tree built is done with respect to previous trees to correct errors made by the previously trained trees.

Using these types of algorithms provides the capability to explore different methods of identifying correlations between the features selected and the predicted class for the samples. Each model uses the same training, validation, and test datasets, while the features for each varied depending on the performance of each classifier. Each classifier was evaluated to determine which model best maximized the recall of classifying processes.

CHAPTER IV

THREATS TO VALIDITY AND LIMITATIONS

In this section, the threats to validity are discussed. By this, we mean issues that could impact experiments, the types of data that have been selected, the limitation of various algorithms, etc. These caveats are being addressed to identify problems and highlight solutions.

4.1 Problem: Information loss due to sandbox evasion

Not every sample that has been processed by our system exhibits its full array of capabilities. Some samples need to download additional components by communicating with a command and control server and will not continue. In other samples, malware authors have begun developing samples that are environmentally aware during their execution and evade dynamic analysis environments [67]. Freely available software, such as PAFish [68], provides simple checks to find dynamic analysis VMs that have minimal RAM configurations, minimum processors or hard drive space requirements, or registry keys that can identify a virtualized environment. Some other samples take things a step further by searching for specific user interactions or evidence of system usage [69]. Our system attempts to mitigate some of these problems with the following hardening techniques:

- Added additional user files to the VM (PDF, TXT, MP3, MP4, and JPEG files were added).
- Installed additional software: Firefox, Notepad++, VLC, etc.

- Installed several versions of Microsoft Visual C++ and Microsoft .NET frameworks
- Ensured programs and documents were opened during the time each malware sample is executed

Even with such precaution, there is still a diverse pool of malicious checks that a malware sample can perform.

4.1.1 Potential Outcome

It was believed that with a large number of malware samples at our disposal, about 3.17 TBs, a subset of samples would be found that exhibit their functionality to train and test quality classifiers. If a problem arose with the number of malware samples available, machine learning techniques such as k-fold cross-validation [70] would be used to fully utilize the data that is available as efficiently as possible. K-fold cross-validation takes a dataset and divides it into slices depending on what 'K' is selected. A machine learning model is trained on K-1 pieces of the dataset, and then its performance is tested on the hold-out slice. This process is repeated where each slice is held out, and all other portions are trained on. The result of this activity is an average of how the model will perform on the provided dataset.

4.2 Guaranteed maliciousness

Malware samples used in this work are collected from VirusShare [37], and VirusTotal [31]. These communities provide repositories of various types of malware samples that have been detected. These resources rely on community submission, so not all samples included in the repositories can be identified as malware. Also, many samples provided can be classified as

Potentially Unwanted Programs (PUPs). These types of programs are considered bloatware/adware and are not the target of this work.

4.2.1 Potential Outcome

To minimize PUP samples being included in our datasets, we use AVClass2 [61] to filter what samples were included. Although not all samples could be classified as malicious, VirusTotal provides the ability to scan malware against known anti-virus signatures to understand how malicious a sample is considered to be. VirusTotal also provides an API to programmatically check the hash of a sample if a growing concern for the legitimacy of malicious samples comes into play.

4.3 Missing some relationship information due to exited processes

As malware executes and starts processes, some samples will exist that are responsible for spawning new processes or injecting code into other processes and then exiting themselves. This procedure could lead to the loss of relationship information for those processes that exit. The memory for processes that have exited is marked as available. The data from an exited process is not immediately removed from memory but can be overwritten once new processes are created. Currently, processes that have exited are filtered from our process datasets for machine learning.

4.3.1 Potential Outcome

Although exited processes can cause problems with being able to identify their relationships with other objects in memory, the running processes on a system and the relationships that those processes create are utilized.

4.4 Find a method for collecting additional benign memory samples

The package management software called Chocolatey [62] is used to introduce 3rd-party processes into the benign dataset. Without these samples, the benign processes used primarily include Windows binaries. Utilizing Chocolatey provides a more diverse pool of processes to help make more robust machine learning classifiers. One drawback of this method is that most of the samples are installation programs. Developing a way to include actionable benign programs would provide additional realism to the benign process subgraph dataset.

4.4.1 Potential Outcome

In the worst case of not having enough benign software, the less desirable option of creating benign memory images manually can be explored in the future.

4.5 Develop a large enough dataset to support the number of features selected for training classifiers

When the number of features used to train a machine learning classifier increases, there is a need for more samples to avoid classifiers overfitting on training data. The problem that can be encountered is the curse of dimensionality [71] when a large number of features are used as well. A large enough dataset size must be available or the feature space used must be reduced to create quality classifiers.

4.5.1 Potential Outcome

To make sure the curse of dimensionality is not a problem, this work uses dimensionality reduction techniques such as feature selection. Also, to confirm that the dataset is the appropriate size, learning curves [72] are plotted to determine the right dataset sizes. An iterative process of removing features and checking the number of data points needed for a given algorithm will be performed.

4.6 Create a dataset of samples that are representative of malware and benign software that is used in the real world

It is not possible to create a dataset that will represent all possible malware and benign software available to execute on a Windows 7 system. This causes a problem trying to train a classifier that will generalize well to samples outside of the current training set.

4.6.1 Potential Outcome

Although we cannot confirm that the malware samples from VirusShare are entirely representative of all other malware currently active, these repositories have been used by a body of work for malware analysis research [73] and machine learning classifier creation.

CHAPTER V

MACHINE LEARNING WORKFLOW

In this section, the full machine learning workflow created is discussed. The pipeline includes six steps:

1. Generate TF-IDF features

2. Connect Processes to Normalized Paths

3. Data Filtering and Feature Engineering

4. Building Machine Learning Models

5. Final evaluation

The system used to generate these results was the Orion supercomputer at MSU. Orion is a PowerEdge C6420 Linux Cluster with 72,000 2.4GHz Intel Skylake processor cores, 345 terabytes of RAM, and a Mellanox HDR100 InfiniBand interconnect. Orion's peak performance is 3.66 PetaFLOPS and debuted at 62nd overall and 4th in US academia on the June 2019 TOP500 list. On the November 2019 TOP500 list, it was upgraded to 60 overall. To execute all code in this workflow, the web-based user interface JupyterLab from Project Jupyter [74] was used. How Orion was utilized in each aspect of the workflow is described in the following sections. Unless otherwise stated, the machine learning pipeline used an Orion development node to run our Jupyter notebooks. The development nodes contain two 2.4GHz Intel Xeon Gold 6148 Skylake CPUs. The CPUs contain twenty cores each, for a total of forty cores per node. It also has twelve sticks of 16GB DDR-4 Dual Rank 2666MHz, for 192GB of memory.

Additional tools leveraged in this work were Singularity [75] containers. Singularity provides the ability to have a consistent system environment across platforms. This enabled code snippets to be tested quickly on a local system before being used on the Orion cluster. It also streamlined the installation process as Orion does not have direct access to the Internet.

5.1 Generate TF-IDF features

As discussed in 3.4.6 IDF is used to determine uncommon normalized paths that malware samples leverage. The top 150 normalized paths of length four are used. The normalized paths of length four include those that have four edges between two processes. Longer path lengths may be more descriptive of the types of activities performed by malware. Additionally, using longer-length normalized paths can be explored when more computing resources are available. Each memory image in our corpus was used to create a document of all paths within that memory image. The document creation procedure was parallelized using threads. The generated documents were then passed to our IDF procedure to select the 150 normalized paths mentioned above.

5.2 Connecting Processes to Normalized Paths

The processes from malicious images, benign memory images, and the normalized paths were stored in separate collections in our MongoDB database. In this step, both types of processes are pulled from the database and saved into a Pandas [76] dataframe. Each process in our dataset has a sample_id that connects a process to the memory image it was extracted from. The normalized paths extracted are also associated with a sample_id, which is used to connect this information. Each process is labeled with a boolean value based on if it was found in a chocolatey memory image (benign) or not. Each normalized path in the dataframe contains either a 0 or 1. 1 for the processes associated with a normalized path and 0 otherwise. The dataframe object is saved to disk and passed to the next section of the workflow.

5.3 Data Filtering and Feature Engineering

In this section, the collected data is filtered to include only relevant samples, and feature engineering is applied to generate features used by the machine learning models. In the filtering step, processes that are exited are removed from the dataset. These samples are missing information needed to generate features, and so they are removed. The section below describes features that have been developed and the process data used to create them.

5.3.1 DLLs

A list of DLLs associated with a process was pulled from memory for each process in the dataset. It is from this information that the following features were created.

- num_dlls - This feature is a sample count of how many DLLs were associated with a given process.
- dll_name_manipulation_count - This features checks to see if a DLL has been unlinked from a process listing but still has a full path to a location on disk. Malware authors will attempt to inject a DLL and write a non-suspicious pathname to avoid detection.

- inmemorder_listing_false_count - This feature counts the number of DLLs for a process where the InMemoryOrderModuleList shows a value of False. The InMemoryOrderModuleList is a linked list that organizes modules in the order they appear in the process's virtual memory layout. Malware authors can unlink from this listing to try and hide a DLL.
- inloadorder_listing_false_count - This feature counts the number of DLLs for a process where the InLoadOrderModuleList shows a value of False. The InLoadOrderModuleList is a linked list that organizes modules in the order they are loaded into a process. Malware authors can unlink from this listing to try and hide a DLL.
- ininitorder_listing_false_count - This feature counts the number of DLLs for a process where the InInitializationOrderModuleList shows a value of False. The InInitializationOrderModuleList is a linked list that organizes modules in the order in which the DLLMain function was executed. Malware authors can unlink from this listing to try and hide a DLL.

5.3.2 Handles

Processes will have several different handles present as they attempt to perform their various activities. In this section, the handle counts for a process are transformed into features. The list of features collected are shown below:

- total_handle_count - This feature counts all of the handles associated with a given process.
- process_handle_count - This feature counts the number of handles a process has to other process objects.

- key_handle_count - This feature counts the number of registry keys to which a process has a handle.
- file_handle_count - This feature counts the number of files that a process has handles to.
- window_station_handle_count - This feature counts the number of window stations that a process has handles to. A window station is an object that acts as a security boundary for processes and desktops. For example, the window station can be used to determine if malware authors are trying to snoop on the clipboard to do such things as steal credentials.
- thread_handle_count - This feature counts the number of handles to threads that a process has.
- desktop_handle_count - This feature counts the number of desktops handles. Malware can use desktops for launching applications out of the view of a current user or even creating a type of ransomware that works by locking a user inside of a desktop.
- mutant_handle_count - This features counts the number of mutexes that a process has handles to. Malware will use a mutant/mutex for actions such as avoiding the reinfection of a system.
- token_handle_count - This feature counts the number of tokens that the current process has handles to.

5.3.3 Privileges

Malware authors often have the goal of increasing their privileges to have more control over a system. These features check for common targets that malware likes to use and check to see if they are enabled properly. Privileges that are enabled should also be present in a process by default. If a privilege is enabled but does not have a privilege by default, this is a sign that they have been escalated maliciously. The features below are used to capture these potential actions:

- num_priv_present_enabled_not_default - This feature counts the total number of privileges for a process that are enabled but not present by default.
- SeBackupPrivilege_present_enabled_not_default - SeBackupPrivilege allows access to any file on a file system regardless of an access control list. This can be used by attackers to copy locked files.
- SeDebugPrivilege_present_enabled_not_default - SeDebugPrivilege allows the ability to write to or read from the memory space of another process. This is a common target for any malware that needs to perform DLL injection.
- SeLoadDriverPrivilege_present_enabled_not_default - SeLoadDriverPrivilege allows the ability to load and unload kernel drivers.
- SeChangeNotifyPrivilege_present_enabled_not_default - SeChangeNotifyPrivilege is designed to allow a callback function to be set any time a specific file or directory has changed.
- SeShutdownPrivilege_present_enabled_not_default - SeShutdownPrivilege allows a process to reboot or shut down a system. Some malware does not show its functionality until a reboot and tries to speed up the process by using this privilege.

5.3.4 Tokens

This final section calculates a simple count of the number of tokens associated with a process. This value is saved in the feature num_tokens.

5.4 Building Machine Learning Models

Additional consideration and actions needed to be performed on the dataset before building our machine learning models.

5.4.1 Filtering Chocolatey Processes

In reviewing the benign processes dataset, it was found that the processes from Chocolatey images also included the standard Windows processes from each Windows system. The purpose of including Chocolatey was to provide variety in the dataset, so it was determined the Windows system processes from Chocolatey needed to be excluded from the dataset.

5.4.2 Training/Validation/Test Set Creation

After filtering the Chocolatey processes, the processes dataset was split into a training, validation, and test set. The training set is used to build the initial set of machine learning models. The trained models are then used to predict if a process is malicious using the validation set. Finally, the training and validation sets will be combined to predict the test set to assess each model's performance with its best features. The training set was structured to 70% of the total dataset, while the validation and test sets were given 15% respectively. The split was also performed in a stratified manner across the Chocolatey samples and the is_malicious target variable. The breakdown for the train/validation/test split is shown below in Table 5.1:

Table 5.1 Training/Validation/Test Breakdown

	Total Processes	Malicious Processes	Non-Malicious Processes
Full Dataset	267,824	48,108	219,716
Training Set	187,476	33,675	15,3801
Validation Set	40,174	7,217	32,957
Test Set	40,174	7,216	32,958

Looking at Table 5.1 concerns could be raised based on the discrepancy between the number of non-malicious to malicious processes, with the ratio being over 4:1 in favor of the non-malicious processes. However, this dataset breakdown is representative of the real world. In practice, malicious artifacts often are outnumbered by benign processes running on a system, so the imbalance of malicious to benign processes is not a concern.

5.4.3 Chocolatey Processes Oversampling

The overall number of non-Chocolatey samples compared to Chocolatey samples still had a large disparity with a ratio of about 339:1. To improve this, we leveraged oversampling techniques to address this imbalance problem. The basic idea behind oversampling is to use a method to duplicate data in the minority class of a dataset to make it more balanced. In this work, we leveraged two techniques mentioned in work by He and Garcia [77] and Branco, Torgo, and Ribeiro [78]. The first is random resampling. The technique is simple where you take the minority class and randomly select a percentage of the sample to copy and add to the original dataset. A more complex method is Synthetic Minority Oversampling Technique (SMOTE), presented in Chawla et. al. [79]. This method basically generates synthetic data from a sample of the minority class and its nearest neighbors. For each oversampling technique, the minority class was increased to be twenty percent of the majority (non-Chocolatey) class. The oversampling was only applied to the training dataset to improve the fit of the models. The overall performance will be still be evaluated on the validation and test sets that do not include any oversampled data points. The general breakdown is shown in Table 5.2.

Table 5.2 Chocolatey Dataset Breakdown

	Total Processes	Non-Chocolatey Processes	Chocolatey Processes
Training set	187,476	186,925	551
Training Set (SMOTE)	224,310	186,925	37,385
Training Set (Resampled)	224,310	186,925	37,385

5.4.4 Feature Selection

During the feature selection steps, the training and validation sets are used. The training set will train our models, and predictions were made on the validation set. To perform the feature selection procedure, a combination of software packages were used. First, a Python library called Papermill [80] was leverage. Papermill allows for the parameterization of a Jupyter notebook. In their normal use-case, a user runs Jupyter notebooks actively to review and interact with results. With Papermill, a base notebook is used, and different options are passed to perform the needed tasks. In our case, Papermill was used to create a Jupyter notebook for each machine learning model trained, with the resulting models being saved for the next step in the workflow. The scikit-learn [65] models used allow for parallelization during the training process. To apply this capability, Apache Spark [81] and joblib-spark [82] were used. Apache Spark enables distributed computing, and joblib-spark provide the ability to leverage that computing power with scikit-learn. Finally, Orion uses the Slurm workload manager to submit jobs that need to be executed. From the software mentioned above, the following solution was created. A Python script was written to submit Slurm jobs to Orion. In this script, Singularity containers were used to create each machine learning model by running parameterized notebooks with Papermill. Each job was configured to use Spark and was given twenty-five cores to execute its tasks.

Before any model training, the variance of each feature is checked. If a feature has a variance of zero, then that feature is removed from the dataset. A feature with zero variance displays no changes and does not provide any value in identifying patterns associated with the target variable is_malicious. Next, Recursive Feature Elimination (RFE) is used to find the optional number of features to maximize the recall of each model. This procedure is performed using stratified cross-validation. RFE works starting with the full set of features provided to the model and recursively considers smaller and smaller sets of features. After each iteration, the features with the least importance are pruned from the current set of features. For each fold of the cross-validation, it is taken into account as the optimal features are selected. The results generated are for four different models with two data types (SMOTE and Resampled training data.) For each classifier the total number of features used are provided in Table 5.3 and the performance of the classifiers on the validation set are provided in Figures 5.1 and 5.2. In the classification reports, four metrics are being displayed. First is the precision score. Precision measures the ability of the classifier not to label as positive a sample that is negative. The equation for precision is shown in Equation 5.1

$$precision = tp/(tp + fp) \tag{5.1}$$

- tp = The number of positives
- fp = The number of false positives.

Next, the recall score is shown. This measures the ability of the classifier to find all the positive samples. The equation for the recall is shown in Equation 5.2

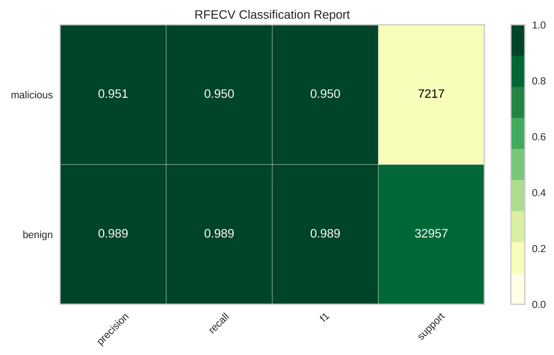
$$recall = tp/(tp + fn) \tag{5.2}$$

- tp = The number of positives
- fn = The number of false negatives.

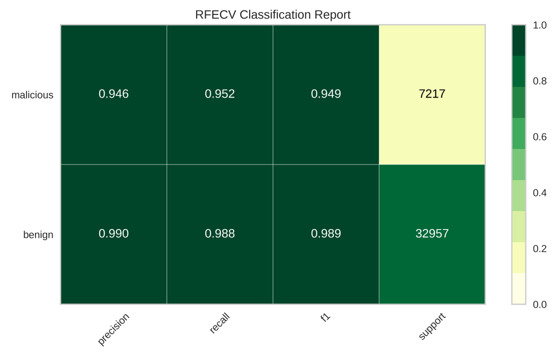
The F1-score is a weighted average of the precision and recall scores. The best value is 1, and the worst score is 0. This equation can be found in Equation 5.3.

$$F1 = 2 * (precision * recall)/(precision + recall) \tag{5.3}$$

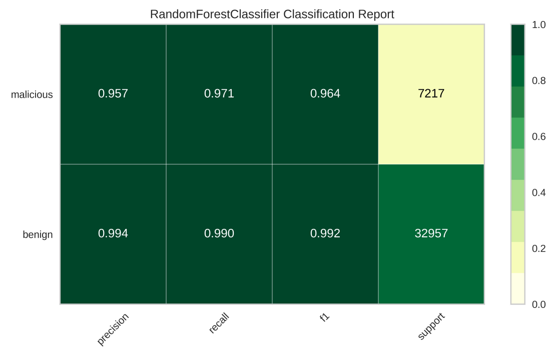
The support metric details the number of samples that are present for each class that is being predicted. The support for the benign and malicious processes match the value shown previously in Table 5.1.



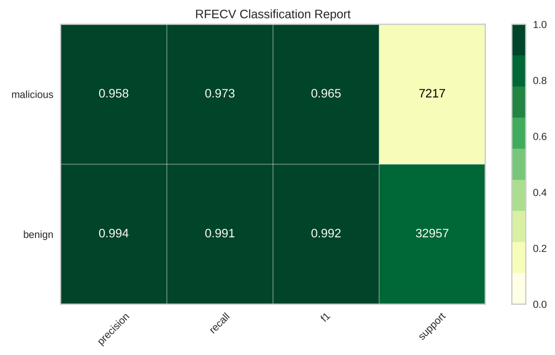
(a) Decision Tree SMOTE



(b) Decision Tree Resampled

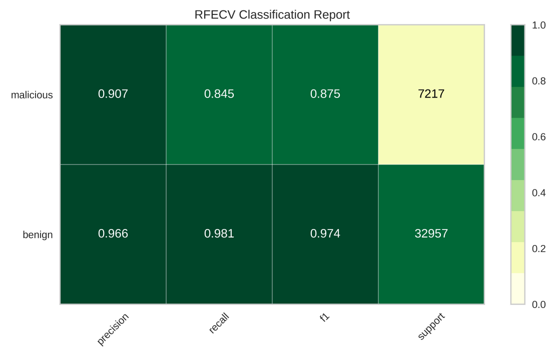


(c) Random Forest SMOTE

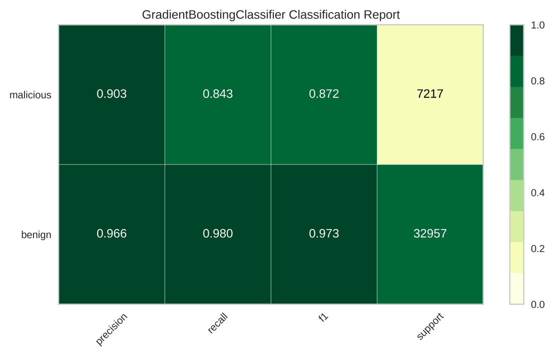


(d) Random Forest Resampled

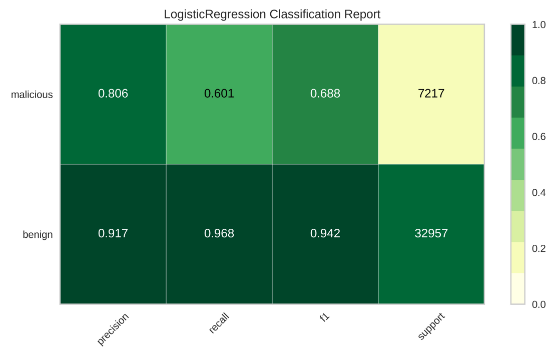
Figure 5.1 Classification Reports for DecisionTree and RandomForest Classifiers



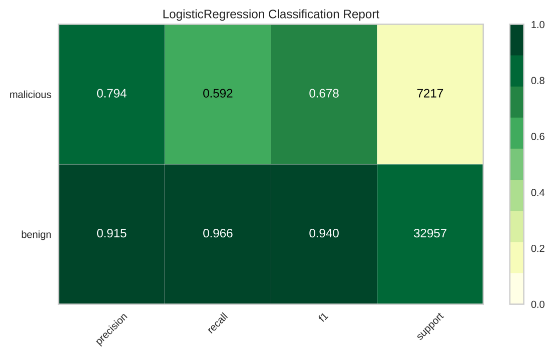
(a) Gradient Boosting SMOTE



(b) Gradient Boosting Resampled



(c) Logistic Regression SMOTE



(d) Logistic Regression Resampled

Figure 5.2 Classification Reports for LogisticRegression and GradientBoosting Classifiers

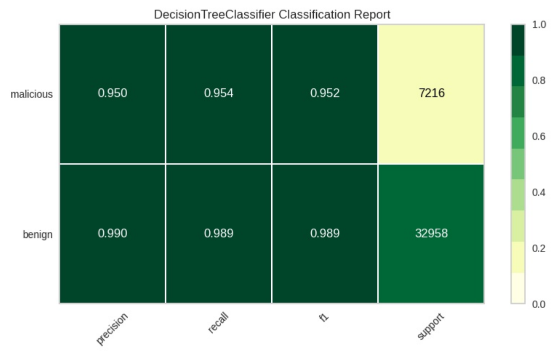
Table 5.3 Classifier Feature Count

Classifier Name	**Number Features Selected**
DecisionTree Smote	27
DecisionTree Resampled	105
RandomForest Smote	24
RandomForest Resampled	24
GradientBoosting Smote	11
GradientBoosting Resampled	9
LogisticRegression Smote	166
LogisticRegression Resampled	167

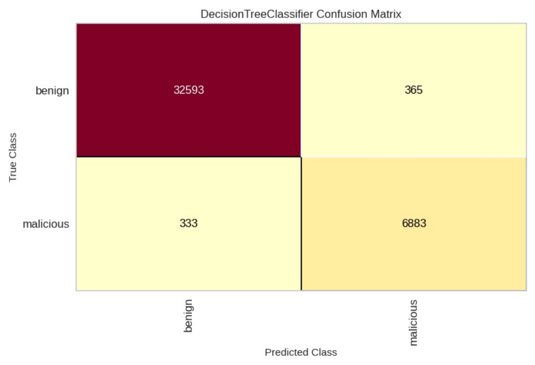
Overall, each of the classifiers performed reasonably well. However, as the stats are reviewed more closely, it can be seen that the Logistic Regression classifiers struggled with the metric recall. Specifically, neither performed well relative to the other classifiers in limiting false negatives. This is important because the worst performance for the classifier in our context would be to label a process as benign that is actually malicious. The four highest-performing models use the Decision Tree and Random Forest classifiers. These models will be finally be evaluated on the test set in the next step.

5.5 Test Set Predictions

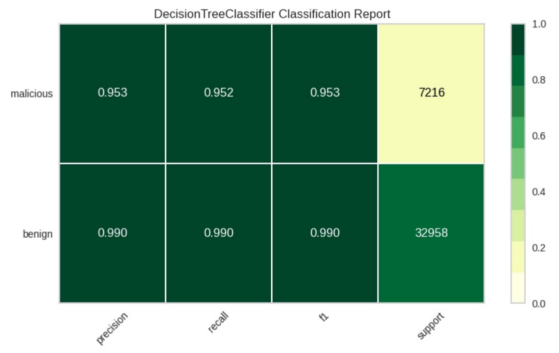
In this step of the machine learning workflow, the final trained models are used to predict the target variable for the test dataset, and then we interpret the performance of each model. The training and validation datasets are combined and used as the final training dataset. Each model is trained, and then predictions are generated. The performance of the top four models are listed in Figure 5.3 and 5.4. In addition to the classification reports for each model, a confusion matrix is shown to get a detailed breakdown of the types of errors the models made for each class type.



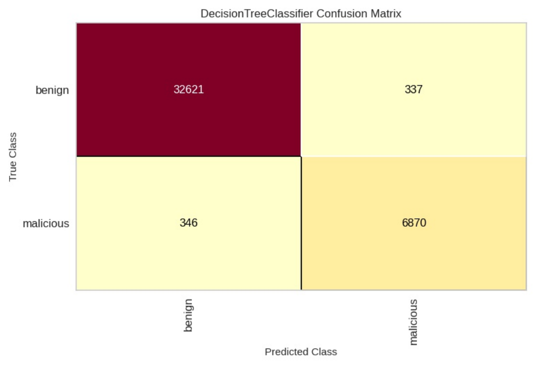
(a) DecisionTree SMOTE Classification Report



(b) DecisionTree SMOTE Confusion Matrix

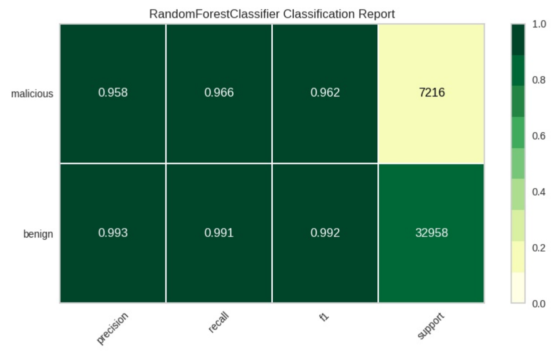


(c) DecisionTree Resampled Classification Report

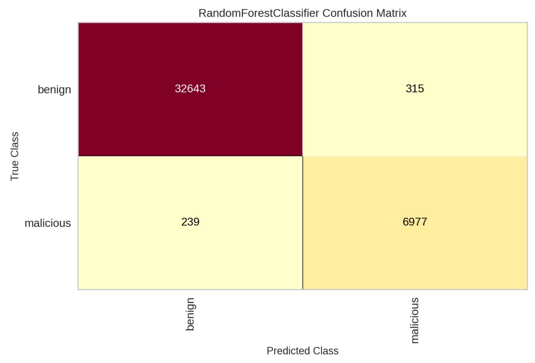


(d) DecisionTree Resampled Confusion Matrix

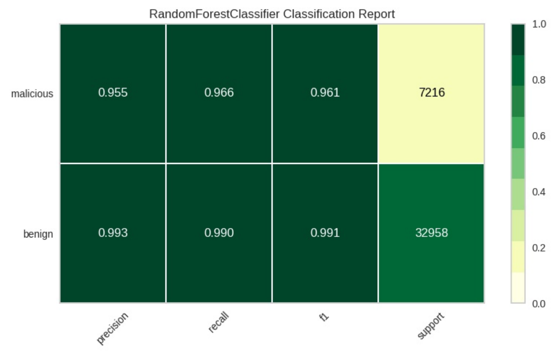
Figure 5.3 Classification Report and Confusion Matrix for DecisionTree Classifiers



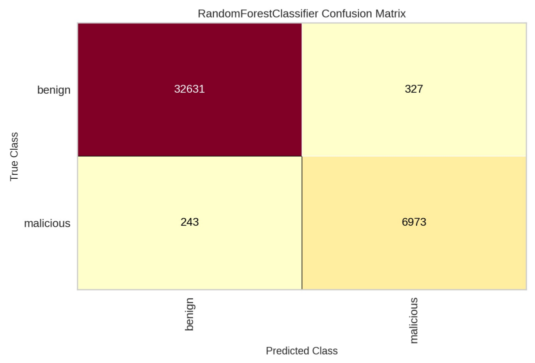
(a) RandomForest SMOTE Classification Report



(b) RandomForest SMOTE Confusion Matrix



(c) RandomForest Resampled Classification Report

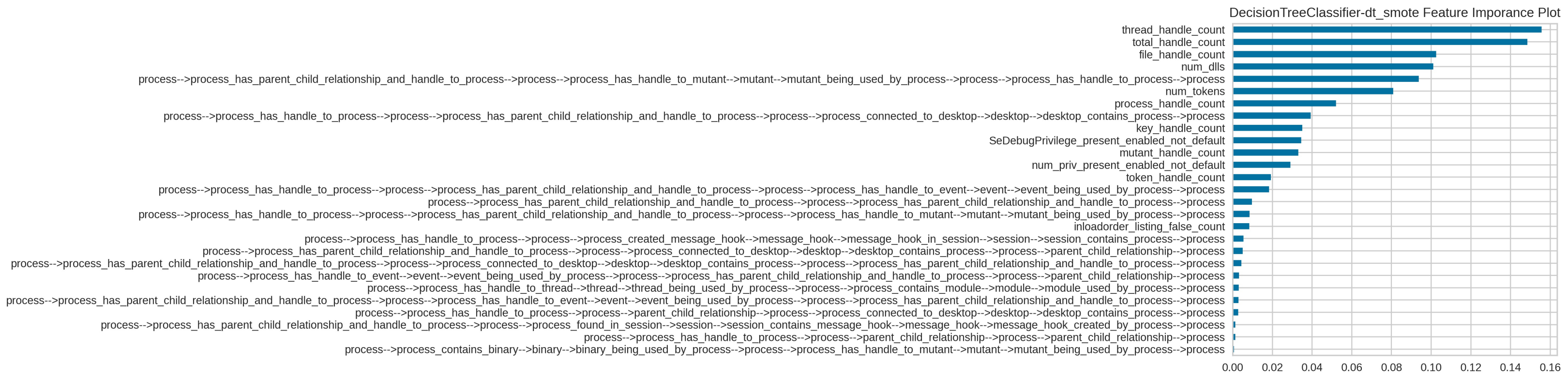


(d) RandomForest Resampled Confusion Matrix

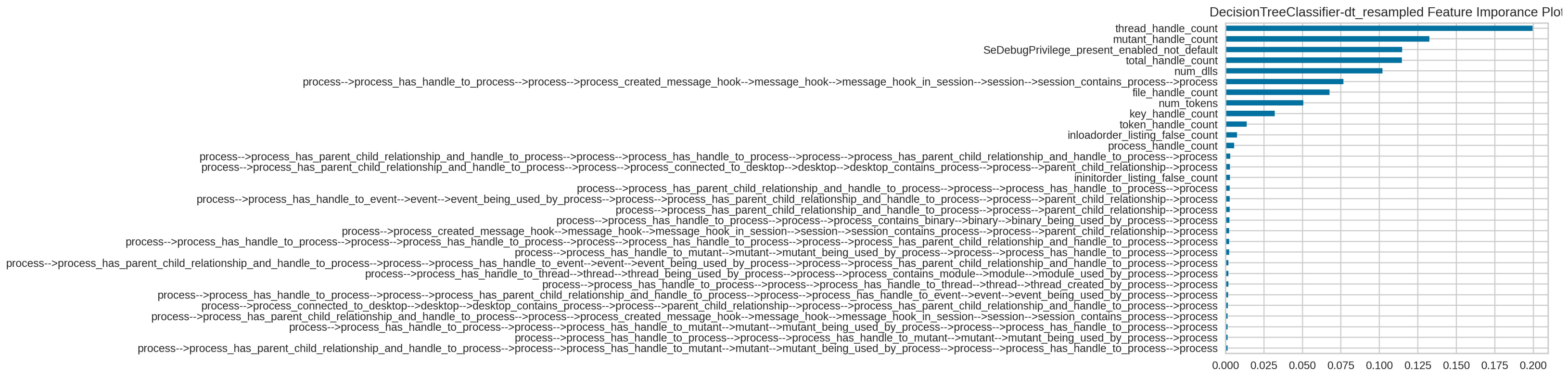
Figure 5.4 Classification Report and Confusion Matrix for RandomForest Classifiers

5.5.1 Model Interpretation

Each of the top four highest-performing models selected different subsets of features to achieve their scores. The feature importance charts for each classifier are shown below for Figures 5.5 and 5.6. An unexpected result was found with each model using the handles features to indicate malicious or benign processes. The top feature for each model was thread_handle_count. This counts the number of handles that a process has to other threads. In reviewing the data from all datasets, the mean number of thread handles is lower for malicious processes than benign processes. The malicious processes had, on average, 26 thread handles per process compared to 29 for benign processes. Another significant feature for the models was total_handle_count. Again comparing this feature for both classes, the malicious processes had 383 handles per process and 281 for benign processes. This matches our line of reasoning because malware will often have to establish handles to objects they need to manipulate. Another feature shared by all the models was num_dlls. This simply counts the number of DLLs that have been loaded by a process. The mean value between malicious and benign processes is 69 to 49, respectively. Depending on its purpose, malware will need to load different DLLs to achieve the tasks it was designed to perform. This number, on average being higher than the benign processes, is also reasonable in our estimation. The feature SeDebugPrivilege_present_enable_not_default is in the top 10 for importance with all four models. SeDebugPrivilege is a privilege routinely used by malware authors to access other processes and perform injection activity. The SeDebugPrivilege being present and not enabled by default is also a major indicator of malicious activity.

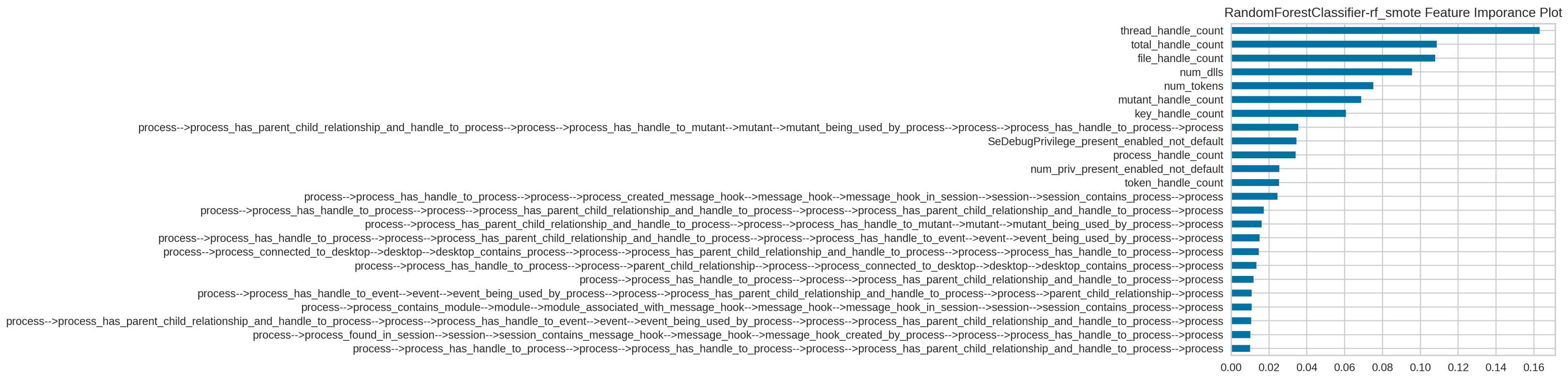


(a) DecisionTree Feature Importance for SMOTE Data

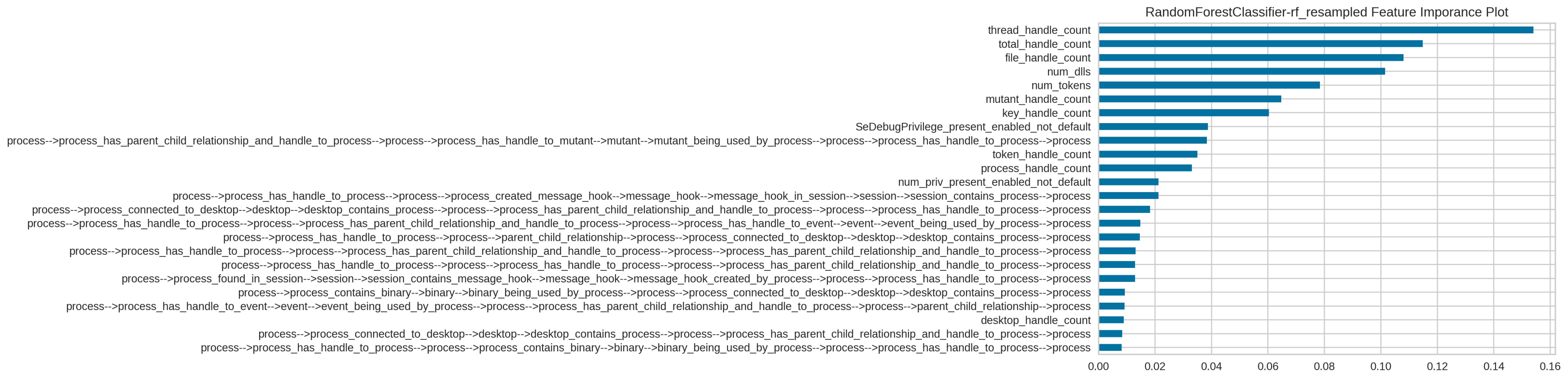


(b) DecisionTree Feature Importance for Resampled Data

Figure 5.5 DecisionTree Feature Importance Charts



(a) RandomForest Feature Importance for SMOTE Data

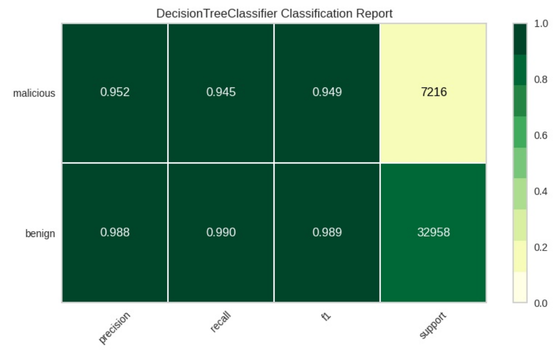


(b) RandomForest Feature Importance for Resampled Data

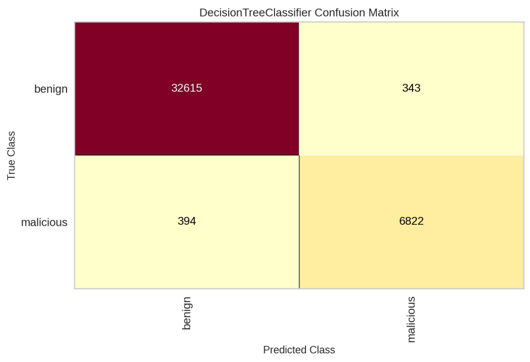
Figure 5.6 RandomForest Feature Importance Charts

Looking at previous research such as Mosli [42] they also used features such as different types of handles for processes to perform classification. Since this work had been done previously, it was decided to separate the features into those extracted directly from the process versus the normalized path features. A breakdown of the performance with process-specific features is shown in Figure 5.7 and 5.8. Again, all four models show strong performance, with the RandomForest Resampled showing the strongest recall scores of almost 97%.

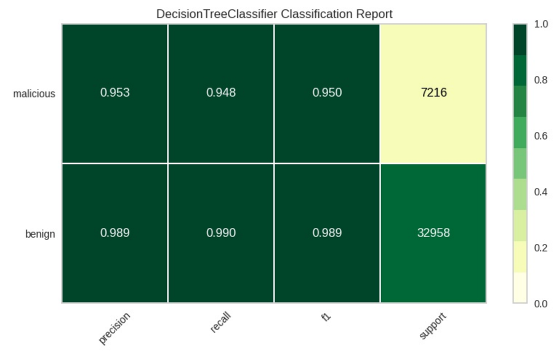
With the models trained on the process-specific features performing so well, we also explored how our models performed with normalized path features only. The performance of these models are shown in Figures 5.9 and 5.10. In viewing these results, it is evident that the models have a decrease in performance for both precision and recall of the malicious samples compared against the process-specific features. The best-performing model based on recall that only used normalized path features was the DecisionTree Resampled model. Its recall score was about 75%. For this case, the normalized path features serve well to supplement the process-specific features to boost model performance when using all features, but additional work needs to be performed to use them as stand-alone features.



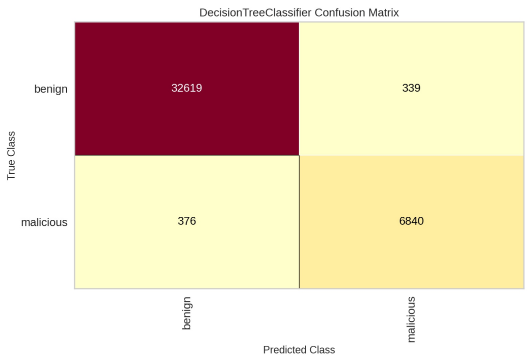
(a) DecisionTree SMOTE Classification Report (Process Features)



(b) DecisionTree SMOTE Confusion Matrix (Process Features)

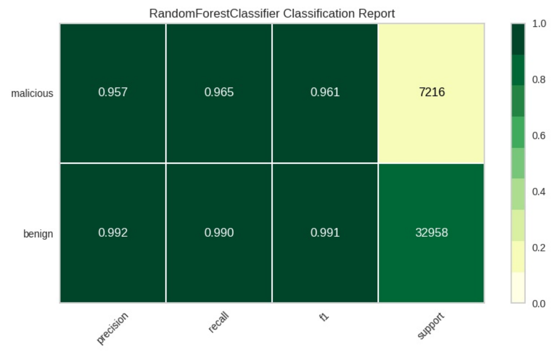


(c) DecisionTree Resampled Classification Report (Process Features)



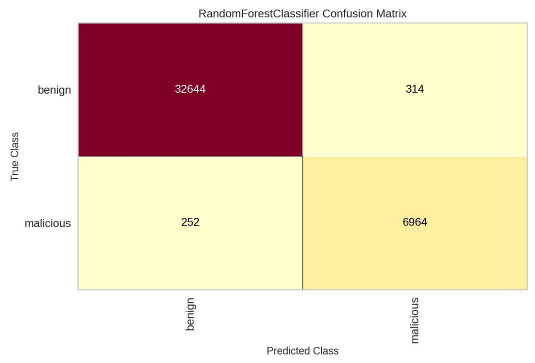
(d) DecisionTree Resampled Confusion Matrix (Process Features)

Figure 5.7 DecisionTree Classification Report and Confusion Matrix (Process Features)



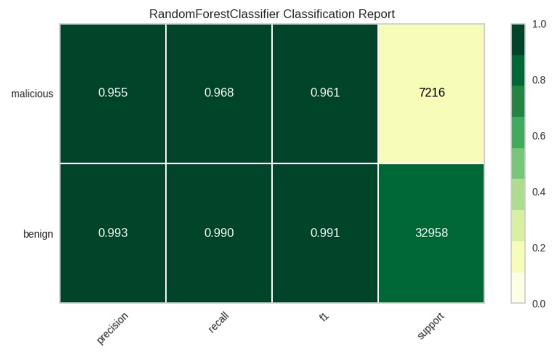
(a) RandomForest SMOTE Classification Report

(Process Features)



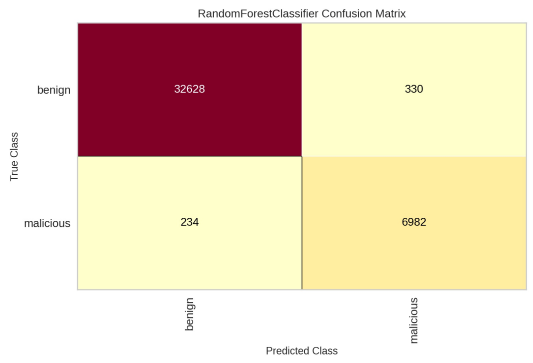
(b) RandomForest SMOTE Confusion Matrix

(Process Features)



(c) RandomForest Resampled Classification Report

(Process Features)

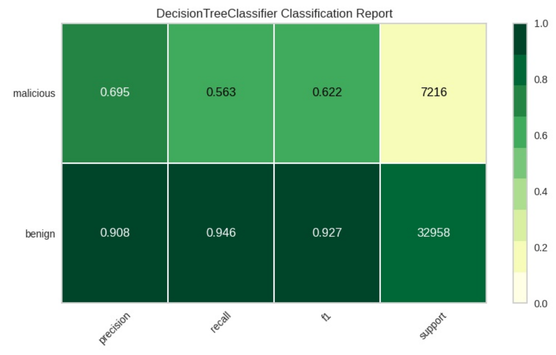


(d) RandomForest Resampled Confusion Matrix

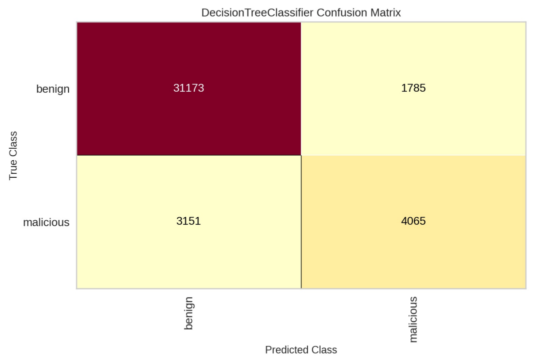
(Process Features)

Figure 5.8 RandomForest Classification Report and Confusion Matrix

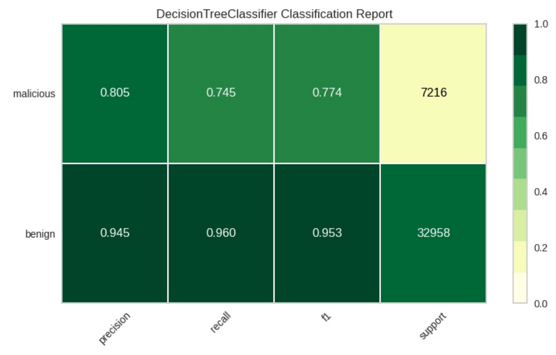
(Process Features)



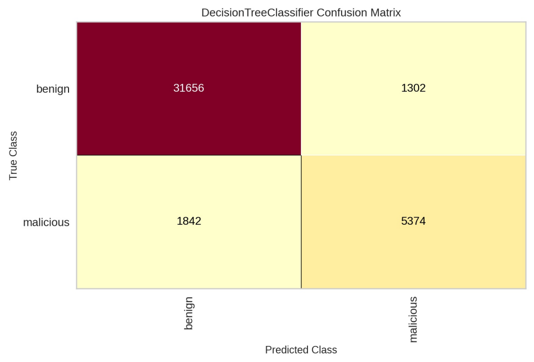
(a) DecisionTree SMOTE Classification Report (Path Features)



(b) DecisionTree SMOTE Confusion Matrix (Path Features)

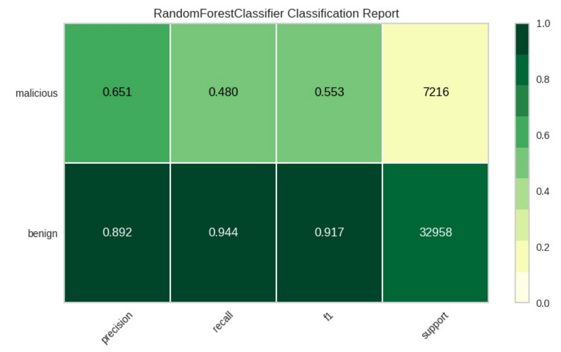


(c) DecisionTree Resampled Classification Report (Path Features)

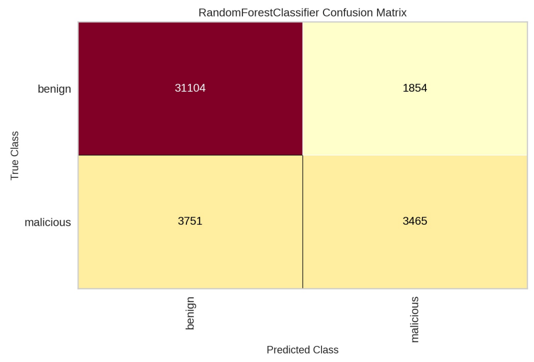


(d) DecisionTree Resampled Confusion Matrix (Path Features)

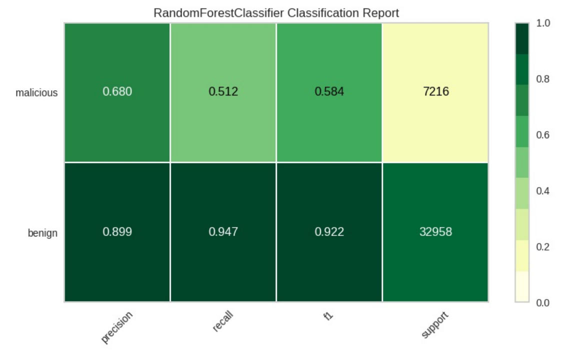
Figure 5.9 DecisionTree Classification Report and Confusion Matrix (Path Features)



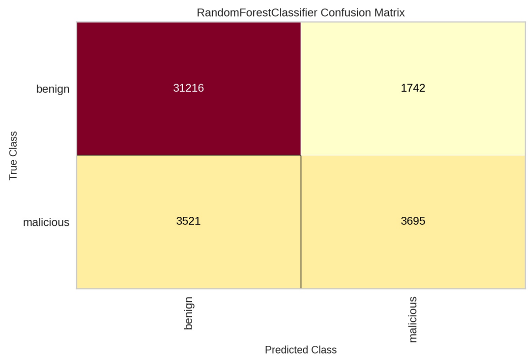
(a) RandomForest SMOTE Classification Report (Path Features)



(b) RandomForest SMOTE Confusion Matrix (Path Features)



(c) RandomForest Resampled Classification Report (Path Features)



(d) RandomForest Resampled Confusion Matrix (Path Features)

Figure 5.10 RandomForest Classification Report and Confusion Matrix (Path Features)

CHAPTER VI

DISCUSSION

Based on the results generated, the research questions outlined in Chapter I are addressed.

6.1 Which memory objects are related to processes in random access memory?

From the object mapping created at the start of this work, it can be seen that many relationships can be established from an _EProcess structure in Windows memory. As expected, objects such as DLLs, network connections, mutexes, etc., are connected to processes. A complete list of how processes interact with objects in memory is in Appendix A.

6.2 What are common memory object relationships that malware tends to manipulate and/or establish in memory?

From the features selected by our top machine learning models, one of the object relationships malware utilizes is centered around handles to other processes in memory. This finding coincides with our expectations. Malicious processes often spawn additional processes and set up handles to perform actions such as DLL injection. Also, the combination of the processes being a child of a malicious process and then the parent having a handle to that child was prevalent. Another relationship that frequently occurred was malicious processes containing privileges that were present but not enabled by default which is a major red flag. From our models, the privilege that seems to show the most significant impact on influencing a prediction was SeDebugPrivilege.

When a process has this privilege, it can inject code into a process. It also allows the ability to grant administrative capabilities to the current process or any others.

6.3 Do malware samples that have multiple processes have distinct path characteristics?

To answer this question, the highest performing model that only used normalized paths was reviewed: DecisionTree Resampled. This model used ninety-two normalized path features to make its predictions. To gain insight to answer this question, we used the feature importance attribute of the model to determine what was important. A feature importance chart is provided with the top thirty normalized path features from the model in Figure 6.1. nineteen of the top thirty features include a parent-child relationship in the paths selected. Twenty of the thirty features used a path that referenced a handle to another process. There could be a connection between this and the strength of the feature process_handle_count (See Figure 5.5 a) from the process-specific features which is used to count how many handles to other processes a target process has.

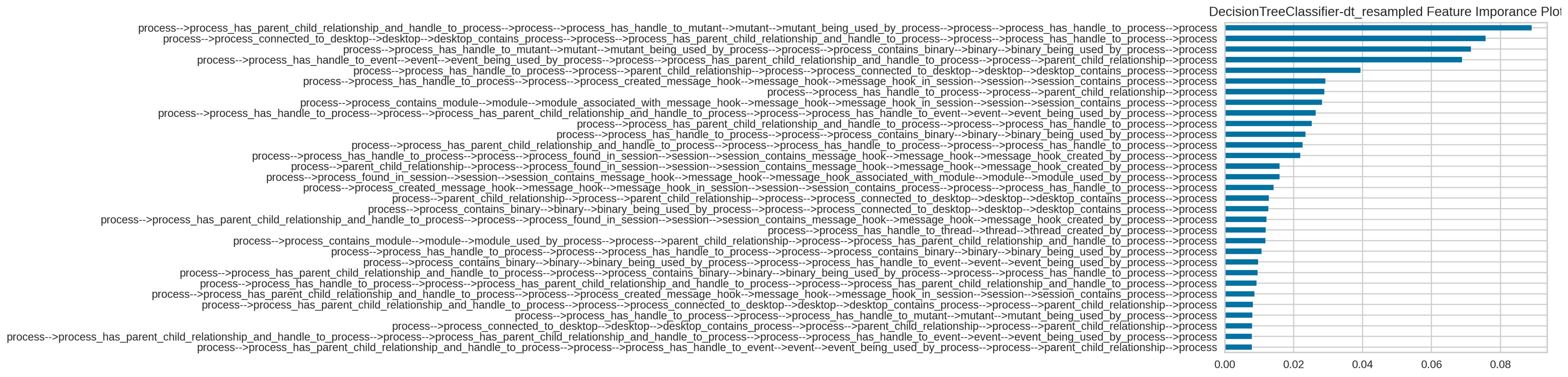


Figure 6.1 DecisionTree Feature Importance for Resampled Data: (Normalized Path Features)

6.4 Do benign and malicious process paths have distinguishing characteristics?

As the normalized path models were studied, there were no immediate indicators that individual features could provide clear delineations between benign and malicious processes. To check this, we used the groupby function in the Pandas [76] data analysis library to group the results of the top 10 path features based on if a process was benign or malicious. In reviewing this data, both malicious and benign processes included the various normalized paths. Therefore, it is thought that the combination of the normalized paths is what serves as the predictive factor classifying the processes when only using normalized path features.

6.5 Which features best support the goal of maximizing accuracy while minimizing false-negative rate in identifying malicious processes?

From the results, a combination of process and normalized path features performed the best overall. When looking at the results, the process features played a significant role in producing the highest performance by the models. Using normalized path features alone provided a moderate performance by the best models but suffered in their ability to identify malicious samples with a recall score of about 75%. For each of the combinations of features (combined, process-only, and normalized_path-only), each of the highest performing models beat the ZeroR baseline specified in Section **??**. Comparisons of the best performing models versus the ZeroR baseline on the test dataset are presented in Tables 6.1 and 6.2. Since the majority class in our dataset is benign, then for ZeroR all samples will be predicted as benign. This explains why the precision, recall, and F1-score for the malicious samples are zero because all predictions were incorrect for the malicious samples.

In the tables, the precision, recall, f1-score, and support are provided for both the benign and the malicious classes. The equations for these metrics are outlined in Section 5.4.4. Also, in the table, the weighted average for each of these metrics is provided. The equation calculating the weighted average is shown below in Equation 6.1.

$$weighted\_average = (support_b/ts) * score_b + (support_m/ts) * score_m \tag{6.1}$$

Where:

- *b* stands for benign
- *m* stands for malicious
- *ts* stands for total samples

Finally, the accuracy is defined in Equation 6.2

$$accuracy = (tp + tn)/(ts) \tag{6.2}$$

Where:

- *tp* stands true positives
- *tn* stands for true negatives
- *ts* stands for total samples

The RandomForest Resampled models performed the strongest, with the recall score reaching 97% for malicious samples. The high recall score is significant because classifying a malicious sample as benign would cause major issues in the real world. Thus, showing a high recall displays that the model is effective at identifying malicious samples. Although the DecisionTree classifier had a weaker performance than the other RandomForest models, it still performs stronger in every metric than the ZeroR method.

Table 6.1 Model Comparison vs. ZeroR Baseline

	Precision	Recall	F1-score	Support
Benign	0.82	1.00	0.90	32958
Malicious	0.00	0.00	0.00	7216
Weighted Avg	0.67	0.82	0.74	40174
Accuracy	0.82			

(a) ZeroR Baseline Metrics

	Precision	Recall	F1-score	Support
Benign	0.99	0.99	0.99	32958
Malicious	0.96	0.97	0.96	7216
Weighted Avg	0.99	0.99	0.99	40174
Accuracy	0.99			

(b) Combined Features: RandomForest Resampled

Table 6.2 Model Comparison vs. ZeroR Baseline cont.

	Precision	Recall	F1-score	Support
Benign	0.99	0.99	0.99	32958
Malicious	0.96	0.97	0.96	7216
Weighted Avg	0.99	0.99	0.99	40174
Accuracy	0.99			

(a) Process Features: RandomForest Resampled

	Precision	Recall	F1-score	Support
Benign	0.94	0.96	0.95	32958
Malicious	0.80	0.74	0.77	7216
Weighted Avg	0.92	0.92	0.92	40174
Accuracy	0.92			

(b) Normalized Path Features: DecisionTree Resampled

CHAPTER VII

CONCLUSION

In this work, it was proven that graph topology attributes and object characteristics from operating system memory images are predictive factors for identifying malicious processes. In this section, we provide a full summary of the work, a description of the contributions, and a discussion of how the research can be expanded in the future.

7.1 Summary

Our work started by identifying how objects interact with one another in memory during normal circumstances. These object interactions have been highlighted in Appendix A. Next, research on how malware abuses relationships was performed, and the Object Association Extractor (OAE) was developed. OAE takes a RAM image as input and generates a memory object graph containing objects that are nodes and edges that describe the relationship between nodes. OAE was used to extract data from 13,822 memory images that contained at least two active malicious processes. These memory images were generated using a dynamic malware analysis system called Cuckoo Sandbox [6] and the infrastructure was designed here at MSU. OAE was also used to extract benign process information from ninety-one memory images that ran software from the Chocolatey [62] package management software. In total, this generated 267,824 processes (48,108 malicious and 219,716 benign).

From this data, two sources of machine learning features were created. First, we extracted information associated with each individual _EProcess structure to develop process-specific features. The raw process data is discussed in Section 3.4.3 and the transformed features are discussed in Section 5.3. The next type of feature created utilized Inverse Document Frequency (IDF) to study the normalized paths between malicious processes in our memory object graphs. The normalized paths between malicious processes were collected, and IDF was used to assign weights to uncommon paths of interest. The reasoning was that paths that are not common and used by malware would occur less frequently. The IDF algorithm is focused on ignoring common occurrences in a dataset and highlighting the rare paths.

A training, validation, and test set was created from the data, and the processes were stratified to ensure that all three sets received similar proportions of malicious, benign, and chocolatey samples. To assist with the imbalance of 3rd-party benign processes, the oversampling techniques Random Resampling and Synthetic Minority Oversampling Technique (SMOTE) were applied. This is discussed in Section 5.4.3. From the training set, four machine learning models were trained based on the Resampled and SMOTE training sets, and the top features for each model were selected using Recursive Feature Elimination based on their ability to predict the validation set. The top two types of models that performed best were the Decision Tree and Random Forest models. This is described in Section 5.4.4. The final performance of the models was outlined in Section 5.5. The performance of these models on the test set determined that topology attributes and object characteristics are predictive factors for identifying malicious processes. The results were compared to the ZeroR baseline for a combined feature set, process-specific features, and normalized path features. Each group of features outperformed the ZeroR baseline as discussed in Section 6.5. The process features turned out to be very predictive of malicious code, with the best classifier reaching a recall of almost 97%. The normalized path features had a recall score of about 75% for malicious processes.

The success of this work will be beneficial for malware analysts and cyber incident responders. This research can be used to identify malicious processes and identify objects that are related to a process of interest. This type of triage tool could be helpful for the tasks these types of professionals would need to perform. What often amounts to "looking for a needle in a haystack" is quite time-intensive, and anything that could speed up this process would be valuable.

7.2 Contributions

This section details the specific contributions of this work.

7.2.1 Memory object mapping

In conducting this study, the target was to first identify what normal activity between memory objects looked like. This is important because anomalies are challenging to pinpoint without knowing what standard behavior entails. We targeted objects of forensic interest as described in work by Ligh et. al [56] and then created a mapping of the types of relationships that exist. From that work, we then reviewed the manner in which malware tends to abuse these relationships and identified signs that malicious activity was occurring. The center of this study was on process objects that interact with almost all other objects in memory. This mapping was used to create the logic behind OAE, which is mentioned in the next section. Having this type of resource is helpful to analysts by providing a guide for the types of object interactions that are possible. This study can serve as the basis for additional memory forensics research to study new ways malware abuses or alters object relationships. The mapping will also help identify similar objects relationship across different operating systems.

7.2.2 Object Association Extractor

This work also produced the Object Association Extractor (OAE). OAE is novel in that it extracts characteristics of objects in memory and creates a graph where objects are nodes and the relationships between objects are edges. The edges also contain information about the types of relationships that exist between two nodes. OAE leverages Rekall, the memory forensics framework, to extract memory object information and the NetworkX Python package to generate the graphs. This tool was used for creating all the datasets in this work that generated the features for predicting malicious processes. Because of OAE, we were able to determine that process-specific and normalized path features are predictive of malicious processes.

7.2.3 Feature set development

Features were created that can be utilized by machine learning classifiers to help identify malicious processes. These features are based on process characteristics and normalized path information from our corpus of memory images. The process features were developed by studying information associated with the base data structure for each process, the _EProcess structure. The normalized path features were developed by using Inverse Document Frequency to identify uncommon paths between malicious processes.

7.2.4 Trained machine learning classifiers

Trained machine learning classifiers were developed to distinguish between benign and malicious processes. Other work such as Mosli et. al. [42] has used process handles as features for machine learning classifiers, but this work is the first to use the combination of both process and normalized path features. This capability is beneficial to analysts that need to identify areas of interest for an investigation. Manually conducting memory forensics is time-intensive and requires a significant amount of prior knowledge. Utilizing OAE and these trained machine learning models reduces the burden of knowing indicators of compromise, understanding how objects should interact in memory, or knowing about specific heuristics that could indicate malicious activity. Also, automatically pinpointing objects of interest will help direct any necessary manual analysis in the proper direction.

7.3 Future Work

Several areas can be investigated to expand on this current work and explore other research topics.

7.3.1 Alternative Approaches

Additional work can be conducted to include benign processes from various sources. In most malware research, the benign samples used are constrained to using system processes on a standard Windows operating system. Our process datasets that contain benign samples from the Chocolatey package management platform could be strengthened by including even more 3rd-party processes. This could be achieved by adding additional Chocolatey samples or by finding other sources of new benign processes. One idea is to find a method to utilize the NIST National Software

Reference Library [83]. It is designed to be used by law enforcement, government, and industry organizations to review files quickly to identify if files are potential evidence or standard software files. Cryptographic hashes are provided for each of the files in their datasets, and searching for online repositories with portable executable (PE) files with the same hash would be another source of benign software. With different benign processes, more performance tests on the process-specific features could be conducted. The goal for adding more benign processes to the training, validation, and test sets would be to bolster the performance of our best models to ensure they continue to generalize well on processes not seen before.

Recursive Feature Elimination (RFE) was used to select each model's features in the current machine learning workflow. Other feature selection techniques such as Forward Stepwise Feature Selection [84] and SelectFromModel [85] could be used. With RFE we started with a full complement of features and then removed features in each iteration. With Forward Stepwise Feature selection, the method would be conducted in reverse, starting with no features and incrementing the number for each iteration. With SelectFromModel, each classifier from Scikit-learn would use an internal feature importance metric to help select the optimal features.

Once features have been selected, the hyper-parameter tuning process using RandomizedSearch and GridSearch could be included to optimize performance. Due to time constraints and the excellent performance achieved by combining process-specific and normalized path features, hyper-parameter tuning was not performed in this work.

7.3.2 Community Detection

The study of graph theory could be used in the future, with an emphasis on community detection. Community detection algorithms are used to find highly connected groups or communities within a graph automatically [86]. Moving forward, we could identify malicious or benign processes based on a community of objects and their underlying memory relationships. The subject of community detection is mature and has a number of algorithms for determining communities [87] [88] [89]. The focus could be using communities that contain processes and the composition of these communities to distinguish between benign and malicious processes.

7.3.3 Kernel Objects

The focus of this work was on identifying malicious processes. In the future, this could be expanded to identify other objects based on their relationships. Those that could be of particular interest are kernel modules and drivers. Targeting these types of objects would help in detecting more advanced malware such as rootkits. OAE would be updated to identify, connect, and extract objects to support this type of classification.

7.3.4 Additional Operating Systems

This work developed a proof of concept for identifying malicious processes in RAM, and it was executed using the Windows 7 operating system. Moving forward, the same principles will be applied to Windows 10 to show similar results. Further work could be developed to apply the principles identified for the Windows operating systems to Linux and possibly Android OSes in the future.

7.3.5 Live Acquisition and Predictions

In its current form, OAE is designed to take memory snapshots as input and then generate graph data for later predictions. Work could be performed to use live acquisition techniques and identify malicious processes in real-time. The primary challenge would be finding RAM acquisition tools that do not fall victim to page smearing as mentioned by Case and Richard III [14]. Page smearing occurs when memory being acquired changes during the acquisition phase. A study of state of the art in RAM acquisition tools would need to be conducted to determine how to integrate these tools with OAE.

7.4 Closing and Acknowledgement of Support

Malware authors continue to change their methods, and tools must be leveraged to identify and thwart their attacks. Memory forensics is a quality technique for identifying malware primarily because malware samples have to exist in memory to perform their actions. We devised a novel way of leveraging memory forensics and machine learning by using object characteristics in memory and discovering patterns in their relationships for detecting malicious processes. The results of this work are helpful to forensic analysts that have to identify malicious activity and often have to rely upon a vast amount of background knowledge and manual analysis. This research will support current investigations by making them more effective and lays the groundwork for the development of new tools in the future.

This work was supported in part by the following funding sources:

- A Supervised Learning Approach for Supplemental Malware Identification in Memory Images, DHS, Grant #: HSHQDC16CB0030

- NCAE-C Cyber Curriculum and Research 2020 Program, NSA, Grant #: H98230-20-1-0386

REFERENCES

[1] N. Idika and A. Mathur, "A Survey of Malware Detection Techniques," Purdue University, Tech. Rep., 2007. [Online]. Available: http://profsandhu.com/cs5323_s17/im_2007.pdf

[2] D. Uppal, V. Mehra, and V. Verma, "Basic survey on Malware Analysis, Tools and Techniques," *International Journal on Computational Science & Applications*, vol. 4, no. 1, pp. 103–112, 2014. [Online]. Available: https://pdfs.semanticscholar.org/b254/79a230816e8566df0937d9ff9265754f826d.pdfhttp://www.airccse.org/journal/ijcsa/papers/4114ijcsa10.pdf

[3] A. Walters, "FATKit: Detecting Malicious Library Injection and Upping the âAntiâ," *4TF Research Laboratories*, pp. 1–12, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.2189{&}rep=rep1{&}type=pdfhttp://www.4tphi.net/fatkit/papers/fatkit{_}dll{_}rc3.pdf

[4] D. Distler, "Malware Analysis: An Introduction Malware Analysis: An Introduction Malware Analysis: An Introduction GSEC Gold Certification Malware Analysis: An Introduction," *SANS Institute Infosec Reading Room*, 2007. [Online]. Available: https://www.sans.org/reading-room/whitepapers/malicious/malware-analysis-introduction-2103

[5] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, 2017. [Online]. Available: https://www.researchgate.net/publication/288905288

[6] C. Foundation, "Automated Malware Analysis - Cuckoo Sandbox," 2014. [Online]. Available: http://cuckoosandbox.org/

[7] M. Egele, T. Scholte, E. Kirda, and S. Barbara, "A survey on automated dynamic malware analysis techniques and tools," *ACM Computing Surveys,*, vol. V, no. 2, pp. 1–49, feb 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2089125.2089126

[8] Mark Wade, "Memory Forensics: Where to Start," p. 1, 2011. [Online]. Available: https://www.forensicmag.com/article/2011/06/memory-forensics-where-starthttp://www.forensicmag.com/article/2011/06/memory-forensics-where-start

[9] A. Walters and N. L. Petroni, "Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process," pp. 1–18, 2007.

[10] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, no. 4, pp. 197–210, 2006. [Online]. Available: https://ac.els-cdn.com/S1742287606001228/1-s2.0-S1742287606001228-main.pdf?{_}tid=7b18539b-2f30-4d25-9993-d5c35818e9cf{&}acdnat=1523892729{_}42d53af466d37bf986ca746197e53e5c

[11] B. Dolan-Gavitt, "The VAD Tree: A Process-Eye View of Physical Memory," *Digital Investigation*, 2007. [Online]. Available: http:/dfrws.org

[12] A. Schuster, "Searching for processes and threads in Microsoft Windows memory dumps," *Digital Investigation*, vol. 3, no. SUPPL., pp. 10–16, sep 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287606000727

[13] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," pp. 3–22, jul 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287611000508

[14] A. Case and G. G. Richard, "Memory forensics: The path forward," *Digital Investigation*, vol. 20, pp. 23–33, 2017. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1742287616301529

[15] K. Amari, K. A. Mil, and C. Cid, "Techniques and Tools for Recovering and Analyzing Data from Volatile Memory Techniques and Tools for Recovering and Analyzing Data from Volatile Memory GCFA Gold Certification Techniques and Tools for Recovering and Analyzing Data from Volatile Memory 3," *SANS Institute*, p. 61, 2009. [Online]. Available: https://www.sans.org/reading-room/whitepapers/forensics/techniques-tools-recovering-analyzing-data-volatile-memory-33049

[16] L. Cai, J. Sha, and W. Qian, "Study on forensic analysis of physical memory," *Proc. 2nd International Symposium on*, 2013. [Online]. Available: http://www.atlantis-press.com/php/download{_}paper.php?id=10172

[17] SANS, "Memory Forensics Training Course | Computer Memory Forensics | SANS FOR526." [Online]. Available: https://www.sans.org/course/memory-forensics-in-depth

[18] C. L. Brooks, *CHFI Computer Hacking Forensic Investigator Certification All-in-One Exam Guide*, 1st ed. McGraw-Hill Education Group, 2014.

[19] K. Hausknecht, D. Foit, and J. Burić, "RAM data significance in digital forensics," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015 - Proceedings*. IEEE, may 2015, pp. 1372–1375. [Online]. Available: http://ieeexplore.ieee.org/document/7160488/

[20] V. Foundation, "The Volatility Foundation - Open Source Memory Forensics." [Online]. Available: http://www.volatilityfoundation.org/

[21] Google Inc., "Rekall Memory Forensic Framework," 2015. [Online]. Available: http://www.rekall-forensic.com/

[22] J. Kornblum, "Exploiting the Rootkit Paradox with Windows Memory Analysis," *International Journal of Digital Evidence*, vol. 5, no. 1, pp. 1–5, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.82.612{&}rep=rep1{&}type=pdf

[23] N. Hubballi, S. Biswas, and S. Nandi, "Sequencegram: n-gram modeling of system calls for program based anomaly detection," in *2011 3rd International Conference on Communication Systems and Networks, COMSNETS 2011*, 2011.

[24] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[25] A. Rajaraman and J. D. Ullman, *Data Mining*. Cambridge University Press, 2011, p. 1â17.

[26] S. Thomas, K. Sherly, and S. Dija, "Extraction of memory forensic artifacts from windows 7 ram image," *Information & Communication*, 2013. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6558230/

[27] J. Scott, "Hunting Malware with Memory Analysis | Solutionary Blog," 2012. [Online]. Available: https://technical.nttsecurity.com/post/102egyy/hunting-malware-with-memory-analysishttps://www.solutionary.com/resource-center/blog/2012/12/hunting-malware-with-memory-analysis/

[28] A. Bridge, "Thanks for the Memories: Identifying Malware from a Memory Capture | Context Information Security," 2015. [Online]. Available: https://www.contextis.com/blog/thanks-for-the-memories-identifying-malware-from-a-memory-capture

[29] M. Ka, "Finding Advanced Malware Using Volatility," 2014. [Online]. Available: https://eforensicsmag.com/finding-advanced-malware-using-volatility/

[30] A. Case, R. D. Maggio, M. Firoz-Ul-Amin, M. M. Jalalzai, A. Ali-Gombe, M. Sun, and G. G. Richard, "Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics," *Computers and Security*, vol. 96, p. 101872, sep 2020.

[31] VirusTotal, "VirusTotal." [Online]. Available: https://www.virustotal.com/{#}/home/upload

[32] O. Sukwong, H. Kim, and J. Hoe, "Commercial Antivirus Software Effectiveness: An Empirical Study," *Computer*, vol. 44, no. 3, pp. 63–70, mar 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5506074/

[33] I. Rijnetu, "These Campaigns Explain Why AV Detection for New Malware Remains Low," 2017. [Online]. Available: https://heimdalsecurity.com/blog/campaigns-av-detection-new-malware-low/

[34] A. Algaith, I. Gashi, B. Sobesto, M. Cukier, S. Haxhijaha, and G. Bajrami, "Comparing Detection Capabilities of AntiVirus Products: An Empirical Study with Different Versions of Products from the Same Vendors," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, jun 2016, pp. 48–53. [Online]. Available: http://ieeexplore.ieee.org/document/7575349/

[35] P. Bajpai and R. Enbody, "Memory Forensics against Ransomware," *International Conference on Cyber Security and Protection of Digital Services, Cyber Security 2020*, jun 2020.

[36] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated malware detection using artifacts in forensic memory images," in *2016 IEEE Symposium on Technologies for Homeland Security (HST)*. IEEE, may 2016, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7568881/

[37] VirusShare, "VirusShare.com." [Online]. Available: http://virusshare.com/

[38] D. Glendowne, C. Miller, W. McGrew, and D. Dampier, "Characteristics of malicious DLLS in windows memory," *IFIP Advances in Information and Communication Technology*, vol. 462, pp. 149–161, 2015.

[39] A. H. Lashkari, B. Li, T. L. Carrier, and G. Kaur, "VolMemLyzer: Volatile Memory Analyzer for Malware Classification using Feature Engineering." Institute of Electrical and Electronics Engineers (IEEE), jun 2021, pp. 1–8.

[40] A. S. Bozkir, E. Tahillioglu, M. Aydos, and I. Kara, "Catch them alive: A malware detection approach through memory forensics, manifold learning and computer vision," *Computers and Security*, vol. 103, p. 102166, apr 2021.

[41] "ProcDump - Windows Sysinternals | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/procdump

[42] R. Mosli, R. Li, B. Yuan, and Y. Pan, "A behavior-based approach for malware detection," in *IFIP Advances in Information and Communication Technology*, vol. 511. Springer, Cham, jan 2017, pp. 187–201. [Online]. Available: http://link.springer.com/10.1007/978-3-319-67208-3{_}11

[43] Y. Duan, X. Fu, B. Luo, Z. Wang, J. Shi, and X. Du, "Detective: Automatically identify and analyze malware processes in forensic scenarios via DLLs," in *IEEE International Conference on Communications*, vol. 2015-Septe, 2015, pp. 5691–5696.

[44] M. Aghaeikheirabady, S. M. R. Farshchi, and H. Shirazi, "A new approach to malware detection by comparative analysis of data structures in a memory image," in *2014 International Congress on Technology, Communication and Knowledge, ICTCK 2014*. IEEE, nov 2014, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/7033519/

[45] VirusSign, "VirusSign - Malware Research and Data Center, Virus Free Downloads," 2017. [Online]. Available: http://www.virussign.com/http://www.virussign.com/contact.html

[46] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, "BotViz: A memory forensic-based botnet detection and visualization approach," in *2017 International Carnahan Conference on Security Technology (ICCST)*. IEEE, oct 2017, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/8167804/

[47] A. Srivastava and J. H. Jones, "Detecting code injection by cross-validating stack and VAD information in windows physical memory," in *2017 IEEE Conference on Open Systems, ICOS 2017*, vol. 2017-Janua. IEEE, nov 2017, pp. 83–89. [Online]. Available: http://ieeexplore.ieee.org/document/8280279/

[48] A. Cohen and N. Nissim, "Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory," *Expert Systems with Applications*, vol. 102, pp. 158–178, jul 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418301283http://linkinghub.elsevier.com/retrieve/pii/S0957417418301283

[49] M. Ajay Kumara and C. Jaidhar, "Leveraging virtual machine introspection with memory forensics to detect and characterize unknown malware using machine learning techniques at hypervisor," *Digital Investigation*, vol. 23, pp. 99–123, dec 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287617303328

[50] C. Miller, D. Glendowne, H. Cook, D. Thomas, C. Lanclos, and P. Pape, "Insights gained from constructing a large scale dynamic analysis platform," *Digital Investigation*, vol. 22, no. S, pp. S48–S56, aug 2017. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1742287617301949

[51] D. Glendowne, "Automating malware detection in Windows memory images using machine learning," *ProQuest Dissertations and Theses, Mississippi State University, 2015, p. 144. Dissertations Theses @ Mississippi State University*, 2015. [Online]. Available: https://search.proquest.com/docview/1679463958?pq-origsite=gscholar

[52] QEMU, "QEMU." [Online]. Available: https://www.qemu.org/

[53] T. Hungenberg and M. Eckert, "Inetsim: Internet services simulation suite." [Online]. Available: https://www.inetsim.org/

[54] The Apache Software Foundation, "Apache CouchDB," p. 495, 2018. [Online]. Available: https://couchdb.apache.org/http://couchdb.apache.org/

[55] J. O. Notation, "Introducing JSON." [Online]. Available: https://www.json.org/json-en.html

[56] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. John Wiley & Sons Inc., 2014, vol. 1.

[57] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows internals. Part 1.* Microsoft Press, 2012.

[58] N. Developers, “NetworkX â NetworkX.” [Online]. Available: https://networkx.github.io/

[59] EEE and T. O. Group, “file.” [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/file.html

[60] E. Carrera, “GitHub - erocarrera/pefile: pefile is a Python module to read and work with PE (Portable Executable) files.” [Online]. Available: https://github.com/erocarrera/pefile

[61] S. Sebastián and J. Caballero, “AVclass2: Massive Malware Tag Extraction from AV Labels,” in *ACM International Conference Proceeding Series*. Association for Computing Machinery, dec 2020, pp. 42–53. [Online]. Available: https://doi.org/10.1145/3427228.3427261

[62] Chocolatey Software Inc., “Chocolatey - The package manager for Windows.” [Online]. Available: https://chocolatey.org/https://chocolatey.org

[63] Scikit-learn, “sklearn.feature_extraction.text.TfidfVectorizer â scikit-learn 0.24.2 documentation,” 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature{_}extraction.text.TfidfVectorizer.html

[64] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, “Gene selection for cancer classification using support vector machines,” *Machine Learning*, vol. 46, no. 1-3, pp. 389–422, 2002. [Online]. Available: https://link.springer.com/article/10.1023/A:1012487302797

[65] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in python,” *J. Mach. Learn. Res.*, vol. 12, no. null, p. 2825â2830, Nov. 2011.

[66] Kaggle, “Kaggle: Your Machine Learning and Data Science Community,” 2019. [Online]. Available: https://www.kaggle.com/

[67] D. Kirat, G. Vigna, and C. Kruegel, “BareCloud: Bare-metal Analysis-based Evasive Malware Detection,” *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 287–301, 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat

[68] A. Ortega, “Pafish (Paranoid Fish).” [Online]. Available: https://github.com/a0rtega/pafish

[69] D. Keragala, “Detecting Malware and Sandbox Evasion Techniques,” 2016. [Online]. Available: https://www.sans.org/reading-room/whitepapers/forensics/detecting-malware-sandbox-evasion-techniques-36667

[70] scikit learn, “sklearn.model_selection.KFold â scikit-learn 0.24.2 documentation.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

[71] R. E. Bellman, *Adaptive Control Processes*. Princeton University Press., 1961.

[72] F. Provost, D. Jensen, and T. Oates, "Efficient progressive sampling," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '99*. New York, New York, USA: ACM Press, 1999, pp. 23–32. [Online]. Available: http://portal.acm.org/citation.cfm?doid=312129.312188

[73] VirusShare, "VirusShare-Research." [Online]. Available: https://virusshare.com/research.4n6

[74] Project Jupyter, "Project Jupyter | Home," 2019. [Online]. Available: https://jupyter.org/index.htmlhttps://jupyter.org/

[75] HPCng, "Singularity." [Online]. Available: https://singularity.hpcng.org/

[76] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[77] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, sep 2009.

[78] P. Branco, L. Torgo, and R. Ribeiro, "A survey of predictive modelling under imbalanced distributions," 2015.

[79] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[80] "nteract/papermill." [Online]. Available: https://github.com/nteract/papermill{#}https://github.com/nteract/papermill

[81] A. S. Foundation, "Apache Sparkâ¢ - Unified Analytics Engine for Big Data," 2018. [Online]. Available: https://spark.apache.org/

[82] "GitHub - joblib/joblib-spark: Joblib Apache Spark Backend." [Online]. Available: https://github.com/joblib/joblib-spark

[83] D. White, "National Software Reference Library," pp. 1–17, 2004. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl

[84] "sklearn.feature_selection.SequentialFeatureSelector â scikit-learn 0.24.2 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature{_}selection.SequentialFeatureSelector.html{#}sklearn.feature{_}selection.SequentialFeatureSelector

[85] "sklearn.feature_selection.SelectFromModel â scikit-learn 0.24.2 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature{_}selection.SelectFromModel.html{#}sklearn.feature{_}selection.SelectFromModel

[86] S. Fortunato and D. Hric, “Community detection in networks: A user guide,” *Physics Reports*, vol. 659, pp. 1–44, jul 2016. [Online]. Available: http://arxiv.org/abs/1608.00163http://dx.doi.org/10.1016/j.physrep.2016.09.002

[87] J. Andrews, “Community Detection In Large Networks,” Ph.D. dissertation, Cornell University, aug 2012. [Online]. Available: https://ecommons.cornell.edu/handle/1813/31046

[88] S. Fortunato, “Community detection in graphs,” pp. 75–174, feb 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0370157309002841

[89] F. D. Malliaros and M. Vazirgiannis, “Clustering and community detection in directed networks: A survey,” *Physics Reports*, vol. 533, no. 4, pp. 95–142, dec 2013. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0370157313002822

APPENDIX A

MEMORY OBJECT MAPPING DEFINITIONS AND RELATIONSHIPS

(a) **Process Definition** - A process is an instance of a program executing in memory. Each process starts from a binary that gives instructions for what the program should do. The operating system takes responsibility for creating, suspending, and terminating processes running in memory. When a process is created it is given a process id, a set of attributes, and its own address space. The address space is a container for other objects such as shared libraries (DLLs), dynamic data from the user, and its runtime stack. Within memory, processes are represented as _EPROCESS structures.

(b) **Binary Definition** - A binary is the Portable Executable (PE) file that, when run, creates a process. Once loaded into memory the binary is represented as an _EPROCESS structure.

(c) **Thread Definition** - Threads are represented as _ETHREAD structures and have the responsibility of executing tasks for a process. At least one thread must exist for a process. Multiple threads can be executing within a single process with each performing different tasks. Threads share all of the same code, data, address space, and operating system resources of a process.

(d) **DLL Definition** - Dynamic linked libraries (DLLs) are shared libraries that are loaded into the address space of a process. This loading can be done intentionally or forcefully. DLLs are designed to be shared between processes and contain code and resources for the processes to use. Since DLLs are created to run inside of a host process, they are often given access to process resources: threads, handles, and the process memory space. DLLs are represented by _LDR_DATA_TABLE_ENTRY structures.

(e) **Session Definition** - A session is responsible for storing information about a user logging into a system as well as keeping track of GUI objects created and drivers loaded on a system. Sessions have a unique ID that can be tied to a user and activities that occur within a session can be attributed back to that user. A session is represented as a _MM_SESSION_SPACE structure.

(f) **Mutant Definition** - A mutant, or mutex, is a kernel object that is represented as a _KMUTANT structure in memory. Their primary responsibility is for interprocess synchronization. Malware uses these objects to mark their presence on a system and prevent other variants of the malware sample to infect the system again.

(g) **Token Definition** - Tokens keep track of the security context for a specific process. These objects identify the types of actions that a given process is allowed to perform. Tokens are represented in memory by _TOKEN structures. Tokens also include privileges, _SEP_TOKEN_PRIVILEGES structures, which set permissions for performing tasks. For example, a common privilege used to write to other processâs private memory space is SeDebugPrivilege. It must be enabled to perform this action.

(h) **Window Station Definition** - Window stations are containers within a session that act as security boundaries for processes and desktops. There can be multiple window stations within a session, but there should only be one interactive window station which is capable of receiving user input. The interactive window station for a session will be âWinSta0â. The main structure for a window station is tagWINDOWSTATION.

(i) **Desktops Definition** - Desktops are containers that manage application windows and user interface objects. Malware can use these types of objects to spawn additional desktops without a userâs knowledge or lock a user out of their current desktop, which is the case with some ransomware. Desktops use the tagDESKTOP structure.

(j) **Atoms Definition** - Atoms are strings that are shared between processes within the same session. An atom is stored inside of an atom table and uses the structure _RTL_ATOM_TABLE_ENTRY in memory. Atoms are of interest because many Windows API calls create atoms indirectly. Because of this some malware authors and security analysts are not aware of the information that is available by searching through atom tables. There are some malware authors that have begun to use these objects as a version of a mutex to avoid similar samples infecting a machine that is already compromised.

(k) **Message Hooks Definition** - Message hooks are objects that can be used to intercept window messages before they reach a specific window. Malware can use this ability to monitor and collect all keyboard-related messages. The structure for message hooks is tagHOOK.

(l) **Event Hook Definition** - Event hooks provide the ability to monitor for specific UI-related events. Example events could include when a command prompt is about to be started, a process attempting to use an audio driver, monitoring mouse input, etc. The structure for an event hook is tagEVENTHOOK.

(m) **VAD Description** - The Virtual Address Descriptor (VAD) for a process describes how memory segments are structured. These data structures are managed by the operating system and contain key information such as the names of memory-mapped files, the total number of pages in the region, and the initial protection for the memory region. The VAD objects are represented as _MMADDRESS_NODE structures.

(n) **Kernel Module Description** - These objects are created from KLDR_DATA_TABLE_ENTRY structures. Malicious kernel modules have a number of different options for evading detection such as altering metadata structures, hooking functions, monitoring information with callbacks, etc.

(o) **Driver Description** - Drivers are created from kernel modules and are based off of _DRIVER_OBJECT structures. These objects contain information about the kernel module that created it. These objects can be key to identifying a kernel module that is attempting to hide itself or alter other modules. Applications in Windows also communicate with drivers via I/O Request Packets (IRPs). The IRP structure has twenty-eight functions that allow for communication, but can be manipulated by other drivers.

(p) **Device Description** - Devices are created by driver objects and use the _DEVICE_OBJECT structure. Devices that attach to a driverâs stack have the ability to monitor and manipulate IRPs as they are being processed. The most common devices that malware will monitor are network, keyboard, and disk devices.

(q) **IRP Function Table Description** - I/O Request Packets (IRP) are the way that Windows communicate with drivers. An IRP function table is an array of twenty-eight function pointers for operations that a specific driver can do. Rootkits will manipulate these pointers so that they either point to malicious kernel modules directly or hook these functions so that a jump to malicious code is performed.

(r) **Callbacks Description** - Callbacks are routines that execute when a certain event occurs on a system. Malware that unlinks its kernel module to be stealthy makes it easy to spot malicious callbacks because the âModuleâ column in Rekall is displayed as UNKNOWN.

(s) **Registry Key Description** - The registry stores various configuration information about applications, users, etc., within the Windows operating system. Registry keys are based on _CMHIVE objects on disk. Most registry keys that are used frequently can be found in memory. This is dependent on how heavily used a system is and if information has been paged out of memory. These objects can be used to keep track of any information that malware may store for it to function properly or to remain present on a system even after shutdown.

(t) **Service Description** - Services are programs running at a higher privilege level than most programs on a system. In most cases they are non-interactive and run consistently in the background. Services are created from _SERVICE_RECORD structures. Malicious actors can often use services as a means of persistence by hijacking services or can hide its existence by altering metadata structures.

(u) **Timer Description** - Malware often uses kernel timer objects as a means to receive notifications when a certain amount of time has elapsed. These objects can also be automatically reset after it expires, which allows it to perform periodic monitoring. Timers are created from _KPCR structures.

(v) **SSDT Description** - The System Service Descriptor Table (SSDT) contains pointers to kernel mode functions. This table allows user level applications to access system services.

Each entry in the SSDT is represented by _SERVICE_DESCRIPTOR_ENTRY structures. Malware will attempt to replace pointers, insert hooks, or duplicate tables as methods for starting malicious code via SSDTs.

(w) **Network Connections** - Identifying network connections is key in discovering the exact intent for malware. Some attackers are looking to extract sensitive information out, while others are seeking to send commands to cause damage on a system that has been infected. When collecting these connections here are some of the structures used for analysis:

- _TCP_ENDPOINT
- _TCP_LISTENER
- _UDP_ENDPOINT

From these structures you can discern the IP address used and, depending on the protocol, the port utilized for communication.

A.1 Object Interactions

Processes and Binaries - Each process running on a system will be started from an origin binary PE file. A process is the representation of a binary executing in memory.

Processes and DLLs - Once a process is started, each will contain a number of DLLs that provide it with the functionality needed to perform its specific actions. Each DLL used is mapped into the virtual address space of a given process.

Processes and Devices - Another type of object that processes have access to are devices. A process can gain access to this object via a handle. The primary device a process needs access to for spawning network connections is \Device\Afd\Endpoint.

Devices and Network Connection - When a device creates a network connection, the type of object which is created is dependent on the protocol that is being used. In addition, the device used for network connections is \Device\Afd\Endpoint. An active handle to this device must be present for the duration of a connection.

Processes, Tokens, and Privileges - To manage what objects a process can access, create, delete, or modify, privileges and tokens are assigned. These components help to implement security mechanisms to manage who has access to what in an operating system and regulate the type of tasks that can be performed.

Tokens and Privileges - Tokens can include multiple tokens which enable a process to perform specific actions.

Processes and Mutants - Processes will often create mutant objects to support mutual exclusion when accessing shared resources. Malware uses mutants as a way to mark its presence on a system to prevent a similar sample from attempting to infect the system again.

Processes and Threads - A process, in order to be running, must have at least one active thread. Each thread will have a segment of process memory set aside for its runtime stack. Threads can execute any part of a process' code, even those being used by other threads.

Processes and VADs - Each process points to a VadRoot of a VAD tree. From here these memory regions can be analyzed to gain additional information about what has been occurring. One activity in particular that can be monitored is the practice of injecting DLLs into processes. One method that can be checked is by reviewing the type of VAD structure that is assigned for a memory region. If the memory region is created with an _MMVAD_SHORT then no PE files should be mapped into this area. If a PE file is found, then this is evidence that DLL injection has been performed.

Processes and Event Hooks - A process will use the SetWinEventHook to create a event hook. The attacker has the ability to set the range of events that can be hooked when created this object. Most attackers will set all possible events to ensure that a DLL is injected.

Processes and Messages Hooks - A process will use the SetWindowsHook function to create message hooks. An attacker could perform keylogging to monitor a victimâs communications by injecting a DLL into a remote process.

Processes and Sessions - Each process that is running on a system will be associated with a specific session. Sessions act as containers for processes and objects on a system. Since sessions can be associated with users on a system, looking at the relationships between sessions and processes can help point to the owning user account on a system.

Processes and Desktops - Processes are assigned to specific desktops, which are containers for user objects. One of the primary ways that malware can abuse desktops is by launching applications in alternate desktops to avoid being noticed by the victim.

Processes and Services - Services can use processes in a standalone or a shared perspective to perform their actions. Shared services will be hosted inside of a svchost.exe process. This will contain multiple DLLs, all running at the same privilege level, with each performing their own specific actions.

Desktops and Threads - Based on what is occurring on a system, multiple threads for a process can be running for a given desktop object.

Desktops and Sessions - Desktops are stored inside of larger containers called Sessions. The desktops are a place to store session specific GUI objects once they are allocated.

VADs and DLLs - VADs keep track of memory allocations for processes. Depending on the type of information that is being saved, a DLL could be mapped into a VAD memory region.

Kernel Modules and Threads - When kernel modules create new threads, the System process (PID 4) becomes the new owner of those objects by default. Each thread owned by System should have offsets into kernel modules that own them. Within the _ETHREAD structure, the SystemThread flag can be used to distinguish between user mode and kernel mode threads. Malicious kernel modules will often create threads and then unload themselves to remain stealthy. Performing this action leaves what are called orphan threads executing on a system.

Message Hooks and DLLs - When a message hook is triggered, the DLL that contains the hooking procedure is mapped into the target thread monitoring for user input. From here, the message that was generated is passed into the hook procedure inside of the DLL.

Event Hooks and DLLs - When event hooks are set, they provide the ability to effectively inject a DLL into the threadâs process address space which performed the event. Using this method provides a simple method to execute code in the context of a remote process.

Event Hooks and Sessions - Event hooks can be set for each of the sessions present on a system.

Kernel Modules and SSDTs - From a non-malicious perspective the pointers inside of the SSDT give access to functions within kernel modules on a system. From a malicious standpoint, kernel modules aim to alter the SSDT using techniques such as pointer replacement or inline hooking to get malicious code executing on a system.

Kernel Modules and Callbacks - Kernel modules use callbacks as an opportunity to monitor events that have occurred on a system. Some types of events monitored by callbacks include: process creation, thread creation, loading an image, system shutdown, registry modification, etc. Malware has used callbacks to continue to perform its functions even after it has performed activities such as direct kernel object manipulation to unlink its KLDR_DATA_TABLE_ENTRY.

Kernel Modules and Drivers - Whenever a kernel module is created a corresponding driver object, _DRIVER_OBJECT, is also initialized. The driver has a copy of the kernel moduleâs base address and itâs unload routine. The driver is useful when attempting to find the creator kernel module that has potentially unloaded itself from memory.

Kernel Modules and IRP Functions - Kernel Modules have the ability to alter the IRP functions set in driver objects. Malicious kernel modules often place hooks or overwrite data in IRP functions. This allows for the kernel modules to potential filter information that is transmitted over the network, written to disk, or other functionality.

Drivers and IRP Functions - Drivers are created with twenty-eight function pointers that can be utilized to perform different actions. The driver object is responsible for configuring these pointers and normally does so before being loaded into memory.

Drivers and Timers - A malicious driver will utilized timers as a method of synchronization and notification. Timers can be set to perform a notification a single time or to send notifications periodically on a system. When a timer expires a deferred procedure call is executed which can point back to the driver that created the timer. This makes timers good artifacts to consult when searching for kernel drivers that could be attempting to conceal themselves.

Kernel Drivers and Sessions - Each session object in memory contains a list of _IMAGE_ENTRY_IN_SESSION structures. Each item in this list represents a device driver that is mapped into the session. Multiple sessions could yield multiple drivers that should be analyzed in each session.

Drivers and Devices - In this relationship, the driver is responsible for creating a device. Once it is created using the API call IoGetDeviceObjectPointer, it can then attach a device to the target device that it desires. After doing this, the new device will receive a copy of the original IRP sent to the target and can âinspectâ it as it sees fit.

Services and Registry - A list of installed services and their configurations are stored in the registry under the key: HKEY\LOCAL\MACHINE\SYSTEM\CurrentControlSet\services.

Each service running on the system has a dedicated subkey that describes how and when each service should be active. It also describes if a service should be a process, dynamic link library (DLL), or kernel driver.

Services and Drivers - Services can implement their actions to be performed through drivers. One such example is the driver that is responsible for implementing the TCP/IP protocol, tcpip.sys. Often services that are associated with drivers help to facilitate communication between the OS and connected hardware.

Sessions and Window Stations - As stated before, sessions act as boundaries for processes and other objects related to a user. The window station is one of the objects that are stored inside of a session. The window station object is used as a security boundary between desktops and processes.

Window Stations and Desktops - Window stations are security boundary objects which can contain multiple desktops.

Window Stations and Atoms - Windows stations can contain multiple atoms which are strings that can be used to communicate with other processes within the same session.

Sessions and Atoms - Sessions can contain multiple atoms that are used for sharing strings between processes in the same session.

Atoms and DLLs - Atoms will often reference different DLLs present on a system. Malware can sometimes create atoms that contain the path to DLLs that have been injected into other processes using message or event hooks.

Atoms and Binaries - Atoms can contain strings that point to the path of binaries that are present on a system.

APPENDIX B

MALWARE FAMILY BREAKDOWN

Table B.1 Dataset Malware Families

Family	Count	Family	Count	Family	Count
singletons	1115	kovter	229	predator	100
agenttesla	996	emotet	190	darkkomet	98
trickbot	946	shyape	183	injuke	89
agensla	807	shiz	167	unruy	88
fareit	784	webdialer	164	remcos	85
zbot	598	bladabindi	160	zusy	83
gozi	474	noon	159	wannacry	82
hlux	387	nanobot	157	avemaria	79
ursnif	379	kelihos	149	zenpak	73
qbot	347	vebzenpak	140	scar	71
gamarue	241	cerber	132	formbook	70
nanocore	240	genkryptik	110	lokibot	69
taskun	65	deyma	35	nakoctb	22
delf	61	sality	34	bitman	22
razy	59	qzonit	33	dapato	22
yuner	58	sodinokibi	33	heye	22
chapak	58	masslogger	33	zboter	21
ceeinject	55	amonetize	32	ymacco	21
wacatac	52	sagecrypt	31	gorgon	21

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
econnect	50	virut	31	prepscram	21
teslacrypt	49	yakes	28	satan	20
agentb	48	netwalker	28	rebhip	20
ramnit	47	mupad	27	upatre	20
chindo	46	farfli	27	msilperseus	20
mansabo	46	bitmin	26	occamy	20
pennybee	42	techsnab	25	cridex	20
rescoms	39	firseria	24	ursu	19
winlock	38	browsefox	24	coins	19
loadmoney	36	bsymem	24	vidar	19
winwebsec	36	softpulse	23	zerber	18
autinject	36	vobfus	23	tofsee	18
trickster	17	azorult	12	glupteba	9
cryptinject	17	omaneat	11	gocloudnet	9
quasar	17	pwszbot	11	stop	9
zegost	16	barys	11	goldenspy	9
bundlore	16	neurevt	11	crysan	9
reconyc	16	dldlmn	11	woool	9
packednet	16	injects	11	bdsnanocoreclient	9

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
tinba	16	racealer	11	boaxxe	8
skeeyah	15	snojan	10	fraud	8
clipbanker	15	nymeria	10	slym	8
banbra	14	spyeye	10	ircbot	8
spybotnet	14	tescrypt	10	mycop	8
nitol	13	llac	10	schinject	8
cosmu	13	spygate	10	redcap	8
wowlik	13	badjoke	10	vasal	8
scarsi	13	nemty	10	maze	8
dial	13	crysis	10	rrat	8
generickdz	13	ponystealer	9	vbkryjetor	7
smartfortress	12	killproc	9	wdfload	7
zeroaccess	7	kpavtoit	7	mekotio	5
cnopa	7	bypassuac	6	flystudio	5
magania	7	mypcbackup	6	parite	5
alien	7	buzus	6	golroted	5
xtrat	7	midie	6	onlinegames	5
zard	7	ngrbot	6	noancooe	5
fsysna	7	sinowal	6	directdownloader	5

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
ymeta	7	kasidet	6	chisburg	5
shopperz	7	poison	6	mudrop	5
sage	7	waledac	6	scrop	5
netwiredrc	7	tiggre	6	danabot	5
bulz	7	swisyn	6	hupigon	5
strongpity	7	neshta	6	windigo	5
delikle	7	msilrandomkrypt	6	avaddon	5
autit	7	gandcrab	6	ragnarlocker	5
eclv	7	equationdrug	6	medusalocker	5
delshad	7	mailto	6	swotter	5
apost	7	dropback	6	injectnet	5
netwire	7	addrop	5	delphi	5
myxah	5	surveyer	4	packy	3
opencandy	4	lockscreen	4	miuref	3
bicololo	4	scenic	4	toga	3
archsms	4	bluteal	4	vbkeyloggerspy	3
urausy	4	disfa	4	psdownload	3
lidared	4	genericgb	4	enestedel	3
shade	4	makoob	4	senta	3

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
msilkrypt	4	titirez	4	ekstak	3
feasu	4	solmyr	4	soft32downloader	3
execcmd	4	autoitinject	4	zapchast	3
sysn	4	rapid	4	dofoil	3
lionsea	4	pwsx	4	runner	3
beebone	4	vbkeylog	4	mydoom	3
banload	4	recam	3	fadok	3
revengerat	4	locky	3	gimemo	3
pincav	4	betacio	3	lotok	3
siscos	4	mikey	3	revetrat	3
bandit	4	lethic	3	lockbit	3
tasker	4	hamweq	3	webdial	3
malwarex	3	miancha	2	njrat	2
pwsteal	3	dynamer	2	disabler	2
wastedlocker	3	scoreem	2	slipafext	2
korplug	3	resetter	2	hesv	2
avemariarat	3	lasdoma	2	hipgnosisbrains	2
hawkeyekeylogger	3	dorkbot	2	quasarrat	2
exorcist	3	dinwod	2	ouroboros	2

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
passwordstealer	3	remoteadmin	2	hosts	2
moderate	2	jimmy	2	crypren	2
usteal	2	dovs	2	wasted	2
badur	2	bankoren	2	black	2
necurs	2	tougle	2	fourthrem	2
sysvenfak	2	invader	2	sharik	2
qhost	2	woozlist	2	macri	2
bifrose	2	steam	2	nymaim	2
induc	2	pliskal	2	vawtrak	2
multiplug	2	nebuler	2	browserpassview	2
adwind	2	orcusrat	2	johnnie	2
bscope	2	variadic	2	wapomi	2
multibar	2	freemake	2	webtoos	1
carberp	2	shelma	2	pontoeb	1
shylock	2	teslaag	2	nbdd	1
blackhole	2	startun	2	vmprotbad	1
xiazai	2	meretam	2	pony	1
nisloder	2	sonbokli	2	mofin	1
orcus	2	conti	2	troldesh	1

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
mucc	2	ragnar	2	prek	1
nagoot	2	sodin	2	kalhine	1
zedopoo	2	manbat	2	2345explorer	1
deceptpcclean	2	mokes	2	schwarzesonne	1
asfortal	2	killmbr	2	betabot	1
carbanak	2	stelega	2	poshkod	1
jeefo	2	agentesla	2	xm4fasx6owcb	1
vbinder	2	luder	2	jacksbot	1
2144flashplayer	2	revenge	2	ukpa	1
dialers	2	batdrop	1	ulise	1
atraps	2	ovidiy	1	blmo	1
diztakun	2	tpyn	1	retefe	1
dothetuk	1	temr	1	regscript	1
philadelphia	1	fbook	1	wysotot	1
convertad	1	lightlogger	1	beaugrit	1
mediamagnet	1	broskod	1	cycbot	1
faketool	1	speedingupmypc	1	swrort	1
pckeeper	1	ransomcwall	1	rbot	1
genericrxaf	1	lynx	1	rsoftware	1

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
dreidel	1	mufanom	1	pandora	1
mint	1	minari	1	instally	1
scriptkd	1	drivertuner	1	panther	1
dodiw	1	karo	1	negasteal	1
installcore	1	fnba	1	autoruns	1
seven	1	pavica	1	hawkeye	1
gepys	1	sunnydigits	1	diffyndor	1
incredimail	1	ananlog	1	sekhmet	1
iiwaaql1eedab	1	fareitvb	1	tocoqi	1
quant	1	generictka	1	pasta	1
megone	1	rincux	1	ruandmel	1
pef13c	1	cutwail	1	inbox	1
xyligan	1	khalesi	1	chir	1
gemius	1	raroger	1	hawkey	1
deceptfakecorel	1	skeeeyah	1	tufik	1
expiro	1	repter	1	systemcheckup	1
sinresby	1	installmonetizer	1	jacard	1
dishigy	1	fakepav	1	fusioncore	1
chromex	1	ruco	1	rovnix	1

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
keybase	1	nefilim	1	dwahk	1
retadup	1	ransomexx	1	sylavriu	1
systemcare	1	masqulab	1	doina	1
pcfixer	1	virlock	1	secureage	1
lamer	1	lightstone	1	cobalt	1
presenoker	1	discord	1	zorab	1
mensa	1	lotoor	1	dropik	1
kuhaname	1	gamini	1	vigua	1
joiner	1	genericml	1	melt	1
genmaliciousa	1	sysdrop	1	revenants	1
betkrypt	1	mbot	1	chepro	1
limerat	1	tenga	1	execution	1
coimine	1	doubleagent	1	pyrgen	1
dridex	1	passup	1	malitrar	1
xmkfrhodjpnbh	1	crimsonrat	1	hebchengjiu	1
graphitefox	1	pretoria	1	indiloadz	1
crypmod	1	socelars	1	deshacop	1
easycleanpc	1	epack	1	msilzilla	1
speedyopt	1	pcrypt	1	nanocoreclient	1

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
foxhiex	1	buerak	1	mimikatz	1
mahato	1	gasti	1	rastarby	1
emogen	1	vigorf	1	bomka	1
pluginaccess	1	blat	1	shellcodecrypter	1
gnnfrk34jphi	1	formatc	1	torjok	1
drmsoft	1	blamon	1	antivm	1
demp	1	fakap	1	plurox	1
spacekito	1	blokrypt	1	crisisht	1
tscope	1	orsam	1	injexa	1
siluhdur	1	mogora	1	sfone	1
icedid	1	trochil	1	regger	1
reveton	1	spybot	1	samca	1
qucan	1	nssm	1	cryfile	1
erica	1	phobos	1	vtflooder	1
virtumonde	1	cryptinjector	1	regotet	1
x1hkg	1	emeka	1	slepak	1
nonoko	1	cutygirls	1	acapulco	1
popclick	1	tspy	1	renos	1
menti	1	schoolgirl	1	crylock	1

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
extenbro	1	dpulsarshellcode	1	regrun	1
cpuguardian	1	shadowbrokers	1	sunburst	1
xmrig	1	topantispyware	1	samas	1
navexcel	1	myplaycity	1	betload	1
armadillo	1	softninjas	1	kpot	1
cometer	1	webcompanion	1	bunitu	1
ludicrouz	1	spyagent	1	staser	1
aitinject	1	psloader	1	proxychanger	1
matrix	1	ashify	1	jobcrypter	1
dawnla	1	hmofa0pd7gfb	1	povertel	1
hydrapos	1	smser	1	cossta	1
ripter	1	maria	1	waldek	1
warezov	1	zeppelin	1	dllcheck	1
pcclient	1	asparnet	1	fakeie	1
folders	1	cnsmin	1	pynamer	1
spyagent	1	pakes	1	zurgop	1
torct	1	rackcrypt	1	sload	1
lulusoftware	1	viking	1	rozena	1
fakesupport	1	downloadadmin	1	msilheracles	1

Table B.1 Dataset Malware Families (continued)

Family	Count	Family	Count	Family	Count
savenow	1	daws	1	ygdata	1
insqter	1	cidox	1	rnaph	1
zpevdo	1	amavaldo	1	agentoit	1
wget	1	bobik	1	liusky	1
valak	1	phorpiex	1	coinstealer	1
xred	1	genasep	1	bobik	1
totavel	1	zxshell	1	phorpiex	1
chthonic	1	valak	1	genasep	1
limeransom	1	liusky	1	zxshell	1
leivion	1	coinstealer	1	wget	1
binder	1	bobik	1	psloader	1
msilheracles	1	phorpiex	1	zurgop	1
ygdata	1	genasep	1	sload	1
rnaph	1	zxshell	1	rozena	1
agentoit	1	wget	1	psloader	1

APPENDIX C

NORMALIZED PATHS INVERSE DOCUMENT FREQUENCY (IDF)

In this section the normalized paths and their respective IDF scores are shown in the table below. These paths were generated from the pairs of malicious processes in our dataset.

Table C.1 Normalized Path IDF Scores

Normalized Path	**IDF Score**
process – parent_child_relationship – process – process_contains_binary – binary – binary_being_used_by_process – process – parent_child_relationship – process	5.558960
process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_has_handle_to_process – process – parent_child_relationship – process	5.543212
process – process_contains_module – module – module_associated_with_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process	5.527707
process – process_connected_to_desktop – desktop – desktop_contains_process – process – process_has_parent_child_relationship_and_handle_to_process – process – parent_child_relationship – process	5.504893

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_process – process – process_contains_binary – binary – binary_being_used_by_process – process – process_has_handle_to_process – process	5.497402
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process	5.467988
process – process_has_parent_child_relationship_and_handle_to_process – process – process_contains_binary – binary – binary_being_used_by_process – process – process_has_handle_to_process – process	5.453600
process – parent_child_relationship – process – process_has_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	5.453600
process – process_found_in_session – session – session_contains_message_hook – message_hook – message_hook_associated_with_module – module – module_used_by_process – process	5.446482

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – parent_child_relationship – process – parent_child_relationship – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	5.439415
process – parent_child_relationship – process – parent_child_relationship – process – parent_child_relationship – process – parent_child_relationship – process	5.404809
process – parent_child_relationship – process – process_has_handle_to_process – process – process_has_handle_to_event – event – event_being_used_by_process – process	5.358289
process – process_found_in_session – session – session_contains_message_hook – message_hook – message_hook_created_by_process – process – parent_child_relationship – process	5.345386
process – process_has_parent_child_relationship_and_handle_to_process – process – process_found_in_session – session – session_contains_message_hook – message_hook – message_hook_created_by_process – process	5.271278

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_contains_binary – binary – binary_being_used_by_process – process – parent_child_relationship – process	5.265343
process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process – process_has_parent_child_relationship_and_handle_to_process – process	5.265343
process – process_has_handle_to_event – event – event_being_used_by_process – process – parent_child_relationship – process – process_has_handle_to_process – process	5.219092
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	5.219092
process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_has_handle_to_process – process	5.196745
process – process_contains_binary – binary – binary_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	5.148218

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_process – process – process_has_handle_to_process – process – parent_child_relationship – process	5.096925
process – process_has_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – parent_child_relationship – process	5.096925
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	5.057704
process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – parent_child_relationship – process	5.052908
process – process_has_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_has_handle_to_process – process	5.052908
process – parent_child_relationship – process – parent_child_relationship – process – process_has_handle_to_thread – thread – thread_created_by_process – process	4.965896

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process – parent_child_relationship – process	4.939921
process – process_found_in_session – session – session_contains_process – process – parent_child_relationship – process – process_has_handle_to_process – process	4.873781
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.850064
process – process_created_thread – thread – thread_being_used_by_process – process – process_has_handle_to_event – event – event_being_used_by_process – process	4.782114
process – parent_child_relationship – process – process_has_handle_to_process – process – process_contains_module – module – module_used_by_process – process	4.739250
process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	4.708267

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.684815
process – parent_child_relationship – process – process_connected_to_desktop – desktop – desktop_contains_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.668394
process – parent_child_relationship – process – process_connected_to_desktop – desktop – desktop_contains_process – process – parent_child_relationship – process	4.642669
process – parent_child_relationship – process – process_has_handle_to_process – process – process_found_in_session – session – session_contains_process – process	4.630051
process – process_has_handle_to_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	4.623801
process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process – parent_child_relationship – process	4.620690

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process – process_has_handle_to_process – process	4.578131
process – parent_child_relationship – process – process_has_handle_to_thread – thread – thread_relationship_with_process_and_process_using_handle_to_thread – process – process_has_handle_to_process – process	4.523124
process – process_has_handle_to_thread – thread – thread_relationship_with_process_and_process_using_handle_to_thread – process – process_has_handle_to_process – process	4.520311
process – process_has_handle_to_thread – thread – thread_being_used_by_process – process – process_has_handle_to_process – process – process_has_handle_to_process – process	4.509138
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process – process_contains_binary – binary – binary_being_used_by_process – process	4.492609

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_event – event – event_being_used_by_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	4.476348
process – process_has_parent_child_relationship_and_handle_to_process – process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process	4.462997
process – parent_child_relationship – process – process_has_handle_to_process – process – process_has_handle_to_token – token – token_being_used_by_process – process	4.457706
process – process_found_in_session – session – session_contains_process – process – process_has_thread_relationship_and_has_handle_to_thread – thread – thread_being_used_by_process – process	4.449821
process – process_has_handle_to_token – token – token_being_used_by_process – process – process_has_thread_relationship_and_has_handle_to_thread – thread – thread_being_used_by_process – process	4.449821

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_thread – thread – thread_relationship_with_process_and_process_using_handle_to_thread – process – process_has_handle_to_event – event – event_being_used_by_process – process	4.449821
process – process_has_handle_to_thread – thread – thread_relationship_with_process_and_process_using_handle_to_thread – process – process_contains_module – module – module_used_by_process – process	4.444599
process – process_has_handle_to_thread – thread – thread_relationship_with_process_and_process_using_handle_to_thread – process – process_found_in_session – session – session_contains_process – process	4.444599
process – process_has_handle_to_thread – thread – thread_relationship_with_process_and_process_using_handle_to_thread – process – process_connected_to_desktop – desktop – desktop_contains_process – process	4.444599
process – process_found_in_session – session – session_contains_process – process – process_has_thread_relationship_and_has_handle_to_thread – thread – thread_created_by_process – process	4.408790

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_contains_module – module – module_used_by_process – process – process_has_thread_relationship_and_has_handle_to_thread – thread – thread_created_by_process – process	4.406281
process – process_created_thread – thread – thread_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.406281
process – process_has_handle_to_token – token – token_being_used_by_process – process – process_has_thread_relationship_and_has_handle_to_thread – thread – thread_created_by_process – process	4.398790
process – parent_child_relationship – process – parent_child_relationship – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	4.396305
process – parent_child_relationship – process – process_has_handle_to_thread – thread – thread_being_used_by_process – process – process_has_handle_to_process – process	4.340803
process – process_has_parent_child_relationship_and_handle_to_process – process – parent_child_relationship – process	4.338458

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_thread – thread – thread_being_used_by_process – process – process_contains_module – module – module_used_by_process – process	4.333785
process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.322197
process – process_contains_binary – binary – binary_being_used_by_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	4.317600
process – process_created_thread – thread – thread_being_used_by_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	4.308467
process – process_found_in_session – session – session_contains_process – process – process_has_handle_to_thread – thread – thread_being_used_by_process – process	4.266192
process – process_has_handle_to_token – token – token_being_used_by_process – process – process_has_handle_to_thread – thread – thread_being_used_by_process – process	4.266192

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_created_thread – thread – thread_being_used_by_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	4.261844
process – process_has_handle_to_process – process – parent_child_relationship – process – process_contains_binary – binary – binary_being_used_by_process – process	4.261844
process – process_has_handle_to_process – process – process_contains_binary – binary – binary_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.261844
process – process_connected_to_desktop – desktop – desktop_contains_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	4.238263
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_event – event – event_being_used_by_process – process	4.229824

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process – process_has_parent_child_relationship_and_handle_to_process – process	4.198798
process – process_has_handle_to_thread – thread – thread_being_used_by_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	4.196763
process – process_has_handle_to_thread – thread – thread_being_used_by_process – process – process_found_in_session – session – session_contains_process – process	4.192706
process – process_has_handle_to_thread – thread – thread_being_used_by_process – process – process_has_handle_to_process – process	4.192706
process – parent_child_relationship – process – parent_child_relationship – process – process_contains_binary – binary – binary_being_used_by_process – process	4.129941

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_event – event – event_being_used_by_process – process – parent_child_relationship – process	4.116721
process – parent_child_relationship – process – process_has_handle_to_process – process	4.090793
process – process_has_handle_to_process – process – parent_child_relationship – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	4.042613
process – process_contains_module – module – module_used_by_process – process – parent_child_relationship – process – process_has_handle_to_process – process	4.039134
process – process_has_handle_to_process – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	4.032214
process – parent_child_relationship – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	4.027055

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_thread – thread – thread_created_by_process – process – process_found_in_session – session – session_contains_process – process	4.005001
process – process_connected_to_desktop – desktop – desktop_contains_process – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	3.988362
process – process_has_handle_to_event – event – event_being_used_by_process – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	3.988362
process – process_has_parent_child_relationship_and_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process – parent_child_relationship – process	3.962302
process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.962302

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_event – event – event_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.941617
process – parent_child_relationship – process – process_has_handle_to_event – event – event_being_used_by_process – process – parent_child_relationship – process	3.935337
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	3.935337
process – process_found_in_session – session – session_contains_process – process – process_created_thread – thread – thread_being_used_by_process – process	3.933774
process – process_has_handle_to_thread – thread – thread_created_by_process – process – process_has_handle_to_token – token – token_being_used_by_process – process	3.915197

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_created_thread – thread – thread_being_used_by_process – process – process_found_in_session – session – session_contains_process – process	3.913665
process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	3.910607
process – process_has_handle_to_process – process – parent_child_relationship – process – parent_child_relationship – process	3.906037
process – process_created_thread – thread – thread_being_used_by_process – process – process_contains_module – module – module_used_by_process – process	3.904518
process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_event – event – event_being_used_by_process – process	3.899976
process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_contains_binary – binary – binary_being_used_by_process – process	3.879049

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.858550
process – process_contains_module – module – module_used_by_process – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	3.851330
process – parent_child_relationship – process – parent_child_relationship – process – parent_child_relationship – process	3.815991
process – process_connected_to_desktop – desktop – desktop_contains_process – process – parent_child_relationship – process – parent_child_relationship – process	3.803568
process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_message_hook – message_hook – message_hook_created_by_process – process	3.799461
process – process_created_thread – thread – thread_being_used_by_process – process – process_has_handle_to_token – token – token_being_used_by_process – process	3.787241

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process – parent_child_relationship – process	3.772506
process – process_has_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.769850
process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_contains_binary – binary – binary_being_used_by_process – process	3.767201
process – process_has_handle_to_token – token – token_being_used_by_process – process – parent_child_relationship – process – process_has_handle_to_process – process	3.750151
process – process_has_handle_to_process – process – process_has_handle_to_process – process – process_has_handle_to_process – process	3.741089

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	3.733387
process – process_has_handle_to_process – process – process_found_in_session – session – session_contains_message_hook – message_hook – message_hook_created_by_process – process	3.728285
process – parent_child_relationship – process – process_has_handle_to_thread – thread – thread_created_by_process – process	3.721944
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	3.720681
process – process_has_handle_to_event – event – event_being_used_by_process – process – parent_child_relationship – process – parent_child_relationship – process	3.698208
process – parent_child_relationship – process – parent_child_relationship – process – process_has_handle_to_event – event – event_being_used_by_process – process	3.659462

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_token – token – token_being_used_by_process – process – process_created_thread – thread – thread_being_used_by_process – process	3.653541
process – parent_child_relationship – process – process_found_in_session – session – session_contains_message_hook – message_hook – message_hook_created_by_process – process	3.638310
process – process_has_handle_to_event – event – event_being_used_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process – parent_child_relationship – process	3.604020
process – process_found_in_session – session – session_contains_message_hook – message_hook – message_hook_created_by_process – process – process_has_handle_to_process – process	3.510755
process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	3.500572

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_connected_to_desktop – desktop – desktop_contains_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process	3.500572
process – process_has_handle_to_process – process – process_contains_binary – binary – binary_being_used_by_process – process	3.478527
process – parent_child_relationship – process – parent_child_relationship – process – process_connected_to_desktop – desktop – desktop_contains_process – process	3.459871
process – process_found_in_session – session – session_contains_process – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	3.455019
process – process_has_handle_to_process – process – process_has_handle_to_process – process	3.454052
process – process_has_handle_to_thread – thread – thread_created_by_process – process	3.429218

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_contains_binary – binary – binary_being_used_by_process – process – process_has_handle_to_event – event – event_being_used_by_process – process	3.380428
process – process_contains_binary – binary – binary_being_used_by_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	3.380428
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_contains_module – module – module_used_by_process – process	3.373266
process – process_contains_module – module – module_used_by_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	3.346856
process – process_contains_module – module – module_used_by_process – process – parent_child_relationship – process – parent_child_relationship – process	3.341656
process – process_found_in_session – session – session_contains_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	3.327070

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_process – process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process	3.308504
process – process_has_handle_to_process – process – parent_child_relationship – process	3.305999
process – process_has_handle_to_token – token – token_being_used_by_process – process – process_has_handle_to_thread – thread – thread_created_by_process – process	3.300178
process – process_has_handle_to_token – token – token_being_used_by_process – process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process	3.285362
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_found_in_session – session – session_contains_process – process	3.258759
process – parent_child_relationship – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	3.255582

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process	3.223585
process – parent_child_relationship – process – process_contains_binary – binary – binary_being_used_by_process – process	3.221284
process – process_has_handle_to_process – process – parent_child_relationship – process – process_connected_to_desktop – desktop – desktop_contains_process – process	3.211371
process – parent_child_relationship – process – process_has_parent_child_relationship_and_handle_to_process – process – process_contains_module – module – module_used_by_process – process	3.200055
process – parent_child_relationship – process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process	3.197807
process – process_created_message_hook – message_hook – message_hook_in_session – session – session_contains_process – process – process_has_handle_to_process – process	3.146738

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_connected_to_desktop – desktop – desktop_contains_thread – thread – thread_created_by_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.113856
process – process_connected_to_desktop – desktop – desktop_contains_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.113856
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_process – process	3.113168
process – process_has_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process	3.104949
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_event – event – event_being_used_by_process – process	3.069434
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_parent_child_relationship_and_handle_to_process – process – process_connected_to_desktop – desktop – desktop_contains_process – process	3.069434

Table C.1 Normalized Path IDF Scores (continued)

Normalized Path	**IDF Score**
process – process_has_parent_child_relationship_and_handle_to_process – process – process_has_handle_to_mutant – mutant – mutant_being_used_by_process – process – process_has_handle_to_process – process	3.062219