Mississippi State University
Scholars Junction

Theses and Dissertations Theses and Dissertations

1-1-2015

Automating Malware Detection in Windows Memory Images using Machine Learning

Dae Glendowne

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation

Glendowne, Dae, "Automating Malware Detection in Windows Memory Images using Machine Learning" (2015). *Theses and Dissertations*. 830.
https://scholarsjunction.msstate.edu/td/830

This Dissertation - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

Automating malware detection in Windows memory images using machine learning

By

Dae Glendowne

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

May 2015

Copyright by

Dae Glendowne

2015

Automating malware detection in Windows memory images using machine learning

By

Dae Glendowne

Approved:

David A. Dampier
(Major Professor)

Christopher Archibald
(Committee Member)

Robert Wesley McGrew
(Committee Member)

Mahalingam Ramkumar
(Committee Member)

T.J. Jankun-Kelly
(Graduate Coordinator)

Jason M. Keith
Interim Dean
Bagley College of Engineering

Name: Dae Glendowne

Date of Degree: May 8, 2015

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. David A. Dampier

Title of Study: Automating malware detection in Windows memory images using machine learning

Pages of Study: 130

Candidate for Degree of Doctor of Philosophy

Malicious software, or malware, is often employed as a tool to maintain access to previously compromised systems. It enables the intruders to utilize system resources, harvest legitimate credentials, and maintain a level of stealth throughout the process. During incident response, identifying systems infected with malware is necessary for effective remediation of an attack. When analysts lack sufficient indicators of compromise they are forced to conduct a comprehensive examination to identify anomalous behavior on a system, a time consuming and challenging task. Malware authors use several techniques to conceal malware on a system, with a common method being DLL injection.

In this dissertation we present a system for automatically generating Windows 7 x86 memory images infected with malware, identifying the malicious DLLs injected into a process, and extracting the features associated with those DLLs. A set of 3,240 infected memory images was produced and analyzed to identify common characteristics of malicious DLLs in memory. From this analysis a feature set was constructed and two datasets

were used to evaluate five classification algorithms. The ZeroR method was used as a baseline for comparison with accuracy and false positive rate (misclassifying malicious DLLs as legitimate) being the two metrics of interest. The results of the experiments showed that learning using the feature set is viable and that the performance of the classifiers can be further improved through the use of feature selection. Each of the classification methods outperformed the ZeroR method with the J48 Decision Tree obtaining the, overall, best results.

Key words: DLL injection, malware classification, memory analysis

DEDICATION

To my wife, Puntitra Sawadpong, and my parents, Jan and Richard Glendowne.

ACKNOWLEDGEMENTS

As with any work of this scale, it would not have been possible alone. I am fortunate to have had support and guidance from my professors, my fellow students, my friends, and my family.

I thank my major professor, Dr. David Dampier. His guidance through these (far too) many years was essential in my success. I thank my committee members, Dr. Christopher Archibald, Dr. Wesley McGrew, and Dr. Mahalingam Ramkumar. Their intellectual contributions have been of great value in this work.

My sincere thanks to my parents, Jan and Richard Glendowne. Your love and support (and DNA) has helped shape me into the person I am today.

Finally, to my loving wife, Tan. You are the love of my life and a daily source of inspiration. I see you and realize there is nothing that I cannot achieve. You have shouldered the weight of this achievement with me, and for that I am truly grateful. You have chosen to stand by my side, and for that I am truly blessed.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1. INTRODUCTION 1

1.1 Memory Forensics 1
1.2 Goal 3
1.3 Malware in Memory 3
1.4 Dynamic Link Library Injection 5
1.5 DLL Data Structures in Memory 7
1.6 Identifying Malware in Memory Samples 7
1.7 Hypothesis 9
1.8 Research Questions 10

2. RELATED WORK 12

2.1 Malware Classification Using Static Features 12
2.2 Malware Classification Using Dynamic Features 14
2.3 Malware Detection in Memory 16
2.4 Classification Algorithms 16
2.4.1 ZeroR 17
2.4.2 Naive Bayes 17
2.4.3 J48 17
2.4.4 Support Vector Machine 18
2.4.5 Nearest Neighbor 18
2.4.6 Voted Perceptron 18
2.5 Correlation-Based Feature Selection 18
2.5.1 Best-First Search 19

2.5.1.1 Forward Selection 19
2.5.1.2 Backward Elimination 19
2.6 Gain Ratio Attribute Selection 19
2.6.1 Ranker Search 20
2.7 Bagging 20

3. METHODOLOGY 21

3.1 Data Generation 21
3.1.1 Virtual Machines 21
3.1.2 Cuckoo Sandbox Architecture 22
3.1.3 INetSim 23
3.1.4 Malware Samples 24
3.2 Extracting DLLs from Memory 24
3.3 Data Cleaning 25
3.4 Data Labeling 26
3.5 Dataset Creation 27

4. PRELIMINARY RESULTS 29

4.1 Injected DLL Characteristics 29
4.1.1 Target Processes 30
4.1.2 Number of Injections per Malware 31
4.1.3 Simultaneous Loads 31
4.1.4 Load Position 32
4.1.5 Init Position 33
4.1.6 Base Address 33
4.1.7 Exported Function Count 33
4.1.8 Imported Function Count 34
4.1.9 Loaded from Temp 34
4.1.10 Load Paths 34
4.1.11 COM Server 34
4.1.12 COM Client 35

5. FEATURE SET 37

5.1 _EPROCESS Features 37
5.2 _LDR_DATA_TABLE_ENTRY Features 38
5.3 _MMVAD_* Features 40
5.4 _PE_HEADER Features 41

6. EXPERIMENTAL RESULTS 43

6.1 Research Question 2 . . . 43
6.1.1 Base Classifier Evaluation - Cross-Validation Set . . . 44
6.1.2 Base Classifier Evaluation - Test Set . . . 44
6.2 Research Question 3 . . . 46
6.2.1 Correlation-based Feature Selection . . . 48
6.2.1.1 Classifier with CFS - Cross-Validation Set . . . 49
6.2.1.2 Classifier with CFS - Test Set . . . 51
6.2.2 GainRatio Attribute Evaluation using Ranker . . . 53
6.2.2.1 Classifier with GainRatio Selection - Cross-Validation Set . . . 53
6.2.2.2 Classifier with GainRatio Selection - Test Set . . . 56
6.3 Research Question 4 . . . 56
6.3.1 Classifiers with Bagging - Cross-Validation Set . . . 58
6.3.2 Classifiers with Bagging - Test Set . . . 59
6.4 Memory Features versus PE Header Features . . . 62
6.4.1 Feature Comparison - Cross-Validation Set . . . 63
6.4.2 Feature Comparison - Test Set . . . 65

7. ANALYSIS . . . 68

7.1 Do malicious DLLs have distinct patterns of behavior in memory? 68
7.2 How do different classifiers perform against the dataset? . . . 68
7.3 Do correlation-based feature selection and gain ratio evaluation improve classifier performance for this dataset? . . . 69
7.3.1 Correlation-Based Feature Selection using Best-First Search 69
7.3.2 Gain Ratio Attribute Evaluation using Ranker . . . 70
7.3.3 Evaluation . . . 70
7.4 Can the ensemble learning technique Bagging improve performance over an individual classifier? . . . 71
7.5 Memory and PE Feature Comparison . . . 72
7.6 J48 Decision Tree . . . 73
7.6.1 J48 False Positives . . . 73
7.6.2 J48 False Negatives . . . 75

8. CONCLUSIONS . . . 76

8.1 Publication Plan . . . 78
8.2 Contributions . . . 78
8.2.1 New Infected Memory Images . . . 78
8.2.2 Datasets . . . 79
8.2.3 DLL Characteristics . . . 79
8.2.4 Experimental Results . . . 80
8.3 Volatility Plugins . . . 80

8.3.1 Aiding Forensic Examiners 80
8.3.2 Automated Detection 81
8.4 Future Work 81
8.4.1 Feature Set 82
8.4.2 Other Types of Malware in Memory 82

REFERENCES 83

APPENDIX

A. SOFTWARE VERSIONS FOR LEGITIMATE DATA 88

B. FEATURE EXTRACTION SOURCE CODE 90

B.1 Classes for Extracting DLL Features from Windows 7 x86 Memory Images 91

C. DATA PREPROCESSING SOURCE CODE 114

C.1 Python Code for Preprocessing Dataset 115

LIST OF TABLES

3.1 Class Distribution for the Cross-Validation and Test Datasets 28

6.1 Base Classifier Accuracy and False Positive Rate - Cross-Validation Set . . 45

6.2 Base Classifier Accuracy and False Positive Rate - Test Set 47

6.3 CFS using Best First Search Attribute Subsets 49

6.4 Classifiers using CFS Attribute Selection - CV Set 50

6.5 Classifiers using CFS Attribute Selection - Test Set 52

6.6 Gain Ratio Attribute Selection using Ranker Search 54

6.7 Classifiers with Gain Ratio - Cross-Validation Set 54

6.8 Classifiers with Gain Ratio - Test Set 57

6.9 Classifiers with Bagging - Cross-Validation Set 59

6.10 Classifiers with Bagging - Test Set 61

6.11 Memory and PE Feature Metrics - Cross-Validation Set 64

6.12 Memory and PE Feature Metrics - Test Set 66

7.1 False Positives from Base J48 Decision Tree 74

7.2 False Negatives from Base J48 Decision Tree 75

A.1 Legitimate Third-Party Software 89

LIST OF FIGURES

1.1 Process Hollowing (Image Taken from [29]) 4

1.2 DLL Injection (Image taken from [22]) 6

1.3 Doubly Linked Lists in the Process Environment Block 8

3.1 Data Generation Process 22

4.1 Processes Commonly Targeted for DLL Injection 30

4.2 Distribution of DLL Injections per Malware 31

4.3 Distribution of Load Paths per Malware 35

6.1 Classifier Accuracy - Cross-Validation Set 45

6.2 Classifier False Positive Rate - Cross-Validation Set 46

6.3 Classifier Accuracy - Test Set 47

6.4 Classifier False Positive Rate - Test Set 48

6.5 Classifier Accuracy Change using CFS - Cross-Validation Set 50

6.6 Classifier FP Rate Change using CFS - Cross-Validation Set 51

6.7 Classifier Accuracy Change using CFS - Test Set 52

6.8 Classifier FP Rate Change using CFS - Test Set 53

6.9 Classifier Accuracy Change using GR Evaluation - Cross-Validation Set . . 55

6.10 Classifier FP Rate Change using GR Evaluation - Cross-Validation Set . . . 56

6.11 Classifier Accuracy Change using GR Evaluation - Test Set 57

6.12 Classifier FP Rate Change using GR Evaluation - Test Set 58
6.13 Classifier Accuracy Change using Bagging - Cross-Validation Set 60
6.14 Classifier FP Rate Change using Bagging - Cross-Validation Set 60
6.15 Classifier Accuracy Change using Bagging - Test Set 61
6.16 Classifier FP Rate Change using Bagging - Test Set 62
6.17 Memory and PE Features Accuracy - Cross-Validation Set 64
6.18 Memory and PE Features FP Rate - Cross-Validation Set 65
6.19 Memory and PE Features Accuracy - Test Set 66
6.20 Memory and PE Features FP Rate - Test Set 67

CHAPTER 1

INTRODUCTION

Malicious software, or malware, is a growing threat seen across multiple domains ranging from the home user to the nation state level. When threat actors target a network for intrusion, malware is often used as a method for maintaining access to a previously compromised network [31, 32, 33, 34, 35]. During and after an attack, it falls to the incident response team to determine which systems have been compromised and purge the intruders and their tools from the network.

The incident response team performs forensic analysis of the environment at the host and network level to determine which systems have been affected by the intrusion. The size and skill of the team will determine the roles each member assumes. Signs of the intrusion are sought in network traffic, on disk, and in the memory of host systems. Being able to identify and analyze resident malware across multiple systems is necessary for effective remediation of an attack [43]. Time is a critical factor in this as intruders may be stealing or destroying data while the incident response team is assessing the situation.

1.1 Memory Forensics

Forensic analysis of captured memory is relatively new [45] with dedicated tools only becoming widely available since 2008 [37, 63, 49, 50]. With the realization of the sheer

volume and usefulness of information contained within memory has come significant research and software developed for extracting and processing this data. Any type of forensic information can potentially be extracted from memory, but malware has been a focal point for research in this area with many tools offering dedicated features for extracting malicious artifacts.

Some pieces of malware are built to remain concealed for as long as possible while performing their tasks, but for the malicious code to be executed it must exist in memory. This is known as the rootkit paradox [28] and memory analysis leverages this to find malware in captured memory. Many of the defenses malware employs to evade detection on a live system, such as hooking or direct kernel object manipulation [30], are rendered inoperable during an offline analysis of memory. Any software executing at the time that memory is captured can eventually be found; it is simply a function of the time and skill of the analyst.

Despite the tools currently available, the process for initially finding malware in a memory capture is a manual process requiring significant knowledge of operating system internals by the analyst. Depending on what is known a priori, this search can be tightly focused on an area of memory. If, however, there is no specific knowledge related to the infection (i.e. specific port, ip address, process, etc...) then the analyst will have to examine all aspects of memory looking for anomalous or suspicious behavior. Process listings, handles to resources, threads, active and recently closed network connections, file system artifacts in memory, and registry values are just some examples of areas of interest.

1.2 Goal

The goal of this dissertation is to determine if malicious DLLs in Windows memory images can be reliably identified using machine learning.

1.3 Malware in Memory

Malware can exist in several forms in memory. The chosen form will depend on the malware author's skill and the requirements of the malware itself. Not all malware authors are concerned with stealth or evading detection mechanisms, but those that are have several options for concealing malicious software. Malware may run as a standalone process, as code injected into another process, as a hollowed process, as a service, or as a driver. Each form of malware on a system has its own characteristics that distinguish it from others and each generates a different set of artifacts that may be used for detection based on that form. The different forms are briefly described here.

Standalone processes are processes that are executed normally such as when a user opens or clicks an application. The executable will be loaded into memory and added to the linked list of processes. The name of the process will match that of the file on disk. If a binary named test.exe is ran, either by another application or the user initiating the program, then it will appear as test.exe in the process listing.

Hollowed processes are legitimate processes that have been started, suspended, and had the legitimate code replaced with malicious code. Malicious launchers are a type of malware responsible for launching the primary malicious code on an infected system. With this technique, they open a benign binary, most often a system binary, with the primary

thread in the suspended state. The malware launcher then frees the memory containing the sections of the legitimate process, hollowing out the process data structure in memory. It then reallocates memory and writes its own PE sections into the body of the process. Figure 1.1 shows the Local Security Authority Subsystem Service being hollowed and replaced with the contents of malware.exe.

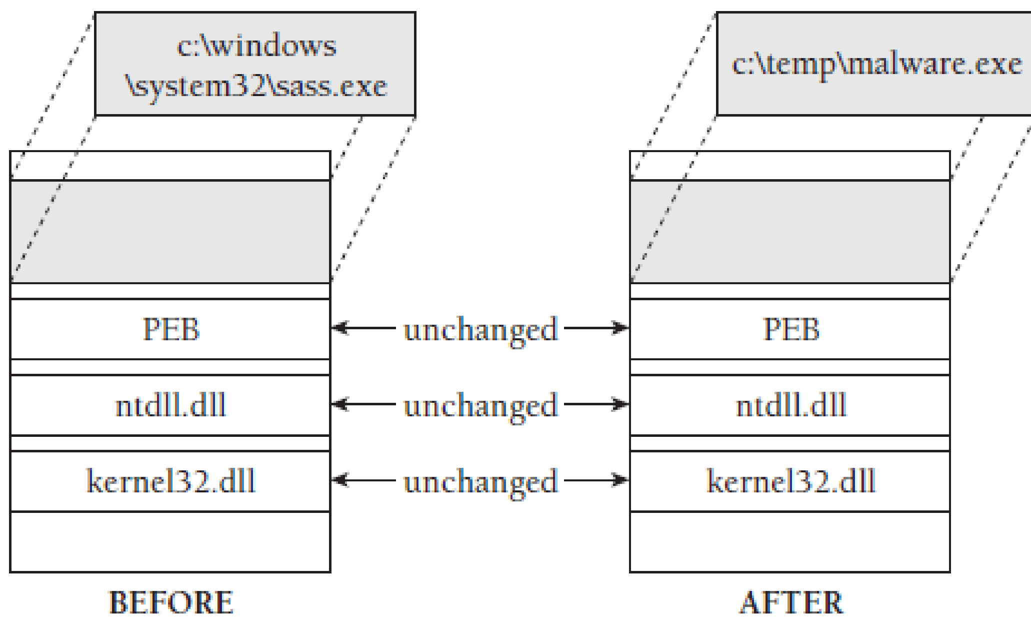

Figure 1.1

Process Hollowing (Image Taken from [29])

Injected code is executable code loaded directly into a portion of allocated memory in a remote process. This lacks the overhead of a DLL as well as the overall structure. The code injected into the remote process is normally shellcode; shellcode is self-contained executable code [22]. Injected code aids malware in avoiding anomaly-based network intrusion detection systems [22, 31].

Windows services run in the background and are managed by the operating system. There are normally several services running at one time in Windows. Services are encapsulated by the system process svchost.exe, a container for multiple services implemented as DLLs.

Windows drivers are binaries that execute in kernel space. They naturally have more privileges than code executing in user space. Malicious drivers are more difficult to implement and dangerous to the system. Arbitrarily modifying kernel memory can result a system crash.

1.4 Dynamic Link Library Injection

A dynamic link library (DLL) is a shared library of code that may be mapped into multiple processes simultaneously. The intended purpose of a DLL is for code reuse and reducing the footprint in memory of shared functions. DLL injection is a technique that forces a running process to load an arbitrary DLL. Malware authors use DLL injection to conceal their code, evade host-based firewalls and process-specific security measures, and leverage the execution context of the containing process [22].

As shown in Figure 1.2, when Launcher Malware tries to initiate an outbound connection it is blocked by a host-based firewall. Launcher Malware loads Malicious DLL into iexplore.exe (Internet Explorer). This enables Malicious DLL to reach the internet because any traffic originating from it appears to come from iexplore.exe which is a trusted process.

DLLs can be loaded into a process by dynamic linking or run-time dynamic linking. Dynamic linking occurs during process initialization. DLLs listed in the import address

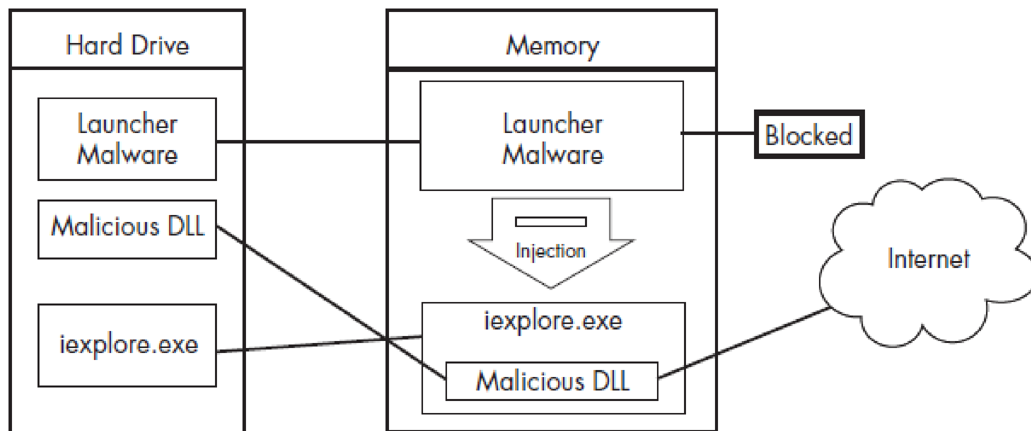

Figure 1.2

DLL Injection (Image taken from [22])

table (IAT) of the executable will be loaded at this stage. Those DLLs that are loaded also contain an IAT and will subsequently load any required DLLs. This process continues until all necessary DLLs are loaded into the process [51].

Run-time dynamic linking is when a DLL is loaded into a process after the process has been initalized and is running. Windows 7 natively supports this functionality and most Windows system processes perform this action repeatedly during their lifetime. When the functionality of a DLL is required by the executable in a process or by another DLL in that process, it will load the desired library and execute the appropriate code. If it no longer needs the DLL it may unload the DLL from the process.

DLL injection can be accomplished through multiple methods, but each method simulates legitimate DLL loading mechanisms within the system. The loading itself does not create any distinctly malicious artifacts. The runtime loading of a DLL into a process is a common event on a running system and in itself is not malicious nor even suspicious.

1.5 DLL Data Structures in Memory

A process is represented by an _EPROCESS structure in memory. Each _EPROCESS structure contains a *Process Environment Block* (PEB). The PEB contains three doubly linked lists that store the DLLs loaded by the process as a *_LDR_DATA_TABLE_ENTRY*. The three linked lists are the *LoadOrderList, MemoryOrderList, and InitOrderList*.

Each list contains the same DLLs for that process, but the order is different per list. The *LoadOrderList* places the DLLs in order they were loaded into process. The *MemoryOrderList* is ordered based on the virtual memory address where a DLL is loaded. The *InitOrderList* is based on the order in which the main function of each DLL was executed.

Each *_LDR_DATA_TABLE_ENTRY* contains members describing the DLL in memory.

- DllBase - Virtual address in the process where the DLL is loaded
- SizeOfImage - Size of the loaded DLL
- BaseDllName - The DLL's name
- FullDllName - Path to the DLL on disk including the DLL's name
- LoadCount - Number of times LoadLibrary was called for the module

1.6 Identifying Malware in Memory Samples

Antivirus software (AVS) is standard in many corporate environments. Utilizing signature-based detection, AVS is capable of detecting numerous known malware samples as well as variants of some samples. However, AVS is not sufficient on its own and it does not handle new malware well [6]. The signatures used by AVS require reverse engineering the malware; a time-consuming, difficult process that does not scale to the millions of new malware samples discovered each month [2].

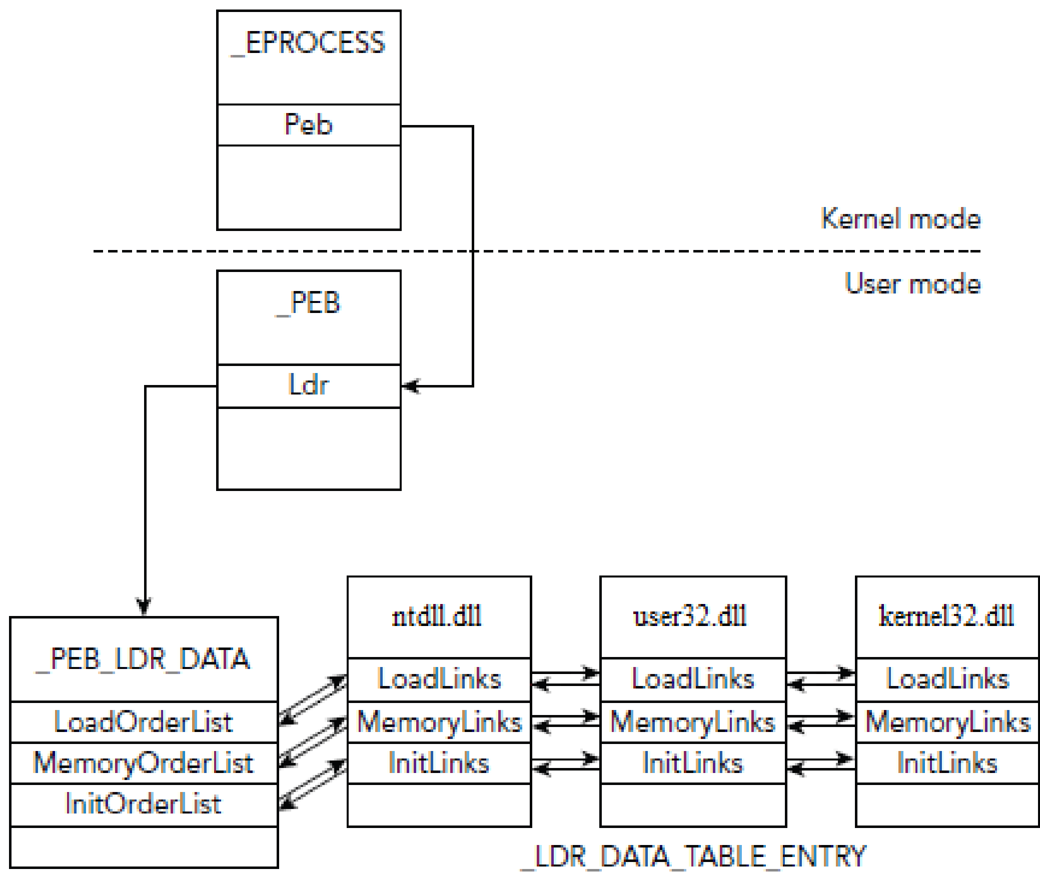

Figure 1.3

Doubly Linked Lists in the Process Environment Block

Typically, there is no clear distinction between malicious and legitimate software. Rather, it is based on usage and context. For example, there are many pieces of legitimate software that enable a remote connection to a server across the Internet. Some software may even routinely transmit information to a remote system. This functionality is not inherently considered malicious. If the software is reading personal files on the system unrelated to its stated purpose and then transmitting their contents to a remote location then it would likely be considered malicious. Because of this lack of definitive distinction, machine learning is a viable approach for classifying code in memory.

Machine learning has previously been used to classify malicious Windows Portable Executable (PE) files with high success. However, the attributes used to train the classifier were more generic as they were extracted from all types of PE files. Additionally, DLLs loaded into memory have features specific to memory that do not exist in the binary. These attributes have not been applied to the malware classification problem.

1.7 Hypothesis

The hypothesis for this dissertation is:

> **A machine learning model can be learned from features extracted from Windows 7 memory images and applied to successfully classify malicious injected DLLs in Windows 7 memory images.**

There are two metrics this research is primarily concerned with regarding the model: accuracy and false negatives. Accuracy is the number of correctly classified data points of either class, legitimate or malicious. False negatives are malicious DLLs that have been classified as legitimate.

Successfully is defined as a classifier that produces a model having a greater accuracy and lower false negative rate than that of the ZeroR classifier applied to the same dataset. Witten et al. [66] propose using the ZeroR method for establishing a baseline. The ZeroR classifier counts the number of instances of each class value, in this case malicious or benign, determines the most frequent class value, and classifies all other data points as that value. A successful classifier should be able to outperform the ZeroR method, thereby indicating a target function, or target distribution, exists in the hypothesis space.

1.8 Research Questions

A set of research questions related to the hypothesis have been identified and prioritized as follows:

Research Question 1: Do malicious DLLs have distinct patterns of behavior in memory?

In order for a classifier to learn from data, there must be some underlying pattern between the data features and the classification values. The features are the memory artifacts that relate to DLLs in a Windows 7 memory image and the class values are either malicious or benign. Several features will be extracted from DLLs in memory. These will be analyzed to identify distinct patterns between malicious and legitimate DLLs. These patterns will be used to construct an initial feature set that can classify malicious DLLs in Windows 7 memory images.

Research Question 2: How do different classifiers perform against the dataset?

Machine learning algorithms are not perfect and as such they do not obtain 100% accuracy and 0% false positive or false negative rates on real data [66]. There is no previous research that has explored the classification of malicious software within a memory image.

A significant portion of this research will explore the performance of several classifiers against the dataset. Several classifiers have been chosen based on their popularity and success for classifying malware.

Research Question 3: Do correlation-based feature selection and gain ratio evaluation improve classifier performance for this dataset?

Feature selection algorithms provide a way to filter the features before a machine learning classifier learns from the data. Depending on the feature selection algorithm used, this may improve the performance of the model by eliminating redundant or irrelevant features. The success of feature selection algorithms depends on the algorithm itself, the data being used to learn the model, and the classifier being used.

Research Question 4: Can the ensemble learning technique Bagging improve performance over an individual classifier?

For a given dataset, different classification algorithms will perform at varying levels. Ensemble learning, applying multiple classifiers to a new instance of data, is an effective technique for improving performance. This helps alleviate the biases of certain classifiers on a given dataset and can provide a more robust prediction model. Combinations of classification techniques will be applied to determine which areas of performance can be improved using an ensemble approach.

CHAPTER 2

RELATED WORK

This chapter is an overview of related works in the areas of malware classification using static and dynamic features and malware detection in memory. It also provides details of the classification, attribute selection, and ensemble learning algorithms used in this work.

2.1 Malware Classification Using Static Features

Static features of a PE file are those that can be extracted without executing the program. They may originate from the file header or the disassembled code itself. Significant research has been done in the area of identifying static features that produce a high classification accuracy.

Byte n-grams are sequences of bytes of length n that are extracted from an executable [55, 3, 41, 54]. Methods using this type of feature tend to generate several thousand or greater attributes to use for the classifier. Some form of feature selection is usually necessary to identify the most useful set of n-grams in order to make the features manageable. This method does not provide meaningful information, but it can classify malware with a high accuracy [56].

PE features pertain to information stored in the header of a PE file [55, 15]. This can include [56]:

- Physical structure information such as file size, MAC times, or machine type
- Logical structure information such as linker version and section details
- Imported DLLs and API calls
- Exported functions (when the PE file is a DLL)
- Resources
- Version information

String features are any plain-text strings stored in the PE file. Schultz et al. [55] used string features to classify unknown malware and achieved greater accuracy than either byte n-grams or PE features.

Opcode n-grams are similar to byte n-grams, but they consist of the opcodes after a PE file has been disassembled [14, 25, 57, 8, 40]. An opcode can be a variable number of bytes depending on the instruction it represents. Karim et al. [karim2005] used opcode permutations to track malware evolution. Siddiqui [57] applied multiple data mining methods to opcde sequences to classify PE files. Opcodes have some of the highest reported accuracies when used to classify files as malicious or benign, but they do suffer from anti-disassembly techniques and packing obfuscations [22].

Function-based characteristics also require disassembly of a PE file as well as analysis to determine function boundaries. Menahem et al. [38, 39] used function characteristics such as the number of functions, size of the shortest and longest function, average function size, and standard deviation of function size as features to train classifiers.

API sequences are groupings of related API calls and have been examined by several researchers [5, 4, 52, 67, 68]. API call sequences can be thought of as API n-grams. Again

the PE file is disassembled and a control flow graph is generally constructed to identify the order of API calls and these are grouped as n-grams. Due to the large number of combinations a feature selection method is often employed to determine the API n-grams with the most value for classification.

Shabtai et al. [56] conducted a survey of research that classified files as malicious or benign using static features. Their research indicates that

- Previously unseen malicious code can be classified with high accuracy and a low false positive rate
- Multiple classifiers should be used
- OpCode and PE file representation patterns yield the best accuracy
- Classifier training should consider the imbalance problem of benign files to malicious files in real-life situations
- An active learning mechanism is recommended to ensure high detection accuracy against new threats

Portable executable files are composed of multiple sections. These sections are mapped to addresses when a file is loaded into memory and the values at these addresses may change during execution. This may render dynamic analysis infeasible for a PE file that has been extracted from memory. However, static analysis, and subsequently static feature extraction, are still viable [48]. Currently, no research explicitly addresses the extraction of static features from memory captures nor the application of a model derived from the aforementioned methods for detecting malware in memory.

2.2 Malware Classification Using Dynamic Features

Dynamic, or runtime, features are those extracted during the execution of malware. Examples of these include created files, network connections, and acquired handles to

system resources. The challenge in using these features is similar to using features from captured memory: ensuring the malware exhibits as much functionality as possible during execution. A memory capture represents the Windows operating system at a single point in time; in a sense, it is purely static. However, it contains data only generated at runtime for some programs. This allows some types of dynamic, or behavioral, features to be extracted.

Wilhelm and Chiueh [65] constructed a system called Limbo that extracted runtime features from kernel-mode rootkits and used them to build a Naive Bayes classifier. This system was designed to identify rootkits in real-time by extracting features and classifying the driver before it was loaded for execution. The authors identify 18 features that distinguish malicious kernel drivers from legitimate drivers. Each feature is of the form *kernel driver rootkits have property x or perform action y in greater or lesser quantity than legitimate drivers*. These types of features are meaningful and show that a classifier can be trained to identify malicious software effectively with a small number of distinctive features, as opposed to the use of n-grams which tend to generate thousands of features that hold no meaning to a human analyst. The authors reported a classification accuracy 96.2% with a false positive rate (malware classified as benign)of 1.4%.

Firdausi et al. [17] evaluated five different classifiers against a dataset of 220 malicious files and an unspecified amount of benign Windows XP system files. The classifiers were evaluated with and without feature selection. Of the five classifiers, J48 without feature selection performed the best in terms of accuracy (96.8%), precision (97.3%), and recall (95.9%). The authors did not discuss why J48 performed the best or why the other classifiers had worse performance.

2.3 Malware Detection in Memory

Scanning memory samples with antivirus software, particularly virustotal [62], has been advocated by practitioners [16, 23, 64] to help identify malware. Creating a signature for a given malware sample is a time-consuming process that often requires-reverse engineering the malware to identify key functionality. Even once the signature is created, polymorphic and metamorphic malware can alter itself sufficiently to evade detection by the signature [46].

Blacksheep [7] is a tool designed to detect kernel-level rootkits in groups of similar hosts. They capture memory from several machines at approximately the same the time and then look for features that appear in a small subset of the hosts. This enables the detection of kernel-level rootkits. The technique does not apply to user-level malware and requires memory captures from several similarly configured machines.

Maggi et al. [36] applied machine learning to detecting in-memory injections by analyzing system call sequences and arguments to the system calls on a live system. Their research focused on the FreeBSD operating system and shellcode injected into processes. It did not utilize memory captures, but instead relied on audit data obtained from the operating system.

2.4 Classification Algorithms

This section explains the selected classifiers used in this dissertation.

2.4.1 ZeroR

The ZeroR learner is one of the simplest classification algorithms. The model it constructs relies solely on the class variable, ignoring all input attributes. The ZeroR classifier simply predicts the majority class. It has previously been used as a baseline for comparison against other classifiers [59, 26]

2.4.2 Naive Bayes

Naive Bayes is a probabilistic classifier based on Bayes' rule. Naive Bayes considers each feature to contribute independently to the probability that an instance belongs to a specific class, regardless of any possible correlations between other features. Naive Bayes has previously been applied in malware-related classification research with good results [58, 10].

2.4.3 J48

The J48 algorithm is a Java implementation of the C4.5 release 8 algorithm [66]. The C4.5 algorithm is used to generate a decision tree, which can be used as a classifier. The C4.5 algorithm is based on information entropy. At each node of the tree, the C4.5 algorithm chooses the attribute of the data with the highest normalized information gain (gain ratio). The C4.5 algorithm repeats the same process on the sublists until one of the base cases is satisfied [47].

2.4.4 Support Vector Machine

Support Vector Machine (SVM) is a supervised learning method for two-group classification problems. It can be used for classification or regression. The SVM uses non-linear transformations to map an input space to a high-dimension feature space where it can construct a decision surface to separate the data points. The method employed to construct these decision surfaces ensures a high generalization ability [11].

2.4.5 Nearest Neighbor

The K-Nearest Neighbor (k-NN) method was selected as one of our experimental methods to represent instance-based learning algorithms. The k-NN method, the distance between the unknown data point and every known data point is computed. We used Normalized Euclidian distance for distance calculation.

2.4.6 Voted Perceptron

Voted perceptron stores all of the weight vectors created during the learning process. Each weight vector is allotted a number of votes based on its performance; in this case that is the number of instances that weight vector correctly classified before it had to be changed [18].

2.5 Correlation-Based Feature Selection

Correlation-Based Feature Selection method evaluates two major aspects of a feature, namely, the predictive ability and the degree of redundancy between features. In other

words, a feature is considered ”good” if it is highly correlated to the class variable and not highly correlated to any other feature [21].

2.5.1 Best-First Search

Best-first search was selected to use in conjunction with the Correlation-Based Feature Selection method. Best-first search is a simple heuristic search algorithm. It uses an evaluation function to calculate a score to each candidate node. The algorithm starts exploring the node with the best score first. Two lists are maintained by Best-first search algorithm, namely, a list of nodes to explore and a list of visited nodes. The algorithm will continue searching other nodes if it reaches a dead-end node.

2.5.1.1 Forward Selection

Forward selection starts with the empty set of features and searches forward by adding one feature at a time.

2.5.1.2 Backward Elimination

Backward elimination starts with the full set of attributes and searches backwards by eliminating one feature at a time.

2.6 Gain Ratio Attribute Selection

Gain Ratio Attribute Selection is based on information theory. It evaluates features by measuring their gain ratio with respect to the class. This method supports nominal class, binary class, and missing values.

2.6.1 Ranker Search

Ranker search sorts attributes or features by their individual evaluations. It can be applied to several attribute selection techniques that evaluate individual attributes. It was used in conjunction with Gain Ratio Attribute Selection in our experiment.

2.7 Bagging

Bagging or Bootstrap Aggregating is an ensemble learning method [9]. It creates separate instances of the fit dataset and creates a classifier for each instance. Bagging method then combines the results of all multiple classifiers by majority voting or averaging. The idea behind this method is that each instance of the fit dataset is different, therefore, each trained classifier perceives different focus and perspective on the problem.

CHAPTER 3

METHODOLOGY

This chapter describes the research methodology. It includes the configuration of the system used for generating data, how the dataset was produced, and how it was initially labeled.

3.1 Data Generation

Supervised machine learning requires sufficient data to learn a hypothesis that approximates a target function. In this research, the data consists of features extracted from memory images. Given the lack of publicly available Windows 7 x86 memory images, it was necessary to generate the samples used in this research.

3.1.1 Virtual Machines

Windows 7 Service Pack 1 x86 was used to create a virtual machine within VMWare. The OS was installed without any additional software. This virtual machine (VM) was copied twenty times. Each VM was saved, or snapshotted, in a clean state. The runtime state of each VM was slightly different in memory due to restarting between copying, but each VM contained the same OS and installed software.

3.1.2 Cuckoo Sandbox Architecture

Cuckoo Sandbox [12] is an open source automated malware analysis system written in Python. It is comprised of two components. The first component is the management system, or Cuckoo host, that is responsible for queuing, submitting, and reporting the results of processed malware samples. The second component is the guest agent that monitors the malware during execution. Figure 3.1 is the architecture of the dynamic analysis environment used in this research to generate the memory images.

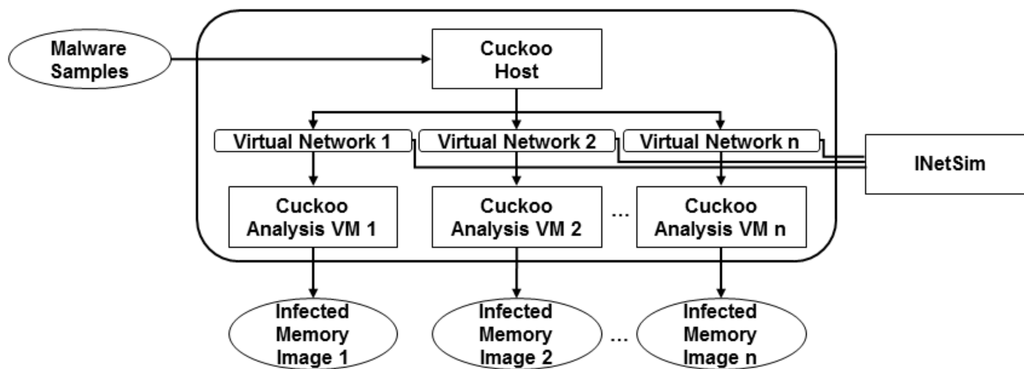

Figure 3.1

Data Generation Process

The Cuckoo host in this setup is a VM running Ubuntu 13.04. When a piece of malware is ready to be analyzed, the host boots a guest VM from its clean snapshot and transfers the file to it. The guest agent executes the file within the VM and records its behavior. The amount of time the malware runs can be configured; for this research the time was set to two minutes. During execution the guest agent can potentially capture the following:

- Windows API calls and native functions

- Files either created or deleted from the filesystem
- Memory dump of the guest VM
- Screenshots of the desktop
- Network traffic

Cuckoo Sandbox relies on API call hooking to capture information about the malware. In version 1.2.1 Cuckoo Sandbox hooks 170 different API calls and native functions. If the malware uses a call not in this list, then that data will not be recorded. An advantage of studying the memory dumps of these samples is that, provided the malware executed properly, there will always be data available for analysis. Any data generated by the guest agent will sent back to the Cuckoo host where it is encapsulated in a .tar.gz file. This file is referred to as a *Cuckoo Sample*.

3.1.3 INetSim

All network traffic is directed to a VM running Ubuntu 13.04 and InetSim [24]. INetSim simulates several network services such as http, https, ftp, dns, and smtp. Network traffic will not be able to reach any nodes beyond the system running INetSim. As such, some malware will not execute fully or may terminate itself before any meaningful malicious code is executed. For instance, any malware part of a botnet that needs to connect to a command and control server to acquire updates or instructions will not be able to do so and will likely self-terminate. The reason that INetSim is used instead of disabling networking altogether is that some malware will exhibit additional behavior if it can detect there is a network connection available or that a specific network service exists.

3.1.4 Malware Samples

The malware samples used by Cuckoo Sandbox to generate the *Cuckoo Samples* were obtained from VirusShare [61] in 2012. The samples downloaded were the most recent uploads at the time and do not include any specific filtering (i.e. file type, size, etc...). The file types contained within the samples include executables, shared libraries, drivers, pdfs, html, vb scripts, and various other types.

Any samples that did not simply run and quit are considered valid for the purpose of this research. Determining which samples did more than just start and quit can be determined by analyzing the saved results from Cuckoo Sandbox. Those that quit early will have fewer API calls and minimal system interaction.

3.2 Extracting DLLs from Memory

Volatility [63] was used to extract DLLs and their associated artifacts from each memory image generated by Cuckoo Sandbox. Each process contains three doubly linked lists that contain the DLLs loaded by the process. However, some malware will use a technique known as direct kernel object manipulation (DKOM) where it removes itself from the doubly linked lists of _LDR_DATA_TABLE_ENTRY structures. In order to identify all of the loaded DLLs within a process, the virtual address descriptor (VAD) tree is examined. The VAD tree consists of VAD nodes. Each node describes the allocated virtual memory ranges for a given process. If a memory range also maps a file, such as when a DLL is loaded into the memory space of a process, the DLL's file name and path will also be included.

Volatility is used to traverse the VAD tree identifying any nodes with a memory mapped file. Once a node is identified, its corresponding entries in the three doubly linked lists are also extracted if they are present. This provides information from both the _MMVAD and _LDR_DATA_TABLE_ENTRY data structures. The possibility does exist that malware can unlink a node in the VAD tree, rendering it invisible to identification in this manner. However, this action cannot be performed without the possibility of leaving the operating system in an unstable state, potentially resulting in a system crash [13]. The features extracted for each DLL are detailed in chapter 5.

3.3 Data Cleaning

The VAD tree defines all of the memory ranges allocated by a process including heaps, thread stacks, DLLs, and other mapped files. Identifying DLLs using the method in the previous section also captures files that are not actually DLLs. These include .exe and .mui files. The executable file of an _EPROCESS is in the list of modules loaded by a process and therefore also defined in the VAD tree. The executable will appear in two of the doubly linked lists, *InLoadOrderModuleList* and *InMemoryOrderModuleList*, but not in the third list, *InInitializtionOrderModuleList*. Any files in the dataset that end with '.exe' and appear in the load order and memory order lists, but not the initialization order list is considered a process executable and removed from the dataset.

The .mui files are language-specific files associated with a language-neutral (LN) file, typically a DLL. These files do not appear in any of the doubly linked lists associated with the _EPROCESS structure. Any files with an extension of '.mui' and that do not appear

in all three of the doubly linked lists are considered .mui files and are removed from the dataset.

3.4 Data Labeling

The learning method in this research is Supervised Learning. This requires the data to be accurately labeled. This section details how each individual data point is classified.

Each DLL and its associated features are written out as comma separated value files. This results in a large set of data containing malicious and legitimate DLLs with the associated information listed above. Each data point falls into one of four categories.

Legitimate processes containing legitimate DLLs (LL) - The majority of the processes and DLLs running on the system will be of this type. These were running before the malware executed and continue to run after. Each memory image produced will contain, mostly, the same set of processes and DLLs.

Legitimate processes containing malicious DLLs (Injected) - These are the processes that have been subject to DLL injection.

Malicious processes containing legitimate DLLs (ML) - Many malware samples will be executed directly by cuckoo resulting in a malicious process. Malicious processes, like legitimate processes, require legitimate DLLs to perform basic tasks on the system. There is a small subset of legitimate DLLs that are required for any process to initialize and run. This set presents the greatest potential variety of legitimate DLLs on the system.

Malicious processes containing malicious DLLs (MM) - Malware authors create DLLs for the same reasons that legitimate DLLs are created: code reuse. Some malware will contain its own DLLs for this purpose. These are not an example of DLL injection.

Only the second category, *legitimate processes containing malicious DLLs*, were used in the datasets. The *ML* and *LL* sets did not contain enough variation to be considered representative of what might be found on an arbitrary Windows 7 system. This is partially due to the lack of third party programs and their associated DLLs, but also to the number of system DLLs that were not present in memory. The legitimate DLLs extracted during this process were mostly duplicates. A set of textitML DLLs extracted from 4,230 memory images had 162,567 legitimate DLLs. The number of unique DLLs within this set was only 465. The base install of Windows 7 SP1 x86 that was used for analysis by Cuckoo contained 6,121 .dll files; only 7.6% of the system DLLs were represented across these memory images.

3.5 Dataset Creation

This section details the process used to create the datasets used for experimentation.

The initial dataset was created from 3,240 *cuckoo samples*. These *cuckoo samples* yielded 299 instances of malicious injected DLLs. This set of malicious DLLs was analyzed to construct the feature set defined in chapter 5. After the features were specified, this dataset was discarded.

A set of 33,160 *cuckoo samples* were generated and the previously determined features were extracted from the memory images. This yielded 2,385 instances of malicious in-

jected DLLs. These were analyzed in conjunction with legitimate DLLs from the analysis system and the characteristics reported in [20].

Twelve memory images were created from three different installations of Windows 7 SP1 x86. For each memory image, several legitimate pieces of software were downloaded, installed, and executed. Memory was captured while these applications were running. The software used to construct these memory images was all freely available software downloaded from the Internet.

The 2,385 injected DLLs were combined with a random sampling from nine of the manually created memory images. This formed the cross-validation dataset.

To further test the generality of each learning model, another set of 8,371 *cuckoo samples* were processed. 424 malicious injected DLLs were obtained from these samples. The remaining three manually created memory images were randomly sampled to obtain the legitimate DLLs. These were combined to form the test set.

Table 3.1 details the class distribution for the cross-validation and test datasets.

Table 3.1

Class Distribution for the Cross-Validation and Test Datasets

	Cross-Validation	**Test**
Total Instances	12,066	2,694
Total Malicious DLLs	2,385	424
Total Legitimate DLLs	9,681	2,270
Unique Malicious DLLs	812	188
Unique Legitimate DLLs	1,076	900
Malicious DLL (%)	19.77%	15.74%
Legitimate DLL (%)	80.23%	84.26%

CHAPTER 4

PRELIMINARY RESULTS

This chapter presents preliminary results that address the first research question for this dissertation:

Do malicious DLLs have distinct patterns of behavior in memory?

This work was published in Advances in Digital Forensics XI [20].

4.1 Injected DLL Characteristics

This section discusses various characteristics of the injected DLLs we identified. These characteristics are drawn from the set of injected DLLs and contrasted with those of legitimate DLLs where deemed appropriate.

We generated 33,160 *cuckoo samples* for this research. Of those, 955 exhibited DLL injection behavior. It is plausible that more than the 955 identified perform DLL injection, but did not in this instance due to a lack of some required resource e.g. configuration files, Internet connection, installed software. The analyzed data was split into two subsets. The first set was all of the injected DLLs and all of the legitimate DLLs. These contained 2,385 and 162,567 DLLs respectively. The second set consisted of the unique injected and legitimate DLLs for a given memory image. These contained 1,168 and 152,883 DLLs respectively.

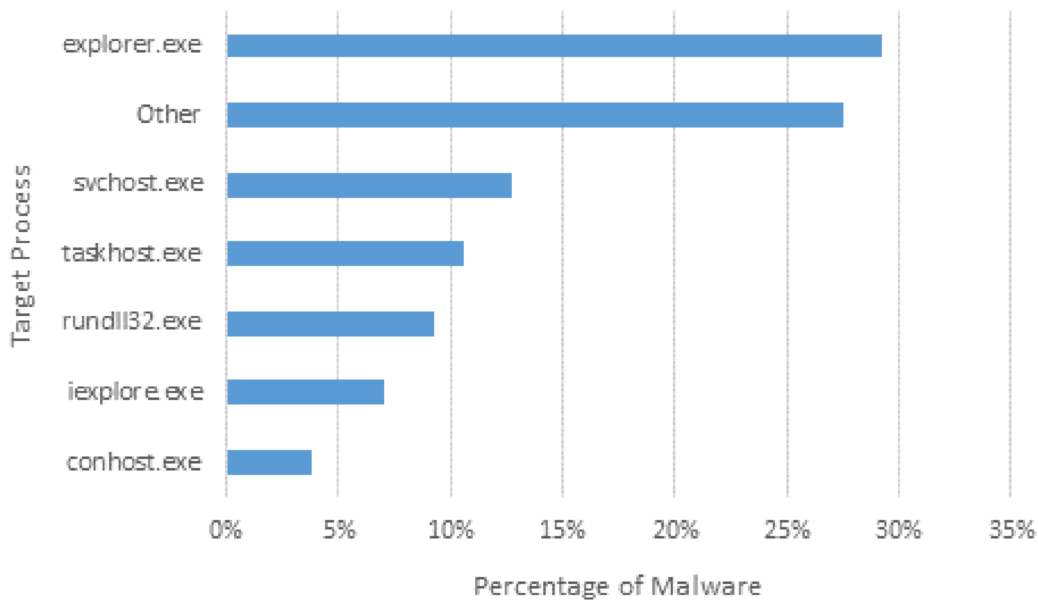

Figure 4.1

Processes Commonly Targeted for DLL Injection

4.1.1 Target Processes

Malware that injects DLLs must specify a target process to host the malicious code. In the dataset, some processes were more common than others. Explorer.exe, svchost.exe, and taskhost.exe were the most common processes targeted for injection, accounting for over 52% of all injections. Each of these processes should always be running on a Windows system. Also, each of them presents a large and/or varied set of DLLs at runtime. Explorer.exe had, on average, 210 legitimate DLLs loaded at one time and there are often several instances of svchost.exe and taskhost.exe running on a system. Figure 4.1 shows the numbers of injections.

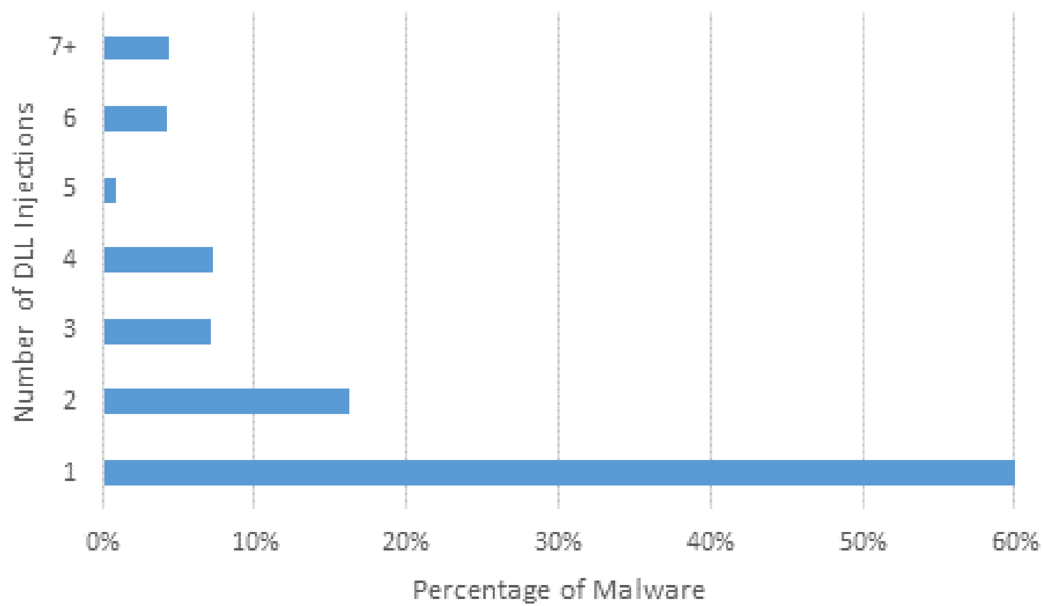

Figure 4.2

Distribution of DLL Injections per Malware

4.1.2 Number of Injections per Malware

Malware can choose to inject a DLL into any running process. Injecting into multiple processes can provide the malware with greater survivability and versatility, but may also increase chances of being detected. 955 malware samples injected a DLL into a process. Of the 955, 60% of them (573) targeted a single process with the remaining 40% injecting into two or more processes. Figure 4.2 shows the distribution of the number of injections performed by malware.

4.1.3 Simultaneous Loads

When malware injects a DLL into several processes, it will often iterate through the active processes and inject the DLL as it finds its target(s). For the malware that loaded into

multiple processes we examined the load time extracted from the associated _LDR_DATA_TABLE_ENTRY to see how many DLLs had the same approximate load time. For a given DLL within a memory image, the load time of that DLL is compared against its load time in all the other processes containing that DLL. For every load time within one second, that DLL's simultaneous load value is incremented by one. If the DLL appears in multiple processes but does not share a load time, then it has the value 0. If a DLL exists only once within an entire memory image, it has a value of -1. Of the DLLs that exist in more than one process (those with a value of 0 or greater) the number of DLLs we detected that has at least one simultaneous load for injected DLLs was 73.3%. For legitimate DLLs, it was 45.4%. This shows that malicious processes tend to be injected into multiple processes at the same approximate time whereas the load times of legitimate DLLs are more varied.

4.1.4 Load Position

The *InLoadOrderModuleList* is a doubly linked list of _LDR_DATA_TABLE_ENTRY structures. The ordering of the list is based on the order in which a DLL was loaded into a process with the executable occupying the first position. The beginning entries will be occupied with dynamically linked DLLs named in the Import Address Table. DLLs loaded at runtime will naturally appear at the end of the list. Some loaded DLLs are volatile in the sense that they are loaded and unloaded repeatedly during the lifetime of a process while others are more stable, remaining loaded for longer periods of time. The load position was calculated for each DLL that existed in the *InLoadOrderModuleList*. The average load position was calculated for both injected and legitimate DLLs. The average load position

for all of the injected DLLs was 83.7. The average load position for the legitimate DLLs was 52.6. We note that our system ran for a short period of time and this may affect the reliability of this result. Depending on the number of DLLs unloaded by a process, an injected DLL may appear closer to the beginning of the list.

4.1.5 Init Position

Similar to the *InLoadOrderModuleList*, the *InInitializationOrderModuleList* represents the order in which a DLL's DLLMain function was executed. The average init position for all of the injected DLLs was 87.4. The average init position for legitimate DLLs was 51.3. This result may also be affected by the short run time of the analysis system.

4.1.6 Base Address

The base address is the virtual address in a process where a DLL is loaded. The default base address for DLLs is 0x10000000. Since a process will normally contain multiple DLLs and only one DLL can occupy a given virtual address within a process, many DLLs will contain a .reloc section in the PE header that specifies how to translate its offsets. 48% of the unique injected DLLs were loaded at the virtual address 0x10000000. This is contrasted sharply against the legitimate system DLLs of which 99.99% were loaded at an address other than 0x10000000.

4.1.7 Exported Function Count

The number of functions exported by each DLL were extracted and used to calculate the mean and mode for injected and legitimate DLLs. The injected DLLs exported consid-

erably fewer functions on average with a mode of 2 and a mean of 13. Legitimate DLLs had a mode of 11 and a mean of 368.

4.1.8 Imported Function Count

The number of functions imported by each DLL were extracted and used to calculate the mean and mode for injected and legitimate DLLs. The mode of each type of DLL was similar with injected having 213 and legitimate 198. The mean was 115 and 257 for injected and legitimate DLLs respectively.

4.1.9 Loaded from Temp

A common heuristic when looking for malware is searching for binaries loaded from a temporary directory such as %TEMP% (C:\Users\UserName\ AppData\Local\Temp). 20% of the unique injected DLLs were loaded from a directory with 'temp' in the path. Nearly all of these were from %TEMP% but a small number were from directories named C:\temp or C:\Windows\temp.

4.1.10 Load Paths

Figure 4.3 shows the most common load paths for injected legitimate DLLs. For legitimate DLLs, 97% are loaded from C:\Windows\System32 or C:\Windows\ System32\en-us.

4.1.11 COM Server

The Component Object Model (COM) is an interface standard used in the Windows operating system. It enables software to call code hosted by other software components

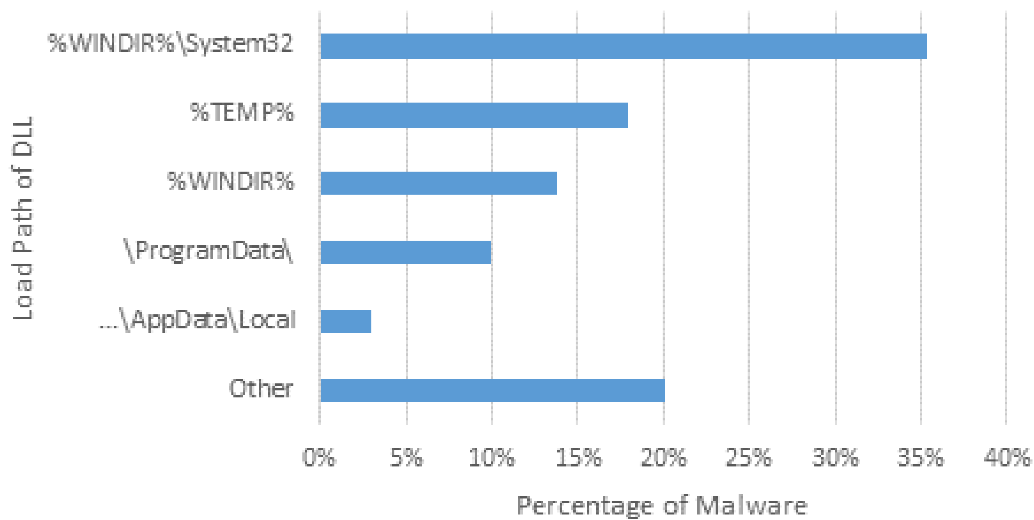

Figure 4.3

Distribution of Load Paths per Malware

without in-depth knowledge of its implementation. The calling component is referred to as the client and the hosting component is referred to as the server. Malware authors sometimes leverage the COM infrastructure to implement malicious code [22]. A COM server is required to export at least two Windows API functions: *DllGetClassObject* [1] and *DllCanUnloadNow* [1]. If these two API calls are seen in the exports of a DLL then it is considered to be a COM server. 5.2% of the unique injected DLLs were implemented as COM servers.

4.1.12 COM Client

Windows binaries can call COM objects as clients. In order to use COM objects, the binary must call *OleInitialize* [1] or *CoInitializeEx* [1] functions. Any DLLs importing

either of these functions were considered to be COM clients. Only 2.1% of the unique injected DLLs in our dataset were capable of calling COM objects.

CHAPTER 5

FEATURE SET

This chapter details the features used to construct the dataset. There are two types of features described here. The first are raw features extracted directly from memory. The other type of features are transformed features. They are derived from one or more existing features. Features are organized by the data structures in memory where they originate. The feature set used consists of 32 features; 20 of the features are nominal and 12 are numeric. A descriptive name of each feature is provided and the feature as it appears in the actual dataset is in parenthesis next to it.

5.1 _EPROCESS Features

Process_name is a string representing the name of the process that contains the DLL. The values for this feature are nominal values: explorer.exe, svchost.exe, taskhost.exe, and other.

Num_processes_with_dll is an integer of the number of processes where a specific DLL was found.

5.2 _LDR_DATA_TABLE_ENTRY Features

These features are either extracted from the _LDR_DATA_TABLE_ENTRY, or use information related to it to determine the value for the feature.

Dll_base is the address in memory where the DLL was loaded. Indicates if the DLL was found at the default base address (0x10000000) in the process address space. The value is 'True' if it was, 'False' if it was not.

Load_count is an integer that indicates the number of times LoadLibrary or LdrLoadDll was called for the DLL. The value is incremented each time one of these functions is called and decremented when the DLL is unloaded from a process. If a DLL was loaded from an import address table in a module, the value will be 0xFFFF (-1). The values are a base 10 integer representation of the hex value.

Dll_in_load is a binary value that specifies if the DLL was found in the *InLoadOrderModuleList*. Value is 'True' if the DLL was found in the list, 'False' if it was not.

Dll_in_mem is a binary value that specifies if the DLL was found in the *InMemoryOrderModuleList*. Value is 'True' if the DLL was found in the list, 'False' if it was not.

Dll_in_init is a binary value that specifies if the DLL was found in the *InInitializtionOrderModuleList*. Value is 'True' if the DLL was found in the list, 'False' if it was not.

Load_path_is_temp is the location on disk from where the DLL was loaded. Due to the variety of values this attribute could describe, it was transformed into a binary attribute. A common heuristic used for identifying malware is if it was launched or loaded from a temporary directory. The value for this feature is 'True' if the path contains temp, 'False' if it does not.

Load_position is a floating point value describing the position of a DLL in the *InLoadOrderModuleList*. DLLs are added to this list based on the order in which they are loaded by the process. The value is calculated as:

$$load_pos = \frac{DLL\,position}{total\ DLLs\ in\ list} \tag{5.1}$$

Init_position is a floating point value describing the position of a DLL in the *InInitializationOrderModuleList*. DLLs are add to this list based on the order in which their DLLMain function is called. The value is calculated as:

$$init_pos = \frac{DLL\,position}{total\ DLLs\ in\ list} \tag{5.2}$$

Is_dkom_present is a binary value determined by the absence of a DLL from one of the three doubly linked lists. Value is 'True' if the DLL is not in all three lists, 'False' if it is in all three lists.

File_extension_is_dll is a binary value. The value is 'True' if the extension is .dll, 'False' if it is anything else.

Size_of_image is an integer specifying the number of bytes of the DLL.

Sim_loads is an integer. The LDR_DATA-_TABLE_ENTRY contains a load time for each DLL. This feature is an integer that expresses the number of times a DLL was simultaneously loaded into different processes. For this feature, simultaneously is defined as having a load time within one second of another load time. For a given DLL, its load time is compared to the other instances of it across all processes to calculate the number of times it was simultaneously loaded.

5.3 _MMVAD_* Features

Vad_allocated_pages is a positive integer that specifies the number of pages allocated for the DLL.

Vad_cf_accessed is a binary value that describes if the control flag 'Accessed' is enabled. The value is 'True' if the flag is enabled, 'False' if it is not.

Vad_cf_image is a binary value that describes if the control flag 'Image' is enabled. The value is 'True' if the flag is enabled, 'False' if it is not.

Vad_protection is a nominal feature that lists the protection level applied to the allocated memory region associated with the DLL when it was first initialized. If subranges of the allocated region are later changed, it will not be reflected through this value. The values in these datasets are 'PAGE_EXECUTE_WRITECOPY', 'PAGE_READONLY', and 'PAGE_WRITECOPY'.

Vad_type is the type of the VAD node. The values in these datasets are 'VadImageMap' and 'VadNone'.

Vad_num_mapped_views is a positive integer specifying the number of times a view of a file has been mapped. A view of a mapped file is a virtual address range that represents a segment of the mapped file.

Vad_num_section_refs is an integer for the references to the section object of the VAD node.

Vad_commit_charge is an integer of the number of memory pages that were committed in the memory range.

Vad_path_ldr_path_differ is a binary feature. If the mapped path listed in the VAD node differs from the path listed in the LDR_DATA_TABLE_ENTRY in the *InLoadOrderModuleList*, this value is 'True', otherwise it is 'False'.

5.4 _PE_HEADER Features

The features in this section are all extracted from the PE header of a DLL in memory. PE features have previously been used to classify software as malicious or benign.

The PE header begins at the base address where a DLL is loaded. It has the same structure in memory as on disk. However, in some instances the value for a given feature is not obtainable due to the underlying physical memory being paged to disk.

Num_exports is a positive integer representing the number of functions exported by a DLL.

Is_com_server is a binary feature. A COM server is required to export at least two Windows API functions: DllGetClassObject and DllCanUnloadNow. If these two API calls exported by the DLL, then the value is 'True', otherwise the value is 'False'.

Num_imports is a positive integer representing the number of functions imported by a DLL.

Is_com_client is similar to COM server. A COM client is required to import specific Windows API functions. The imported functions of a DLL are parsed and if it contains either OleInitialize or CoInitializeEx then the value of this feature is 'True', otherwise the value is 'False'.

Dll_flag_set is a binary feature. The PE header contains a field that specifies if the file is actually a DLL. The value is ’True’ if the flag is set, ’False’ if it is not set.

Is_entrypoint_section_name_text is a binary feature. The entrypoint of a PE file is the address where the code begins execution. Often the name will be .text, though any name is valid. The value for this feature is ’True’ if the name of the section containing the entrypoint is .text, ’False’ if it is anything else.

Has_reloc_section is a binary feature that specifies if a .reloc section exists in the PE header. The value is ’True’ if the DLL has a .reloc section, ’False’ if it does not.

Is_PE_image_base_default is a binary feature specifying if the base address where the DLL should be loaded is the default. The value is ’True’ if the image base is 0x10000000, ’False if it is not.

Raw_virt_size_diff is a nominal feature. Each section in a PE file has a raw size (size on disk) and a virtual size (size in memory). Large differences between these two sizes sometimes indicate that a PE file is packed. For the section containing the address of the entrypoint, this feature is the raw size subtracted from the virtual size. The values of this feature are ’Neg’ if the difference between the two is less than zero, ’None’ if the sizes are equal, ’Low’ if they are within one page (4096 bytes) of each other, or ’High’ if the size difference is greater than one page.

CHAPTER 6

EXPERIMENTAL RESULTS

This chapter presents the results of the experiments for each research question. All experiments were conducted using Weka 3.6.11. Unless otherwise specified, the default settings for each algorithm in Weka were used. Each experiment was run against two sets of data: the cross-validation set and the test set. The two metrics of interest in these experiments are accuracy and false positive rate. Accuracy is the set of correctly classified instances out of the total instances available. False positive rate refers to malicious DLLs that have been classified as legitimate.

Evaluation against the cross-validation set used ten iterations of stratified 10-fold cross-validation. An average accuracy and false positive rate is produced for each iteration; these averages are then averaged together to produce the final results. The standard deviation column represents the deviation among iterations. Evaluation for the test set constructed a model from the entire training set and then applied it to the test set. Statistical significance between metrics was determined using a two-tailed paired t-test [44]. The comparison is explained in each section where appropriate.

6.1 Research Question 2

How do different classifiers perform against the dataset?

The experiment for this research question was designed to evaluate different categories of classifiers against the dataset. ZeroR was used to establish a baseline for comparison between the classifiers; the specific metrics of interest are accuracy and false positive rate. The following classifiers were evaluated using the cross-validation set and the test set:

- ZeroR
- Naive Bayes Classifier
- Support Vector Machine
- Voted Perceptron Learning Algorithm
- Nearest Neighbor with Euclidian Distance
- J48 Decision Tree

6.1.1 Base Classifier Evaluation - Cross-Validation Set

Table 6.1 shows the accuracy and false positive rate of each classifier against the cross-validation dataset. Figure 6.1 and Figure 6.2 depict the accuracy and false positive rate respectively.

ZeroR and the Voted Perceptron Learning Algorithm (VPLA) performed the worst of all six classifiers with respect to both accuracy and false positive rate. The results from VPLA were identical to that of ZeroR. The best model it could produce classified every instance as legitimate. The other four classifiers performed significantly better than ZeroR with J48 obtaining the highest accuracy and lowest false positive rate.

6.1.2 Base Classifier Evaluation - Test Set

Table 6.2 shows the accuracy and false positive rate of each classifier against the test dataset. Performance of the classifiers using the test set was similar to that of the cross-

Table 6.1

Base Classifier Accuracy and False Positive Rate - Cross-Validation Set

Classifier	Accuracy	Significant	Std Dev	FP Rate	Significant	Std Dev
ZeroR	80.23	N/A	0.04	100.00	N/A	0.00
NaiveBayes	91.82	Yes	2.59	6.11	Yes	0.02
SVM	98.38	Yes	0.33	3.91	Yes	0.01
Voted Perceptron	80.23	No	0.04	100.00	No	0.00
Nearest Neighbor	98.74	Yes	0.29	3.75	Yes	0.01
J48 Decision Tree	99.24	Yes	0.26	2.45	Yes	0.01

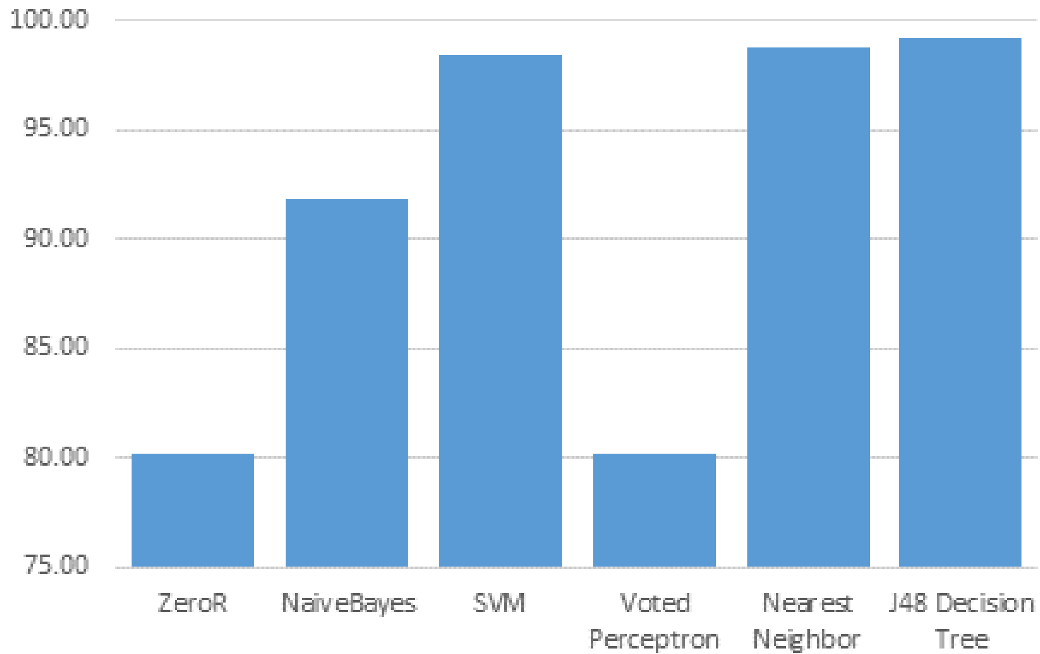

Figure 6.1

Classifier Accuracy - Cross-Validation Set

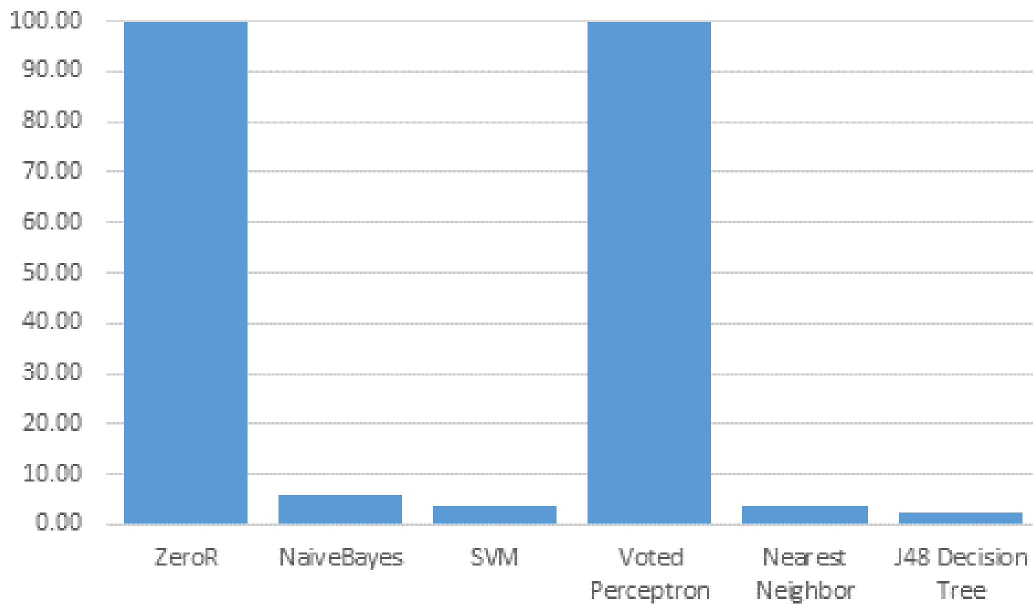

Figure 6.2

Classifier False Positive Rate - Cross-Validation Set

validation set. ZeroR and VPLA again performed identically. The remaining classifiers each achieved better accuracy and false positive rates than the baseline with J48 obtaining the highest accuracy and lowest false positive rate. The increased performance of the four classifiers was statistically significant.

6.2 Research Question 3

Do correlation-based feature selection and gain ratio evaluation improve classifier performance for this dataset?

The experiments associated with this research question are designed to identify which features are considered more valuable and how they affect the chosen set of classifiers. The ZeroR classifier was not used during these experiments as feature selection has no impact on the learning model it constructs.

Table 6.2

Base Classifier Accuracy and False Positive Rate - Test Set

Classifier	Accuracy	Significant	FP Rate	Significant
ZeroR	84.26	N/A	100.00	N/A
NaiveBayes	91.80	Yes	9.90	Yes
SVM	97.59	Yes	5.70	Yes
Voted Perceptron	84.26	No	100.00	No
Nearest Neighbor	98.00	Yes	7.80	Yes
J48 Decision Tree	98.74	Yes	3.50	Yes

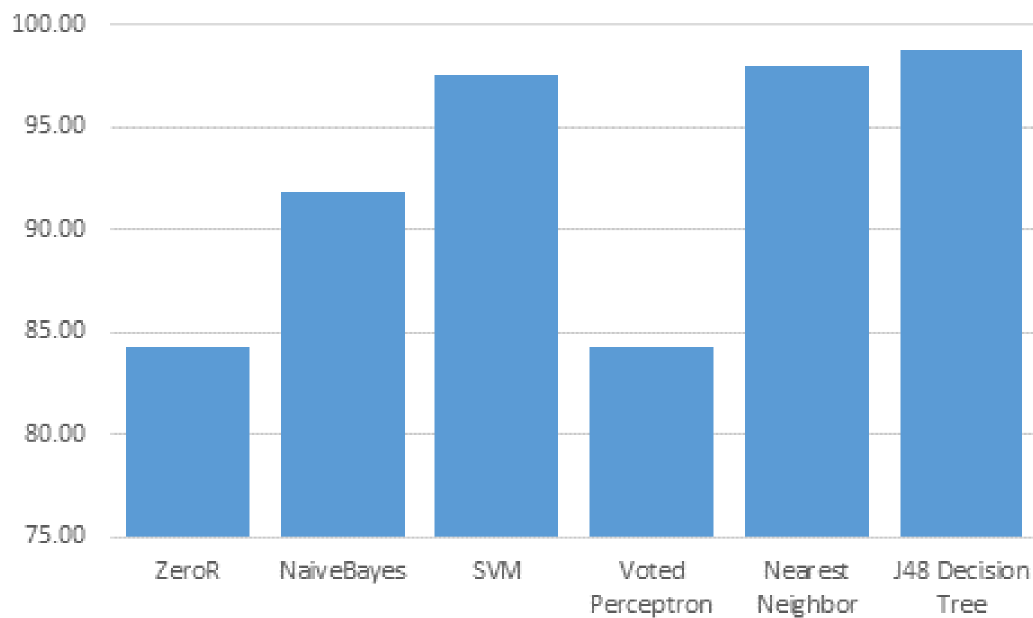

Figure 6.3

Classifier Accuracy - Test Set

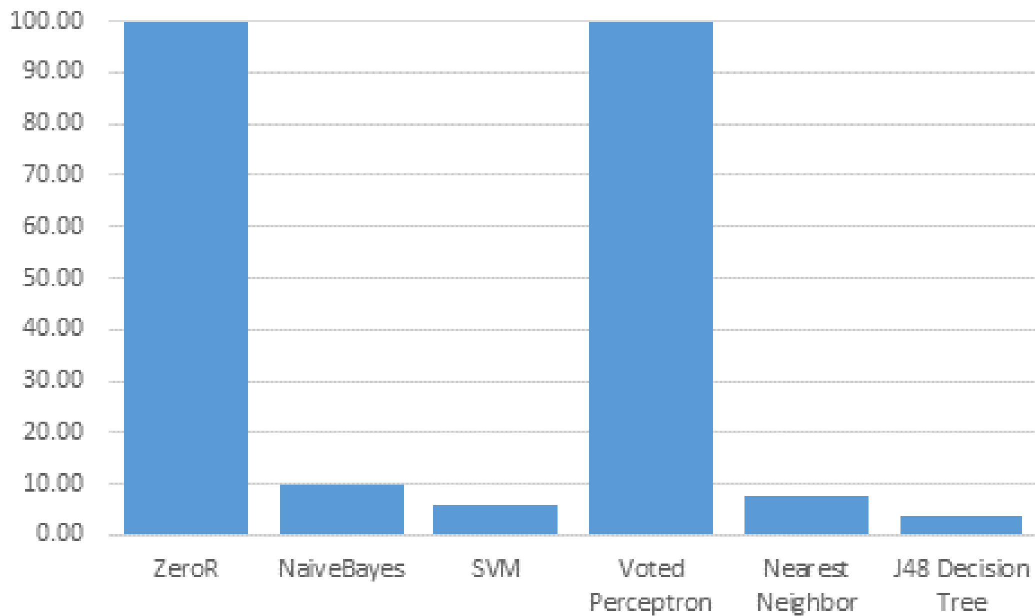

Figure 6.4

Classifier False Positive Rate - Test Set

6.2.1 Correlation-based Feature Selection

Correlation-based Feature Selection (CFS) using Best-First search with forward selection and backward elimination was applied to the training set. Table 6.3 lists the subset of features selected for each search technique. The subsets are identical. These features are each highly correlated with the class and not highly correlated with each other.

Weka provides a meta-classifier that incorporates a feature selection algorithm into the learning process. This meta-classifier was used to evaluate how the CFS algorithm affected the specified classifiers. This ensures cheating does not occur during the learning process.

Table 6.3

CFS using Best First Search Attribute Subsets

Forward Selection	**Backward Elimination**
dll_base	dll_base
entrypoint_section_name	entrypoint_section_name
init_position	init_position
load_count	load_count
load_path_is_temp	load_path_is_temp
PE_image_base	PE_image_base
vad_allocated_pages	vad_allocated_pages

6.2.1.1 Classifier with CFS - Cross-Validation Set

Table 6.4 lists the results of the five classifiers augmented with CFS on the Cross-Validation data set. The 'Significant' column is the significance of the metric using CFS compared to that of just the classifier.

Figure 6.5 and Figure 6.6 represent the performance difference for accuracy and false positive rate respectively.

Naive bayes showed the only improvement in accuracy. Because Naive Bayes assumes independence across all attributes in the input space, its performance suffers from redundant attributes in the dataset. The CFS algorithm is designed to select subsets of features that are not highly correlated with each other (i.e. not redundant). Naive bayes also had a lower false positive rate. Using CFS, Naive Bayes misclassified 5.57% of the malicious DLLs as legitimate in the cross-validation set.

Nearest neighbor exhibited better performance for both accuracy and FP rate. Euclidian distance was used as the distance measure. Noisy attributes have a strong affect on this as

Table 6.4

Classifiers using CFS Attribute Selection - CV Set

Classifier	**Accuracy**	**Significant**	**Std Dev**	**FP Rate**	**Significant**	**Std Dev**
NaiveBayes	95.57	Yes	0.51	5.57	No	0.02
SVM	97.15	Yes	0.50	7.33	Yes	0.02
Voted Perceptron	72.78	Yes	1.11	74.16	Yes	0.04
Nearest Neighbor	98.58	No	0.39	2.77	Yes	0.01
J48 Decision Tree	98.91	Yes	0.29	3.51	Yes	0.01

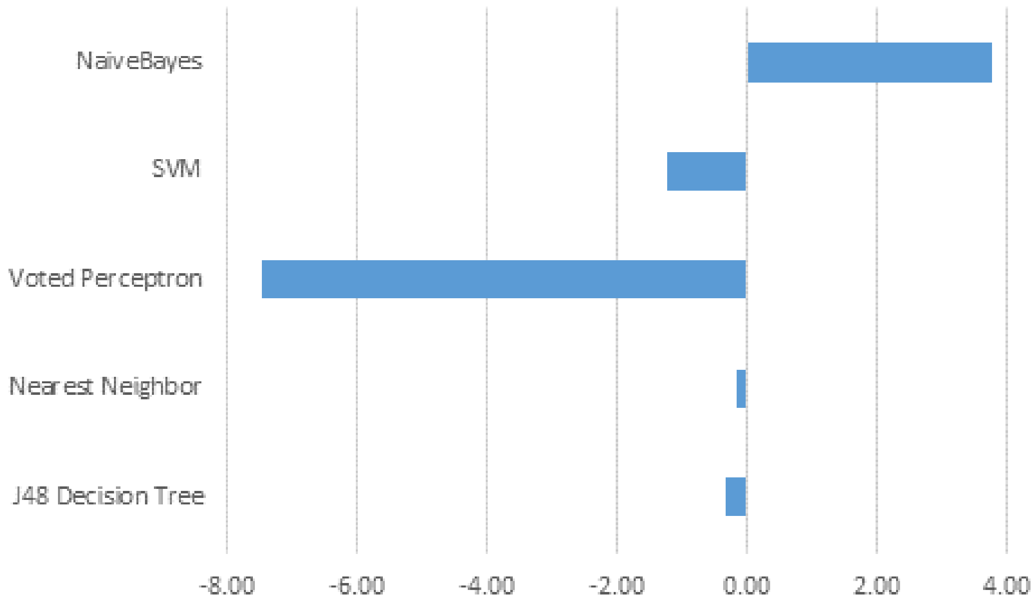

Figure 6.5

Classifier Accuracy Change using CFS - Cross-Validation Set

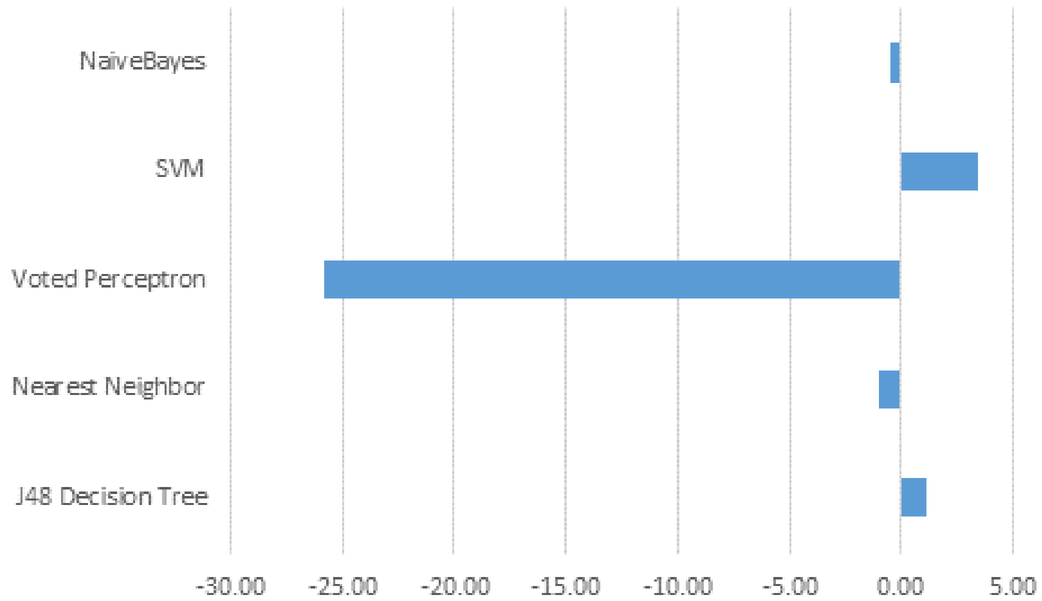

Figure 6.6

Classifier FP Rate Change using CFS - Cross-Validation Set

it weights each attribute equally. The subset chosen by CFS should contain only relevant features.

6.2.1.2 Classifier with CFS - Test Set

Table 6.5 lists the results of the five classifiers augmented with CFS on the test data set. The 'Significant' column is the significance of the metric using CFS compared to that of just the classifier.

Figure 6.7 and Figure 6.8 represent the performance difference for accuracy and false positive rate respectively. For the test set, Naive Bayes was the only classifier with an increased accuracy. Its false positive rate was worse than without CFS, but this was not considered statistically significant.

Table 6.5

Classifiers using CFS Attribute Selection - Test Set

Classifier	Accuracy	Acc Significant	FP Rate	FP Significant
NaiveBayes	94.32	Yes	7.3	No
SVM	96.66	Yes	7.3	No
Voted Perceptron	82.70	Yes	70	Yes
Nearest Neighbor	97.34	No	4.5	Yes
J48 Decision Tree	98.22	Yes	5	Yes

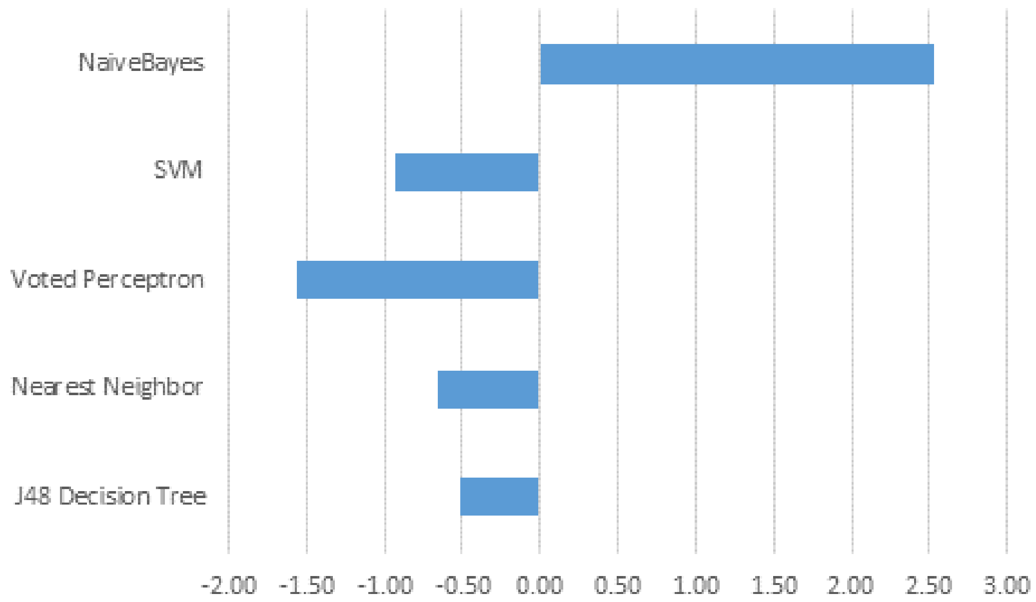

Figure 6.7

Classifier Accuracy Change using CFS - Test Set

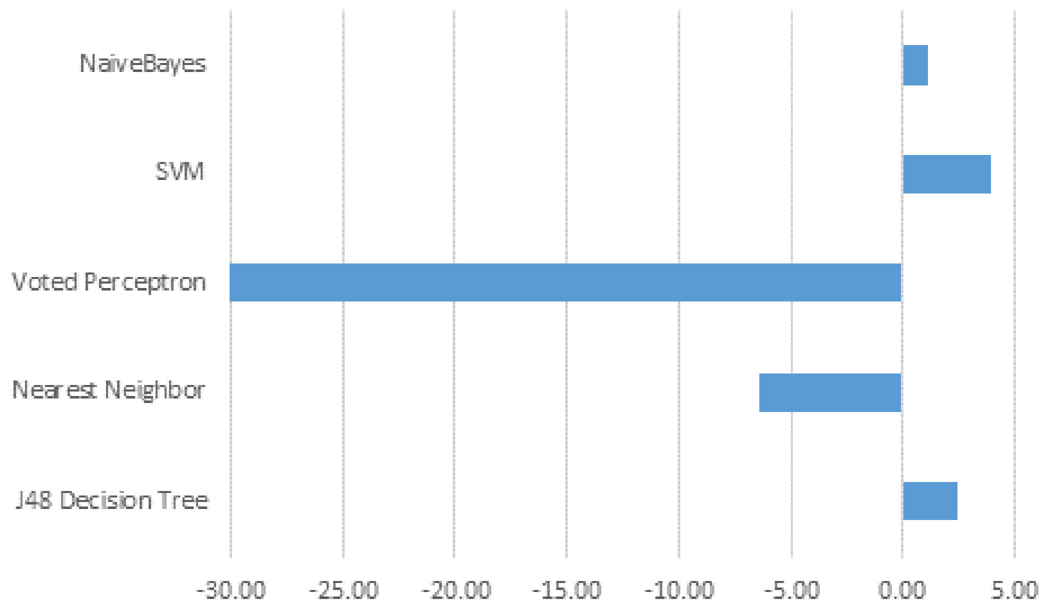

Figure 6.8

Classifier FP Rate Change using CFS - Test Set

6.2.2 GainRatio Attribute Evaluation using Ranker

Gain Ratio is a variation of the Information Gain algorithm often employed in constructing decision trees. The J48 decision tree used in Weka uses Gain Ratio to determine the structure of the tree. The top seven ranked attributes were used for classification and are listed in Table 6.6. The attributes listed are those that represent the greatest gain ratio relative to the class attribute.

6.2.2.1 Classifier with GainRatio Selection - Cross-Validation Set

Table 6.7 contains the results of each classifier using GainRatio on the cross-validation dataset. The 'Significant' column states if the change of the associated metric was statistically significant.

Table 6.6

Gain Ratio Attribute Selection using Ranker Search

Rank	**Feature**
1	dll_base
2	PE_image_base
3	raw_virt_size_diff
4	load_path_is_temp
5	entrypoint_section_name
6	file_extension_is_dll
7	has_reloc_section

Table 6.7

Classifiers with Gain Ratio - Cross-Validation Set

Classifier	**Accuracy**	**Significant**	**Std Dev**	**FP Rate**	**Significant**	**Std Dev**
NaiveBayes	93.88	No	1.29	11.59	Yes	0.02
SVM	96.42	Yes	0.71	6.11	Yes	0.02
Voted Perceptron	92.96	Yes	6.07	24.97	Yes	0.36
Nearest Neighbor	96.98	Yes	0.98	3.89	No	0.01
J48 Decision Tree	95.87	Yes	0.70	7.48	Yes	0.03

Figure 6.9 and Figure 6.10 represent the performance difference for accuracy and false positive rate respectively. VPLA was the most enhanced in terms of accuracy and false positive rate using the smaller feature space acquired through the GainRatio attribute selection algorithm. However, the false positive rate for VPLA is still higher than the other four classifiers. Also, the standard deviation of its accuracy was high relative to the other classifiers.

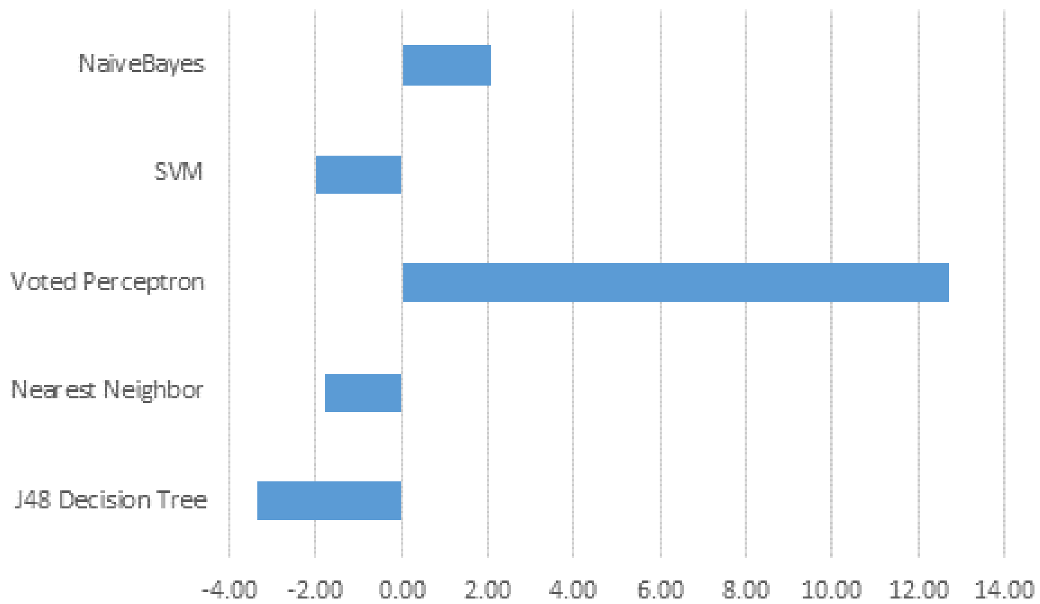

Figure 6.9

Classifier Accuracy Change using GR Evaluation - Cross-Validation Set

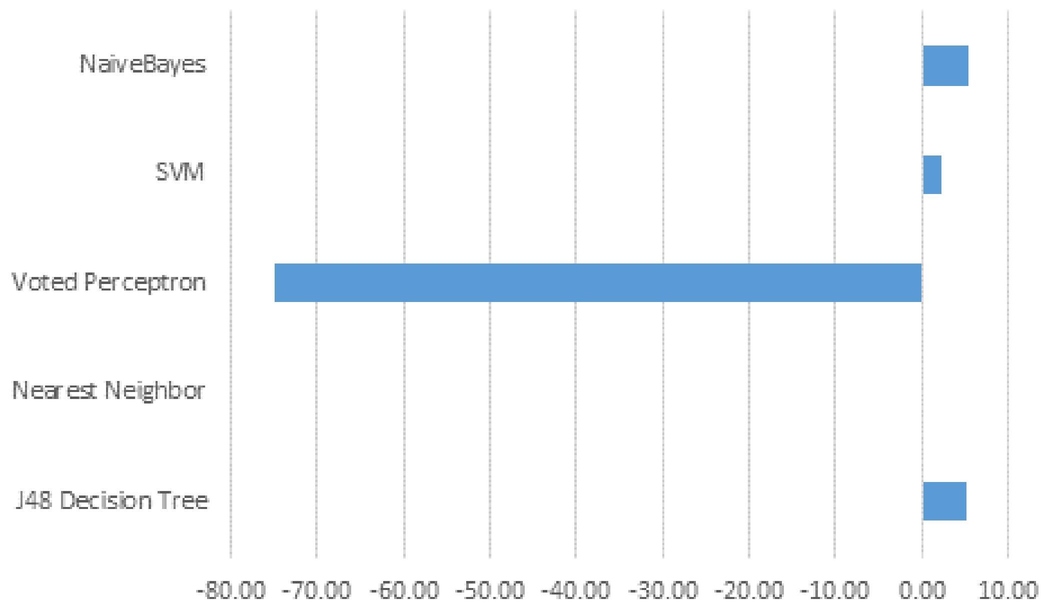

Figure 6.10

Classifier FP Rate Change using GR Evaluation - Cross-Validation Set

6.2.2.2 Classifier with GainRatio Selection - Test Set

Table 6.8 contains the results of each classifier using GainRatio on the test dataset. The 'Significant' column states if the change of the associated metric was statistically significant.

Figure 6.11 and Figure 6.12 represent the performance difference for accuracy and false positive rate respectively. VPLA was again the most affected by the use of the GainRatio algorithm. Unlike the results from the cross-validation set, it surpassed both Naive Bayes and support vector machine for accuracy and false positive rate.

6.3 Research Question 4

Can the ensemble learning technique Bagging improve performance over an individual classifier?

Table 6.8

Classifiers with Gain Ratio - Test Set

Classifier	Accuracy	Significant	FP Rate	Signficant
NaiveBayes	91.80	No	14.4	Yes
SVM	94.36	Yes	9.2	Yes
Voted Perceptron	94.43	Yes	8.7	Yes
Nearest Neighbor	95.10	Yes	6.8	No
J48 Decision Tree	95.03	Yes	4.5	Yes

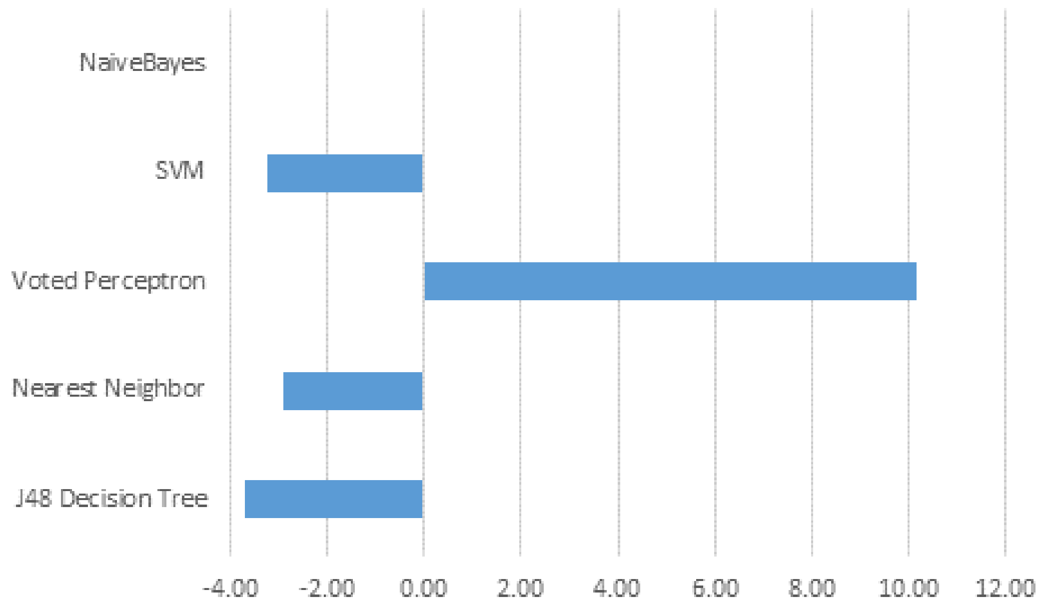

Figure 6.11

Classifier Accuracy Change using GR Evaluation - Test Set

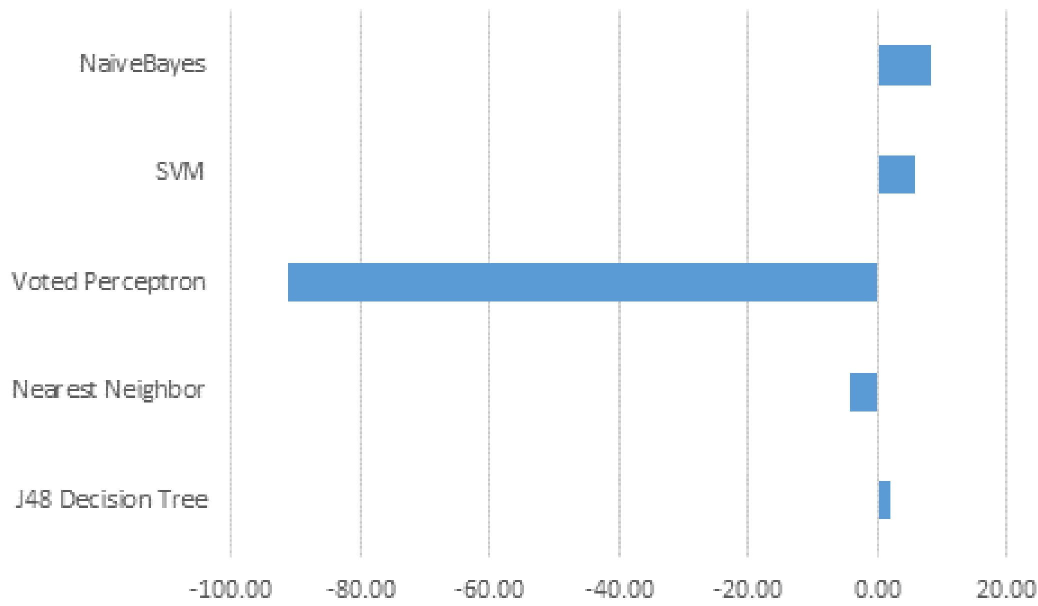

Figure 6.12

Classifier FP Rate Change using GR Evaluation - Test Set

This experiment compared the base classification methods to the bagging method using the same classifier. Ensemble learning in general has been shown to increase performance of the associated classifier, particularly when the classifier is unstable [9].

6.3.1 Classifiers with Bagging - Cross-Validation Set

Table 6.9 contains the results of each classifier using the bagging technique on the cross-validation dataset. The 'Significant' column states if the change of the associated metric using bagging was statistically significant versus that of the base classifier.

Figure 6.13 and Figure 6.14 represent the performance difference for accuracy and false positive rate respectively. The only significant change was to Naive Bayes which had a lower accuracy using bagging (90.18%) and a lower false positive rate (5.0%). The

Table 6.9

Classifiers with Bagging - Cross-Validation Set

Classifier	**Accuracy**	**Significant**	**Std Dev**	**FP Rate**	**Significant**	**Std Dev**
NaiveBayes	90.18	Yes	2.71	5	Yes	0.02
SVM	98.35	No	0.36	4.01	No	0.01
Voted Perceptron	80.23	No	0.04	100	No	0.00
Nearest Neighbor	98.82	No	0.30	3.46	No	0.01
J48 Decision Tree	99.30	No	0.25	2.18	No	0.01

bagging method did not have any effect on VPLA. Bagging had a negligible effect on support vector machine, nearest neighbor, and J48.

6.3.2 Classifiers with Bagging - Test Set

Table 6.10 contains the results of each classifier using the bagging technique on the test dataset. The 'Significant' column states if the change of the associated metric using bagging was statistically significant versus that of the base classifier.

Figure 6.15 and Figure 6.16 represent the performance difference for accuracy and false positive rate respectively. The changes in accuracy were similar to those of the cross-validation dataset. Naive bayes had the only statistically significant change in accuracy and false positive rate though the changes to false positive rate were more substantial than when evaluated using the cross-validation dataset.

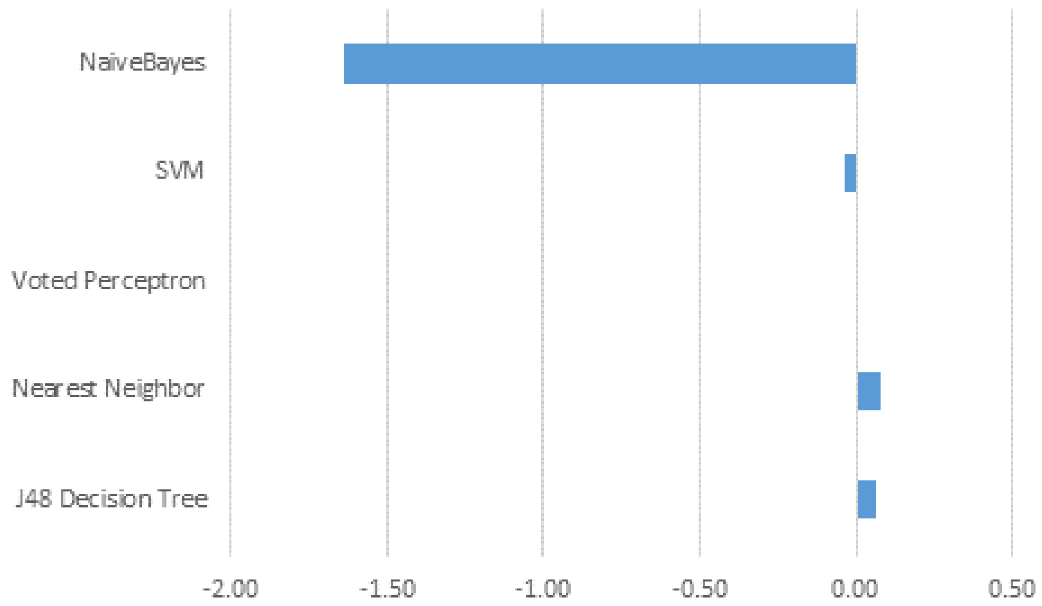

Figure 6.13

Classifier Accuracy Change using Bagging - Cross-Validation Set

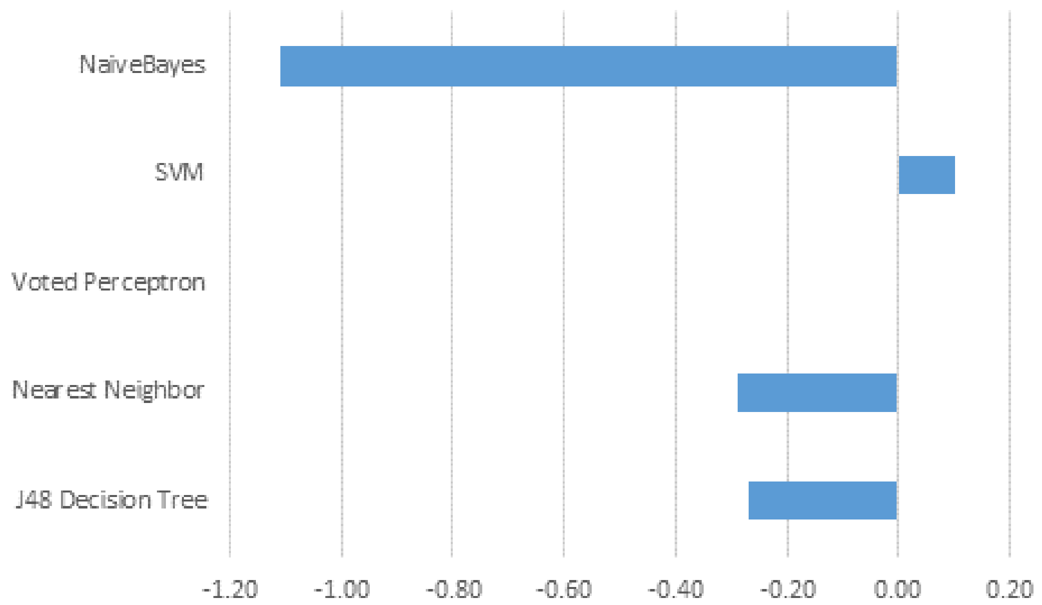

Figure 6.14

Classifier FP Rate Change using Bagging - Cross-Validation Set

Table 6.10

Classifiers with Bagging - Test Set

Classifier	Accuracy	Significant	FP Rate	Std Dev
NaiveBayes	87.6	Yes	5.4	Yes
SVM	97.55	No	5.7	No
Voted Perceptron	84.26	No	100	No
Nearest Neighbor	98.11	No	7.8	No
J48 Decision Tree	98.4	No	3.5	No

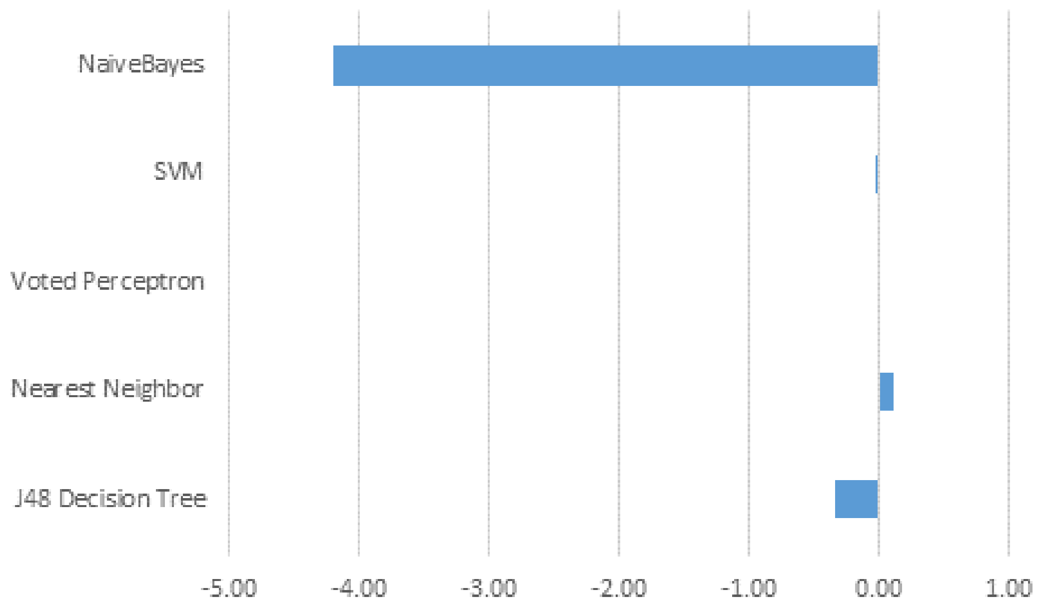

Figure 6.15

Classifier Accuracy Change using Bagging - Test Set

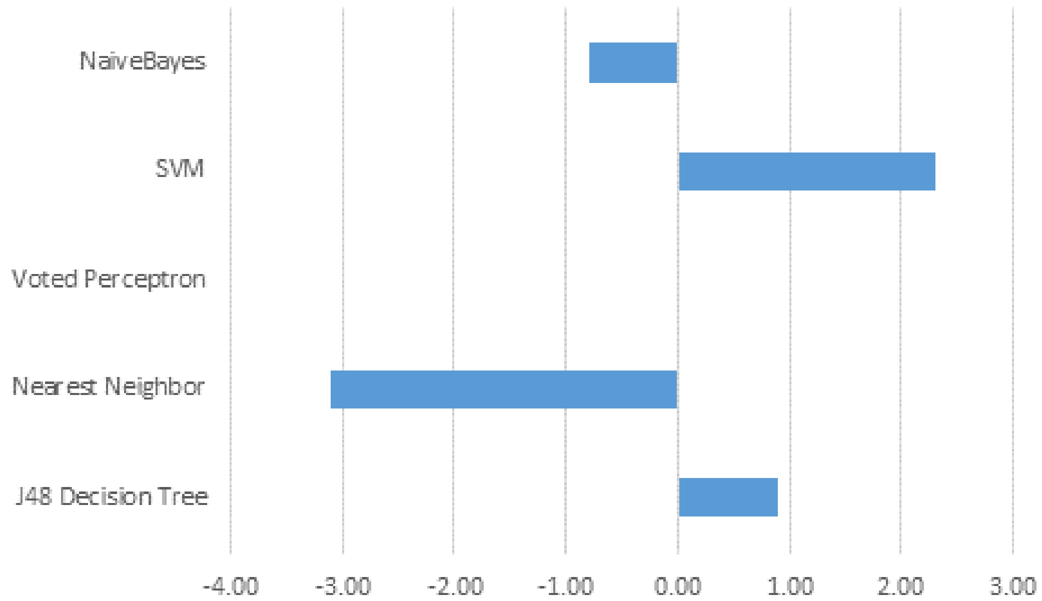

Figure 6.16

Classifier FP Rate Change using Bagging - Test Set

6.4 Memory Features versus PE Header Features

PE header features have been used in several previous works (see chapter 2) as a method for classifying malware. Though the PE header features defined in this work are obtained from memory, they are the same as if extracted from the file on disk. For this experiment, the datasets were split based on the types of features into four separate datasets:

- Cross-Validation_Memory
- Cross-Validation_PE
- Test_Memory
- Test_PE

Any feature related to the file itself was considered a PE feature for this experiment, even if it is not extracted from the PE header. The features used to create the PE datasets, and consequently removed from the Memory datasets, are:

- dll_flag_set
- entrypoint_section_name
- file_extension_is_dll
- has_reloc_section
- is_com_client
- is_com_server
- num_exports
- num_imports
- PE_image_base
- raw_virt_size_diff
- size_of_image

These were used to compare the performance of several classifiers using only memory or PE features. As with the previous experiments, the classifiers were run against the cross-validation datasets using ten iterations of stratified 10-fold cross-validation. For the test sets, the classifiers used a model learned from training on the appropriate cross-validation set.

6.4.1 Feature Comparison - Cross-Validation Set

Table 6.11 lists the accuracy and false positive rate of each classifier using the cross-validation datasets. The 'Significant' column refers to the statistical significance between

Table 6.11

Memory and PE Feature Metrics - Cross-Validation Set

Classifier	Mem Acc	PE Acc	Significant	Mem FP	PE FP	Significant
Naive Bayes	92.40	92.29	No	18.80	8.67	Yes
SVM	94.54	95.89	Yes	22.65	6.55	Yes
Voted Perceptron	84.32	80.97	Yes	42.16	100.00	Yes
IBK	98.01	94.07	Yes	4.91	24.65	Yes
J48	99.00	97.90	Yes	3.41	5.56	Yes

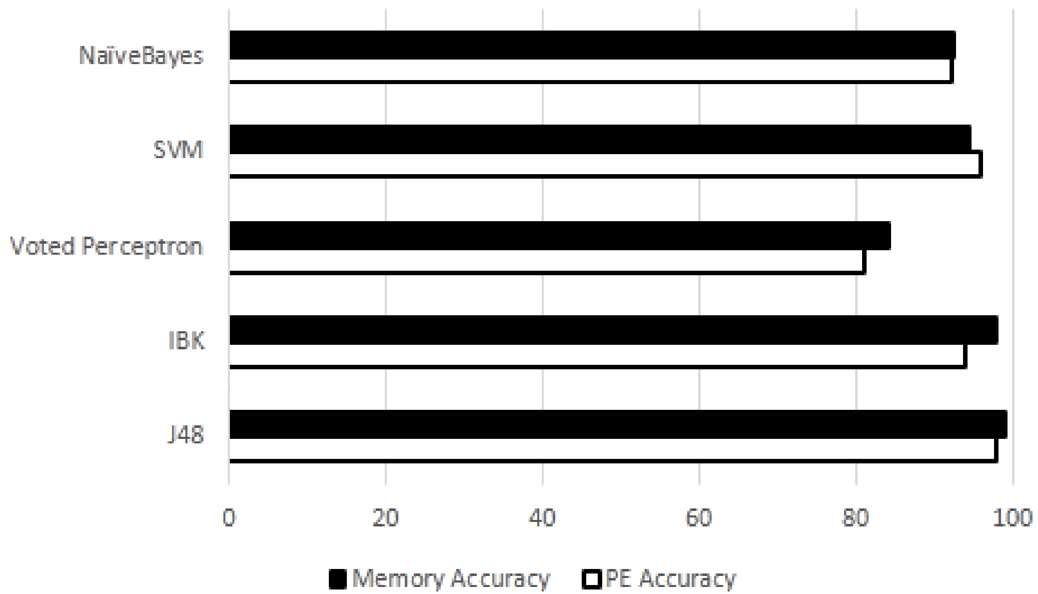

Figure 6.17

Memory and PE Features Accuracy - Cross-Validation Set

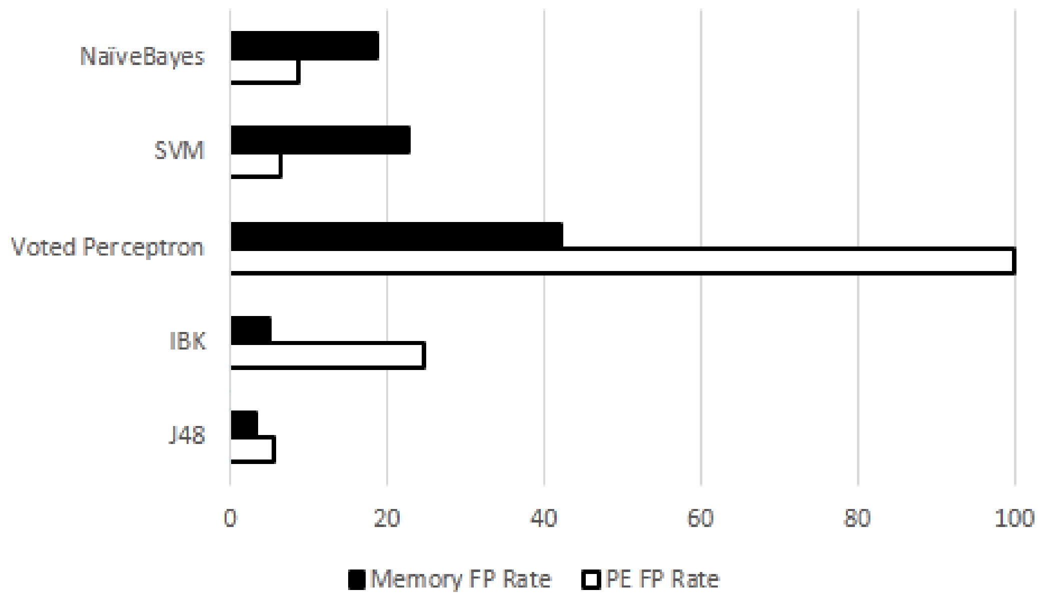

Figure 6.18

Memory and PE Features FP Rate - Cross-Validation Set

a given metric for the memory features and PE features. Figure 6.17 and Figure 6.18 show the accuracy and false positive rate respectively.

Regarding accuracy, the only statistically significant result that favored PE features was from the SVM algorithm that had a 95.89%. The IBK and J48 algorithms using memory features performed the best in terms of false positive rate.

6.4.2 Feature Comparison - Test Set

Table 6.12 lists the accuracy and false positive rate of each classifier using the test datasets. The 'Significant' column refers to the statistical significance between a given metric for the memory features and PE features. Figures Figure 6.19 and Figure 6.20 show the accuracy and false positive rate respectively.

Table 6.12

Memory and PE Feature Metrics - Test Set

Classifier	Mem Acc	PE Acc	Significant	Mem FP	PE FP	Significant
Naive Bayes	92.76	91.72	No	18.40	10.10	Yes
SVM	96.18	94.58	Yes	19.10	9.20	Yes
Voted Perceptron	89.12	84.26	Yes	46.50	100.00	Yes
IBK	97.55	95.73	Yes	3.10	7.30	Yes
J48	98.40	96.44	Yes	2.80	7.50	Yes

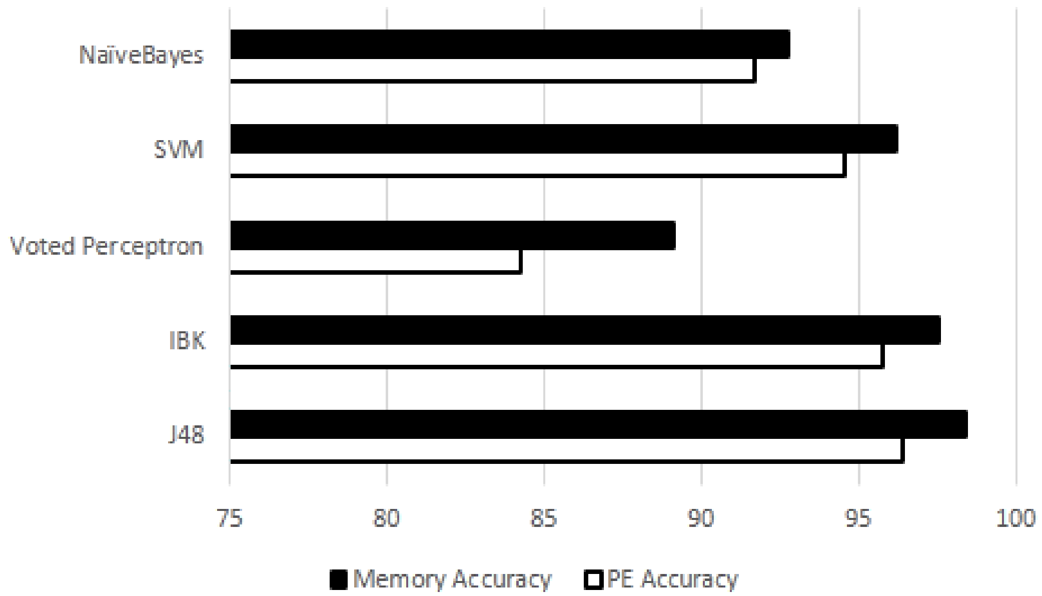

Figure 6.19

Memory and PE Features Accuracy - Test Set

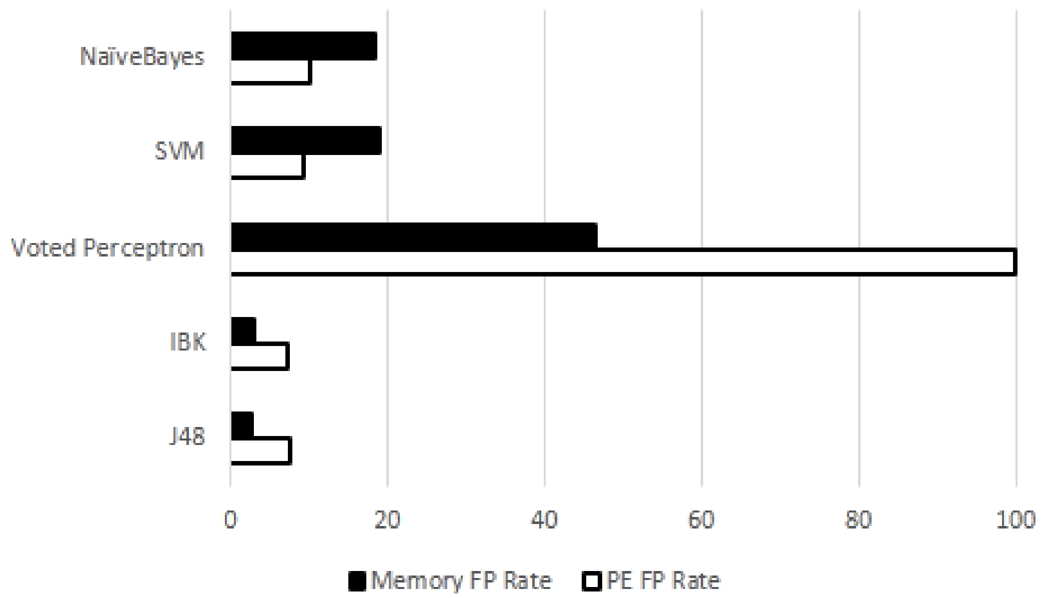

Figure 6.20

Memory and PE Features FP Rate - Test Set

Using just the memory features achieved higher accuracy than any of the classifiers that used the PE features. The results were more varied for false positive rate, but again the IBK and J48 algorithms applied to the memory feature datasets showed the best performance.

CHAPTER 7

ANALYSIS

This chapter presents an analysis of the experimental results to answer the research questions.

7.1 Do malicious DLLs have distinct patterns of behavior in memory?

The results of the study presented in chapter 4 indicate that malicious DLLs do have distinct patterns of behavior in memory. These patterns, represented as features in the dataset, serve to distinguish between legitimate and malicious DLLs. Some of the features identified are trivial for a malware author to adjust requiring little more than setting a flag in the compiler. Examples of trivial features include the base address where a DLL is loaded or the file extension used for the DLL. Other features capture more fundamental aspects of the malware such as the names and numbers of the processes it targets and the timing of injections.

7.2 How do different classifiers perform against the dataset?

Six classifiers were evaluated against the two datasets described in chapter 3. The ZeroR classifier was used as a baseline to evaluate the performance of the other five classifiers using the feature set defined in chapter 5.

On both datasets, cross-validation and the test set, four of the five classifiers (Naive Bayes, Support Vector machine, Nearest Neighbor, and J48) outperformed ZeroR in terms of both accuracy and false positive rate. The Voted Perceptron Learning Algorithm (VPLA) produced the same results as the ZeroR method for both accuracy and false positive rate. The J48 Decision Tree had the highest accuracy and lowest false positive rate of the tested classifiers.

With four of the five classifiers achieving high accuracy and low false positive rates as well as surpassing the baseline ZeroR classifier, it appears the feature set captures distinguishing behavior between malicious and leigitmate DLLs extracted from Windows 7 x86 memory images. If the feature set only represented noise, the results were likely to have been more erratic.

7.3 Do correlation-based feature selection and gain ratio evaluation improve classifier performance for this dataset?

Two different feature selection algorithms were applied to each of the five classifiers: Correlation-based Feature Selection and Gain Ratio Attribute Evaluation. The ZeroR classifier was not evaluated due to the fact that the hypothesis space it searches is not affected by a reduction in the dimensionality of the attribute space.

7.3.1 Correlation-Based Feature Selection using Best-First Search

CFS using best-first search was applied to the cross-validation dataset. It produced a subset of seven features it considered highly correlated with the class but not highly correlated with each other.

Using this subset of features, the Naive Bayes classifier achieved better accuracy on both datasets, although it still performed worse than the SVM, Nearest Neighbor, and J48 classifiers. The Naive Bayes classifier in particular suffers from redundant features in the dataset which the CFS algorithm helps alleviate.

The VPLA had lower accuracy using CFS on both datasets, but was able to classify instances of both classes correctly as opposed to using the entire feature set when it had a 100% false positive rate.

7.3.2 Gain Ratio Attribute Evaluation using Ranker

The Gain Ratio algorithm was used to rank the attributes in order of the gain ratio they provided relative to the class. The top seven were chosen and the five classifiers were evaluated using these features. Four of the seven features matched those produced by CFS.

The most significant improvement to accuracy using Gain Ratio was for the VPLA. It achieved a 92.96% and 94.43% accuracy on the cross-validation and test sets respectively. On the test set it also had a lower false positive rate than Naive Bayes and the SVM classifiers.

7.3.3 Evaluation

For some classifiers, attribute selection offered direct improvements in performance. With the high accuracy and low false positive rates obtained by some of the base classifiers (SVM, Nearest Neighbor, and J48), improvements were unlikely. However, neither did these classifiers suffer any significant losses in performance. This indicates that the feature

space contains at least some noise and a more reliable model could be produced through the application of proper feature selection algorithms.

The model produced by the base J48 decision tree that was evaluated on the test had a size of 93 with 49 leaves. The model for J48 with CFS had a size of 93 with 47 leaves. The model for J48 with the Gain Ratio feature selection method had a size 25 with 13 leaves. While it did lose nearly 4% in accuracy, its false positive rate only increased by 1%. Additionally, given the reduced complexity of a decision tree of size 25, as opposed to 93, the model should generalize better.

7.4 Can the ensemble learning technique Bagging improve performance over an individual classifier?

For the cross-validation dataset the Bagging technique had a small impact on both accuracy and false positive rate. The only statistically significant change was on Naive Bayes.

The results on the test dataset were similar for accuracy (compared to the cross-validation dataset) but more varied in regards to false positive rate. However, Naive Bayes was again the only statistically significant change for either metric.

Bagging did not offer any significant improvement beyond the false positive rate of Naive Bayes. However, aside from Naive Bayes, it did not significantly reduce the performance of any of the classifiers.

7.5 Memory and PE Feature Comparison

The datasets were divided by feature type, memory and PE, and the classifiers compared against each other using the same base datasets. For the cross-validation dataset, the SVM and Voted Perceptron classifiers performed better using PE features than memory features while IBK and J48 performed better using memory features. The accuracy of Naive Bayes was similar but the false positive rate was significantly worse using memory features (18.8% vs 8.67%).

For the test set, all the classifiers trained using memory features performed better than those using PE features. The results for false positive rates were mixed with Naive Bayes and SVM performing worse using memory features and Voted Perceptron, IBK, and J48 performing better using only memory features.

Though the ZeroR classifier was not used in these tests, its accuracy and false positive rate would be the same as those from the base classifier accuracy as the ratio of class values (malicious vs legitimate) did not change. None of the classifiers in this experiment performed worse than the ZeroR classifier.

These results indicate that memory features on their own are viable for distinguishing malicious DLLs from legitimate. However, appropriately chosen PE features can enhance the reliability of the learned model. Contrasting the results from these experiments against the base classifiers using the full feature set shows a marked improvement for several of the classifiers, particularly in regards to false positive rate.

7.6 J48 Decision Tree

In several of the experiments the J48 decision tree performed the best for both accuracy and misclassification rate of malicious DLLs. Using the base algorithm without any modifications, J48 correctly classified 2,660 of the 2,694 instances in the test dataset. Part of this is likely due to the variety of features in the dataset. It consists of binary and multivalued nominal attributes as well as both discrete and continuous numeric data. There are also, as with most real datasets, missing values. J48 is capable of handling all of these types of features, though it may benefit from discretizing the continuous attributes (load_position and init_position) [27]

The J48 classifier had 15 false positives (malicious DLLs classified as legitimate) and 19 false negatives (legitimate DLLs classified as malicious) when evaluated against the test set.

7.6.1 J48 False Positives

Table 7.1 lists the instance number, malicious DLL name, and the reasons it was misclassified using the model generated by the base J48 algorithm trained on the cross-validation set and evaluated against the test set. Altering any one of the values listed in the 'Reason for Misclassification' column would result in that instance being correctly classified. Many of these instances evaded correct classification because the malware author followed common practices for creating Windows DLLs. Most of the instances had the standard section name that contained the entrypoint (.text), used .dll as a file extension, and contained a .reloc section.

Table 7.1

False Positives from Base J48 Decision Tree

Instance	Malicious DLL	Reason for Misclassification
147	utilman.dll	entry = .text; extension is .dll
168	bvfdgqpksdmtpt.dll	entry = .text; has .reloc section; extension is .dll
169	bvfdgqpksdmtpt.dll	entry = .text; has .reloc section; extension is .dll
171	00000b30.tmp	entry = .text; has .reloc section
207	newdotnet6_38.dll	entry = .text; has .reloc section; extension is .dll
208	newdotnet6_38.dll	entry = .text; has .reloc section; extension is .dll
209	newdotnet6_38.dll	entry = .text; has .reloc section; extension is .dll
210	newdotnet6_38.dll	entry = .text; has .reloc section; extension is .dll
211	newdotnet6_38.dll	entry = .text; has .reloc section; extension is .dll
212	newdotnet6_38.dll	entry = .text; has .reloc section; extension is .dll
251	IEShims.dll	entry = .text; has .reloc section; extension is .dll
267	svcdlzi.dat	entry = .text; has .reloc section; high CC; high AP
278	sfc_my.dll	entry = .text; has .reloc section; extension is .dll
311	318C.tmp	$mapped_views \leq 1$
381	bootext.dll	entry = .text

7.6.2 J48 False Negatives

Table 7.2 lists the instance number, legitimate DLL name, and the associated program of the DLL. As with many systems that seek to identify malware, anti-virus software tends to be misclassified. 15 of the 19 (79%) false negatives were associated with an anti-virus application. The reason these were misclassified was duedue to a variety of features so these were not listed in the table.

Table 7.2

False Negatives from Base J48 Decision Tree

Instance	Legitimate DLL	Program
652	Tools.dll	Spybot
674,675	snlFileFormats150.bpl	Spybot
721,722	snlThirdParty150.bpl	Spybot
747	7z.dll	Malwarebytes
831	SDLicense.dll	Spybot
851,852	SDLists.dll	Spybot
1061	snlBase150.bpl	Spybot
1094,1095	SDAdvancedCheckLibrary.dll	Spybot
1126	DEC150.bpl	Spybot
1251	SDfileScanLibrary.dll	Spybot
2473	focuskiller.dll	PyCharm
2484,2485	NppExport.dll	Notepad++
2588	sacore.dll	McAfee AV
2637	bass.dll	MyRealGames.com\Space Bubbles

CHAPTER 8

CONCLUSIONS

The goal of this dissertation was to design and evaluate a feature set created from DLLs in Windows 7 x86 memory images. To accomplish this, malicious DLLs in memory images were studied to identify behavioral patterns. These patterns, or characteristics, were used to develop a feature set that could be extracted from the DLLs in a memory image. Features were extracted directly from several data structures in memory or created by applying different transformations to features.

Cuckoo Sandbox was used to automatically create memory images from malware samples obtained from VirusShare. Volatility was used to build a feature extractor which was then applied to the created memory images. Ground truth for each set of features associated with a given DLL was determined using a whitelist of legitimate files built from a clean image of the analysis virtual machine.

Two datasets were created using this technique. The legitimate data was generated manually from different systems run Windows 7 x86. Several third-party applications existed in memory at the time of capture. The first dataset was the cross-validation dataset. Each classifier evaluated using 10 iterations of stratified 10-fold cross-validation on this dataset. The cross-validation dataset consisted of 12,066 data points with 2,385 (19.77%)

being malicious and the remaining 9,681 (80.23%) legitimate. The second dataset was an independent test set. The model was learned from the cross-validation dataset and then evaluated against the test set. The test set consisted of 2,694 data points with 424 (15.74%) malicious and 2,270 (84.26%) being legitimate.

The feature set was empirically evaluated using six machine learning classification algorithms. Each classifier was applied to both datasets in several experiments to determine if the features captured meaningful behaviors of the DLLs and could be used to learn a model capable of acceptable performance on new data. The ZeroR method was used as the performance baseline for comparison against five classifiers with regards to accuracy and false positive rate. Performance on each dataset was similar in each experiment.

The hypothesis for this dissertation is

> **A machine learning model can be learned from features extracted from Windows 7 memory images and applied to successfully classify malicious injected DLLs in Windows 7 memory images.**

The empirical evidence obtained from the experiments in this dissertation support the hypothesis. We have shown that the base classification algorithms, without applying any algorithm optimizations or feature engineering, significantly outperform the ZeroR method.

The most useful features were identified using feature selection. These included the base address of the DLL in memory, the name of its entrypoint section, the file extension used, the path where it was loaded, and the presence of a .reloc section in the PE header.

Across several of the experiments, the J48 classifier consistenly had the best performance in terms of accuracy and false positive rate. We believe this is due to its ability to handle various types of data as well as its built in feature selection, Gain Ratio. The

base J48 classifier in Weka, without the addition of parameter tuning, feature selection, or ensemble learning achieved a 99.25% accuracy (2.45% FP rate) on the cross-validation set and 98.74% accuracy (3.5% FP rate) on the test set.

A machine learning classifier can be trained using features extracted from a Windows 7 x86 memory image to classify DLLs as legitimate or malicious.

8.1 Publication Plan

The analysis in chapter 4, preliminary results, is published in

- D. Glendowne, C. Miller, W. McGrew, and D. Dampier, Characteristics of Malicious DLLs in Windows Memory, *Advances in Digital Forensics XI*. Springer Berlin Heidelberg, 2015.

The experimental results and analysis of the machine learning algorithms on the feature set will be submitted to the *Journal of Digital Forensics, Security, and Law (JDFSL)*.

8.2 Contributions

This dissertation makes the following contributions.

- Infected memory images
- Datasets
- DLL characteristics
- Experimental results
- The machine learning model

8.2.1 New Infected Memory Images

The number of available Windows memory images publicly available is limited. The largest collection is hosted or linked to by the Volatility Foundation [53], and it contains

less than one hundred images. Many of the images do not contain malware and the majority of the samples are created from Windows XP. With the end of Windows XP support from Microsoft, it is expected that the majority of the remaining Windows XP machines will steadily phase out. There is a strong need for memory images to enable tool development, testing, education, and research in memory forensics.

In 2011, Vidas [60] created a corpus of Windows memory images for these same reasons. Vidas originally hosted the corpus with NIST agreeing to take over the hosting in the future. Neither the website supplied by the author nor the NIST Computer Forensics Reference Data Sets (CFReDS) [42] contain any mention of the corpus.

So far, as a result of this research, over 40,000 Windows 7 memory images have been generated. Additionally, there is some limited analysis of the memory images indicating the type or form of the malware within the image.

8.2.2 Datasets

As with memory images, there is a need for datasets created from malware for researchers to experiment on. There are no datasets currently available that were constructed from memory images. The two datasets created during the course of this research will be made available for evaluation by the research community.

8.2.3 DLL Characteristics

The preliminary results of this research have identified several common characteristics of malicious DLLs. These characteristics establish that a pattern exist among malicious DLLs and form the basis of the feature set being constructed in this research. They may

also be used as heuristics for forensics examiners to leverage during an investigation or incorporated into a behavioral detection system.

8.2.4 Experimental Results

This research evaluated several classification algorithms against two datasets using features extracted from Windows memory images. The results of these experiments provide a baseline for comparison against in future works.

8.3 Volatility Plugins

The software used for extracting features from memory images will be incorporated into a Volatility plugin. This will enable easy use for practitioners through a widely available and well supported memory forensics platform. A separate Volatility plugin will implement the classification algorithm and a model derived from a large set of training data. We foresee two uses of these plugins:

- Aiding forensic examiners during an investigation
- Automating detection of malware in memory on live systems

8.3.1 Aiding Forensic Examiners

Indicators of compromise (IOCs) are artifacts used by forensic examiners and incident responders to identify systems that have been infected or compromised. These may originate from an intrusion detection system (IDS), an antivirus alert, or from forensic examination of a separate system. IOCs may be unique strings, IP addresses, domain names, or any other system artifact uncovered during a forensic examination. These provide the

examiner leads when a system is being investigated allowing them to reduce noise and focus their search.

When an examiner lacks IOCs, the analysis is less focused and transitions to a search for anomalies within the system. The classification model produced by this research offers the potential to identify leads for the examiner automatically. The necessary features can be extracted from a memory image and processed by the classifier prior to the examiner beginning the analysis. This is the reason that minimizing false negatives is a priority for this model. If the model determines DLLs within the memory image are malicious, it is preferable for the analyst to further investigate these DLLs and prove the model wrong than to miss them altogether because the model classified them as legitimate.

8.3.2 Automated Detection

The classification model could be applied as an additional line of defense for a network as a host based monitoring system. Applying the model to a given system simply requires extracting the features from memory. The features could also be extracted from live memory in real time. Gionta et. al [19] designed an architecture that enables the efficient acquisition and scanning of memory in massive virtual environments. During the course of this scanning, the appropriate features could be extracted and supplied to the classifier to determine the presence of malicious DLLs.

8.4 Future Work

These are areas we see as logical next steps in this research.

8.4.1 Feature Set

A direct step from this work is improving the feature set described in this work. We believe there are additional features in memory that can be extracted that help distinguish between malicious and legitimate DLLs. We did not use any of the libraries imported by a DLL or specific API calls. Additionally, there are ways of determining, for some DLLs, if the DLL belong in that process. These include examining the import address table of the all the loaded modules and scanning for *Call/JMP* instructions that point to the virtual address range occupied by a given DLL. This will not account for all the DLLs loaded in a given process, but will affect some of them.

8.4.2 Other Types of Malware in Memory

In addition to DLLs, malware may be implemented as shellcode, a process, or a kernel driver in Windows. Each of these areas presents different structures, and subsequently different features, that can be used for classifying malware in Windows memory. For instance, there any many objects that map directly back to a process (i.e. they are associated with a PID) that may be suitable as features in classification.

REFERENCES

[1] "Microsoft Developers Network, Windows API Reference,", http://msdn.microsoft .com, 2014.

[2] "Report: Average of 82,000 new malware threats per day,", http://www.pcworld .com/article/2109210/report-average-of-82-000-new-malware-threats-per-day-in-2013 .html, 2014.

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

[4] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, "Using spatio-temporal information in API calls with machine learning algorithms for malware detection," *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*. ACM, 2009, pp. 55–62.

[5] M. Alazab, S. Venkataraman, and P. Watters, "Towards understanding malware behaviour by the extraction of API calls," *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*. IEEE, 2010, pp. 52–59.

[6] "AV Is Not Enough for the Enterprise,", http://www.symantec.com/connect/blogs/av-not-enough-enterprise, 2014.

[7] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Blacksheep: Detecting compromised hosts in homogeneous crowds," *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 341–352.

[8] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, 2007, pp. 156–168.

[9] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, 1996, pp. 123–140.

[10] A. Caglayan, M. Toothaker, D. Drapeau, D. Burke, and G. Eaton, "Real-time detection of fast flux service networks," *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*. IEEE, 2009, pp. 285–292.

[11] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, 1995, pp. 273–297.

[12] "Cuckoo Sandbox: Automated Malware Analysis,", http://www.cuckoosandbox.org/, 2014.

[13] B. Dolan-Gavitt, "The VAD tree: A process-eye view of physical memory," *digital investigation*, vol. 4, 2007, pp. 62–64.

[14] S. Dolev and N. Tzachar, "Malware signature builder and detection for executable code,", May 26 2010, EP Patent App. EP20,090,014,289.

[15] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer, "Applying machine learning techniques for detection of malicious code in network traffic," *KI 2007: Advances in Artificial Intelligence*, Springer, 2007, pp. 44–50.

[16] "Finding Hidden Processes with Volatility and Scanning with Sysinternals Sigcheck,", https://www.myotherpcisacloud.com/post/2013/12/19/Finding-Hidden-Processes-with-Volatility-and-Scanning-with-Sysinternals-Sigcheck1.aspx, 2013.

[17] I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*. IEEE, 2010, pp. 201–203.

[18] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Machine learning*, vol. 37, no. 3, 1999, pp. 277–296.

[19] J. Gionta, A. Azab, W. Enck, P. Ning, and X. Zhang, "SEER: Practical Memory Virus Scanning as a Service," *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2014.

[20] D. Glendowne, C. Miller, W. McGrew, and D. Dampier, "Characteristics of Malicious DLLs in Windows Memory," *Advances in Digital Forensics XI*, Springer, 2015.

[21] M. A. Hall, *Correlation-based feature selection for machine learning*, doctoral dissertation, The University of Waikato, 1999.

[22] A. Honig, *Practical Malware Analysis*, No Starch Press, 2012.

[23] "Hunting Malware with Memory Analysis,", http://www.solutionary.com/resource-center/blog/2012/12/hunting-malware-with-memory-analysis/, 2012.

[24] "INetSim,", http://www.inetsim.org/, 2008.

[25] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, 2005, pp. 13–23.

[26] A. R. Katz, "Image Analysis and Supervised learning in the automated Differentiation of White Blood cells from Microscopic Images," 2000.

[27] F. Kaya, "Discretizing Continuous Features for Naive Bayes and C4. 5 Classifiers," *University of Maryland publications*, 2008.

[28] J. D. Kornblum and C. ManTech, "Exploiting the rootkit paradox with windows memory analysis," *International Journal of Digital Evidence*, vol. 5, no. 1, 2006, pp. 1–5.

[29] M. Ligh, S. Adair, B. Hartstein, and M. Richard, *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*, Wiley Publishing, 2010.

[30] 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, 2014.

[31] "M-Trends 2010 The Advanced Persistent Threat,", https://www.mandiant.com/blog/m-trends-the-advance-of-the-persistent-threat/, 2010.

[32] "M-Trends 2011 When Prevention Fails,", https://www.mandiant.com/media-room/publications/, 2011.

[33] "M-Trends 2012 An Evolving Threat,", https://www.mandiant.com/media-room/publications/, 2012.

[34] "M-Trends 2013: Attack the Security Gap,", http://connect.mandiant.com/mtrends 2013-eml, 2013.

[35] "M-Trends 2014 Beyond the Breach,", https://dl.mandiant.com/EE/library/WP_M-Trends2014_140409.pdf, 2014.

[36] F. Maggi, S. Zanero, and V. Iozzo, "Seeing the invisible: forensic uses of anomaly detection and machine learning," *ACM SIGOPS Operating systems review*, vol. 42, no. 3, 2008, pp. 51–58.

[37] "Memoryze: Find Evil in Live Memory,", https://www.mandiant.com/resources/download/memoryze/, 2014.

[38] E. Menahem, L. Rokach, and Y. Elovici, "Troika–An improved stacking schema for classification tasks," *Information Sciences*, vol. 179, no. 24, 2009, pp. 4097–4122.

[39] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving malware detection by applying multi-inducer ensemble," *Computational Statistics & Data Analysis*, vol. 53, no. 4, 2009, pp. 1483–1494.

[40] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, "Unknown malcode detection using OPCODE representation," *Intelligence and Security Informatics*, Springer, 2008, pp. 204–215.

[41] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, and Y. Elovici, "Unknown malcode detection via text categorization and the imbalance problem," *Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on*. IEEE, 2008, pp. 156–161.

[42] "The CFReDS Project,", http://www.cfreds.nist.gov/, 2014.

[43] "NIST Computer Security Incident Handling Guide,", http://csrc.nist.gov/publications/nistpubs/800-61rev2/SP800-61rev2.pdf, 2014.

[44] N. O'Rourke, L. Hatcher, and E. Stepanski, *A Step-by-step Approach to Using SAS for Univariate & Multivariate Statistics*, 2nd edition, SAS Publishing, 2005.

[45] N. L. Petroni Jr, 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, 2006, pp. 197–210.

[46] "Polymorphic and Metamorphic Malware,", https://www.blackhat.com/presentations-/bh-usa-08/Hosmer/BH_US_08_Hosmer_Polymorphic_Malware.pdf, 2008.

[47] J. R. Quinlan, *C4. 5: programs for machine learning*, Elsevier, 2014.

[48] "Reconstructing a Binary,", http://computer.forensikblog.de/en/2006/06/reconstructing-a-binary-3.html, 2006.

[49] "Rekall,", http://www.rekall-forensic.com/, 2014.

[50] "Responder Pro,", http://mcsi.mantech.com/products/responder

[51] M. E. Russinovich and D. A. Solomon, *Microsoft windows internals*, Microsoft Press, 2005.

[52] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining API calls," *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 1020–1025.

[53] "Sample Memory Images,", https://code.google.com/p/volatility/wiki/SampleMemoryImages, 2014.

[54] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, "N-grams-based File Signatures for Malware Detection.," *ICEIS (2)*, 2009, pp. 317–320.

[55] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 38–49.

[56] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *Information Security Technical Report*, vol. 14, no. 1, 2009, pp. 16–29.

[57] M. Siddiqui, M. C. Wang, and J. Lee, "Data mining methods for malware detection using instruction sequences," *Proceedings of Artificial Intelligence and Applications, AIA 2008*, 2008.

[58] E. Stalmans and B. Irwin, "A framework for DNS based detection and mitigation of malware infections on a network," *Information Security South Africa (ISSA), 2011*. IEEE, 2011, pp. 1–8.

[59] A. R. Tindale, A. Kapur, W. A. Schloss, and G. Tzanetakis, "Indirect acquisition of percussion gestures using timbre recognition," *Proc. Conf. on Interdisciplinary Musicology (CIM)*, 2005.

[60] T. Vidas, "MemCorp: An open data corpus for memory analysis," *System Sciences (HICSS), 2011 44th Hawaii International Conference on*. IEEE, 2011, pp. 1–6.

[61] "VirusShare - Because Sharing is Caring,", http://virusshare.com/, 2014.

[62] "VirusTotal - Free Online Virus, Malware and URL Scanner,", https://www.virus total.com/, 2014.

[63] "Volatility Foundation,", https://github.com/volatilityfoundation, 2014.

[64] "Volatility 2.0 Plugin Vscan,", http://malwarereversing.wordpress.com/2011/09/17/-volatility-2-0-plugin-vscan/, 2011.

[65] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," *Recent Advances in Intrusion Detection*. Springer, 2007, pp. 219–235.

[66] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann, 2005.

[67] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An intelligent PE-malware detection system based on association mining," *Journal in computer virology*, vol. 4, no. 4, 2008, pp. 323–334.

[68] D. Yuxin, Y. Xuebing, Z. Di, D. Li, and A. Zhanchao, "Feature representation and selection in malicious code detection methods based on static system calls," *Computers & Security*, vol. 30, no. 6, 2011, pp. 514–524.

APPENDIX A

SOFTWARE VERSIONS FOR LEGITIMATE DATA

Table A.1

Legitimate Third-Party Software

Software	Version
7-Zip	9.2
Adobe PDF Reader XI	11.0.10
Audacity	2.0.6
Dropbox	3.2.6
GIMP	2.8.14
Google Chrome	40.0.2214.115
Google Drive	1.19.8406.6504
HandBrake	0.10.0
Itunes	12.1.1.4
Malware Bytes	2.0.4.1028
Microsoft Access	2003
Microsoft Excel	2003
Microsoft Powerpoint	2003
Microsoft Word	2003
Mozilla Firefox	35.0.1
Notepad++	6.7.4
PhotoScape	N/A
PyCharm	4
Skype	7.1.105
Spybot	2.4.40
Tortoise SVN (32-bit)	1.7.24257
Virtual Clone Drive	5.4.7.0
VLC Media Player	2.1.5

APPENDIX B

FEATURE EXTRACTION SOURCE CODE

B.1 Classes for Extracting DLL Features from Windows 7 x86 Memory Images

```
#Written and tested using Python 2.7 and Volatility 2.4
import volatility.registry as registry
import volatility.conf as conf
import volatility.commands as commands
import volatility.addrspace as addrspace
import volatility.plugins.kdbgscan as kdbgscan
import volatility.plugins.taskmods as taskmods
import volatility.utils as utils
from volatility.plugins.malware.malfind import LdrModules
  as LdrModules
import volatility.plugins.vadinfo as vadinfo
import volatility.obj as obj
import csv


class DLL:
        """Object representing DLL features extracted from
           a memory image"""
        def __init__(self):#, process, base_address,
           modules_tup, positions_tup, vad_node):
             self.features = {}
```

```
    def columns(self):
        return sorted(self.features.keys(), key=lambda
          k: k.lower())


class DLLList:
    """Class for storing list of DLL objects.
       Can write list out to file as csv.
       Ensures all stored DLLs have same features"""
    def __init__(self):
        self.dlist = []
        self.column_names = None

    #Append to internal list
    #Assure added DLL has the same column names as the
       other dlls
    def append(self, dll):
        if not self.column_names:
            self.column_names = dll.columns()

        for col in self.column_names:
```

```
            if col not in dll.columns():
                print "Could not add DLL to list. Missing
                   column name: {0}".format(col)
                return

        self.dlist.append(dll)

    def write_to_csv(self, filename):
        with open(filename, 'wb') as csvfile:
            csvwriter = csv.DictWriter(csvfile, dialect='
               excel', quoting=csv.QUOTE_NONNUMERIC,
               fieldnames=self.column_names)
            csvwriter.writeheader()
            for dll in self.dlist:
                csvwriter.writerow(dll.features)



class vae:
    """Extracts specified features"""
```

```
def __init__(self, imagepath, profile, kdbg=None):
    self.imagename = imagepath.split('\\')[-1]
    registry.PluginImporter()
    self.config = conf.ConfObject()
    self.config.PROFILE = profile
    self.config.LOCATION = "file://" + imagepath
    self.cmds = registry.get_plugin_classes(commands.
        Command, lower=True)
    registry.register_global_options(self.config,
        commands.Command)
    registry.register_global_options(self.config,
        addrspace.BaseAddressSpace)


def build_dll_dicts(self, process):
    """Construct dictionaries of _LDR_DATA_TABLE_ENTRYs
        from the three doubly linked lists.
     Key: DLL base address
     Value: (module, module's position in list)"""
```

```
loadpos = [mod for mod in process.get_load_modules
   ()]
initpos = [mod for mod in process.get_init_modules
   ()]
mempos = [mod for mod in process.get_mem_modules()]

cnt = 1.0
inloadorder = {}
for mod in loadpos:
    inloadorder[mod.DllBase.v()] = (mod, (cnt / len
       (loadpos)) * 100)
    cnt += 1

cnt = 1.0
ininitorder = {}
for mod in initpos:
    ininitorder[mod.DllBase.v()] = (mod, (cnt / len
       (initpos)) * 100)
    cnt += 1

cnt = 1.0
```

```
        inmemorder = {}
        for mod in mempos:
            inmemorder[mod.DllBase.v()] = (mod, (cnt / len(
                mempos)) * 100)
            cnt += 1

        return inloadorder, ininitorder, inmemorder

    def extract_dll_features(self):
        """Extracts features from each DLL"""

        #Code taken and modified from Volatility's malfind.
            LdrModules
        for process in LdrModules(self.config).calculate():
            inloadorder, ininitorder, inmemorder = self.
                build_dll_dicts(process)
            mapped_files = {}
            for vad, address_space in process.get_vads(
                vad_filter=process._mapped_file_filter):
                #Note this is a lot faster than acquiring
                    the full
```

```
        #vad region and then checking the first two
            bytes.
        #If malware overwrites the PE header (e.g.
           Coreflood) it will be skipped
        if obj.Object("_IMAGE_DOS_HEADER", offset=
           vad.Start, vm=address_space).e_magic != 0
           x5A4D:
            continue

        #Storing VAD node for feature extraction
        mapped_files[int(vad.Start)] = vad


    for base_address in mapped_files.keys():
        # Does the base address exist in the PEB
           DLL lists?
        load_mod, load_pos = inloadorder.get(
           base_address, None) or (None, None)
        init_mod, init_pos = ininitorder.get(
           base_address, None) or (None, None)
```

```
mem_mod, mem_pos = inmemorder.get(
    base_address, None) or (None, None)

dll = DLL()

dll.features['malware_name'] = self.
    imagename
dll.features['process_name'] = str(process.
    ImageFileName) if process else '?'
dll.features['process_id'] = str(int(
    process.UniqueProcessId)) if process else
     '?'
dll.features['dll_base'] = "{0:#x}".format(
    base_address) if base_address else '?'
dll.features['load_count'] = str(hex(
    load_mod.LoadCount)).rstrip('L') if
    load_mod else '?'
dll.features['load_position'] = load_pos if
     load_pos else '?'
dll.features['init_position'] = init_pos if
     init_pos else '?'
```

```
dll.features['mem_position'] = mem_pos if
    mem_pos else '?'

#These features can be extracted from any
    _LDR_DATA_TABLE_ENTRY
if load_mod:
    loadtime = int(load_mod.LoadTime)
    imagesize = str(load_mod.SizeOfImage)
elif init_mod:
    loadtime = int(init_mod.LoadTime)
    imagesize = str(load_mod.SizeOfImage)
elif mem_mod:
    loadtime = int(mem_mod.LoadTime)
    imagesize = str(load_mod.SizeOfImage)
else:
    loadtime = None
    imagesize = None

dll.features['load_time'] = str(loadtime)
    if loadtime else '?'
```

```
dll.features['size_of_image'] = imagesize
    if imagesize else '?'
dll.features['dll_in_load'] = str(not
    load_mod is None)
dll.features['dll_in_init'] = str(not
    init_mod is None)
dll.features['dll_in_mem'] = str(not
    mem_mod is None)
dll.features['dll_mapped_path'] = str(
    mapped_files[base_address].FileObject.
    FileName) if mapped_files else '?'
dll.features['load_full_dll_name'] = str(
    load_mod.FullDllName) if load_mod else
    '?'
dll.features['init_full_dll_name'] = str(
    init_mod.FullDllName) if init_mod else
    '?'
dll.features['mem_full_dll_name'] = str(
    mem_mod.FullDllName) if mem_mod else '?'
```

```
#exports = self.get_exports(load_mod,
   init_mod, mem_mod)
#dll.features['num_exports'] = str(len(
   exports)) if exports else '?'

#imports = self.get_imports(load_mod,
   init_mod, mem_mod)
#dll.features['num_imports'] = str(len(
   imports)) if imports else '?'

#isComServer = str('dllgetclassobject' in
   exports and 'dllcanunloadnow' in exports)
    if exports else None
#dll.features['is_com_server'] =
   isComServer if isComServer else '?'

#isComClient = str('oleinitialize' in
   imports or 'coinitializeex' in imports)
   if imports else None
#dll.features['is_com_client'] =
   isComClient if isComClient else '?'
```

```
vad = mapped_files[base_address]
dll.features['vad_num_section_refs'] = str(
   vad.ControlArea.NumberOfSectionReferences
   ) if vad else '?'
dll.features['vad_type'] = str(vadinfo.
   MI_VAD_TYPE.get(vad.VadFlags.VadType.v(),
    hex(vad.VadFlags.VadType))) if vad else
   '?'
dll.features['vad_num_mapped_views'] = str(
   vad.ControlArea.NumberOfMappedViews) if
   vad else '?'
dll.features['vad_protection'] = str(
   vadinfo.PROTECT_FLAGS.get(vad.VadFlags.
   Protection.v(), hex(vad.VadFlags.
   Protection))) if vad else '?'
dll.features['is_memory_private'] = str('
   PrivateMemory' in str(vad.VadFlags)) if
   vad else '?'
```

```
control_flags = str(vad.ControlArea.u.Flags
   ) if vad else None
dll.features['vad_cf_accessed'] = str('
   Accessed' in control_flags) if vad else
   '?'
dll.features['vad_cf_file'] = str('File' in
    control_flags) if vad else '?'
dll.features['vad_cf_image'] = str('Image'
   in control_flags) if vad else '?'
dll.features['vad_commit_charge'] = int(vad
   .VadFlags.CommitCharge) if vad else '?'
dll.features['vad_allocated_pages'] = ((vad
   .End - vad.Start + 1) / 4096) if vad else
    '?'
dll.features['virtual_address'] = str(hex(
   vad.Start)).rstrip('L') if vad else '?'
dll.features['all_control_flags'] = str(
   control_flags) if vad else '?'

#self.get_pe_features(base_address, process
   .get_process_address_space(), dll)
```

```
            yield dll





#Check for the first three doubly linked lists of
   _LDR_DATA_TABLE_ENTRY structures
#Return the list of exported functions
def get_exports(self, load, init, mem):
    exported_functions = []

    if load:
        for ordinal, function, name in load.exports():
            exported_functions.append(str(name if name
               else ordinal).lower())
        return exported_functions
    if init:
        for ordinal, function, name in init.exports():
```

```
            exported_functions.append(str(name if name
                else ordinal).lower())
        return exported_functions
    if mem:
        for ordinal, function, name in mem.exports():
            exported_functions.append(str(name if name
                else ordinal).lower())
        return exported_functions

    return None

def get_pe_features(self, baseaddr, addrspace, dll):
    try:
        peobj = obj.Object("_IMAGE_DOS_HEADER", offset=
            baseaddr, vm=addrspace)
        ntheader = peobj.get_nt_header()
    except Exception as ex:
        dll.features['has_reloc_section'] = '?'
        dll.features['entrypoint_section_name'] = '?'
        dll.features['entrypoint_virtual_size'] = '?'
        dll.features['entrypoint_raw_size'] = '?'
```

```
        dll.features['dll_flag_set'] = '?'
        dll.features['PE_image_base'] = '?'
        return

    if peobj and ntheader:
        try:
            dll.features['has_reloc_section'] = str(
                self.check_reloc_section(ntheader))
        except Exception as ex:
            print "Exception checking for .reloc: {0}".
                format(ex.message)
            dll.features['has_reloc_section'] = '?'

        try:
            entry_point, virt_size, raw_size = self.
                get_entrypoint_section(ntheader)
            dll.features['entrypoint_section_name'] =
                str(entry_point)
            dll.features['entrypoint_virtual_size'] =
                str(virt_size)
```

```
        dll.features['entrypoint_raw_size'] = str(
            raw_size)

    except Exception as ex:
        "Exception getting entrypoint section:
            {0}".format(ex.message)
        dll.features['entrypoint_section_name'] =
            '?'
        dll.featuress['entrypoint_virtual_size'] =
            '?'
        dll.features['entrypoint_raw_size'] = '?'

    try:
        dll.features['dll_flag_set'] = str(self.
            check_dll_flag(ntheader))
    except Exception as ex:
        print "Exception checking dll flag: {0}".
            format(ex.message)
        dll.features['dll_flag_set'] = '?'

    try:
```

```
            dll.features['PE_image_base'] = self.
                get_image_base(ntheader)
        except Exception as ex:
            print "Exception getting image base: {0}".
                format(ex.message)
            dll.features['PE_image_base'] = '?'

    else:
        dll.features['has_reloc_section'] = '?'
        dll.features['entrypoint_section_name'] = '?'
        dll.features['entrypoint_virtual_size'] = '?'
        dll.features['entrypoint_raw_size'] = '?'
        dll.features['dll_flag_set'] = '?'
        dll.features['PE_image_base'] = '?'



#Check if the module contains a .reloc section. True if
    it does, False if not
def check_reloc_section(self, ntheader):
    for sec in ntheader.get_sections():
```

```
        if sec.Name.lower() == '.reloc':
            return True
    return False

#Find the entrypoint for the module and return the
   section containing it
def get_entrypoint_section(self, ntheader):
    entrypoint = int(ntheader.OptionalHeader.
       AddressOfEntryPoint) if ntheader.OptionalHeader
       else None

    if entrypoint:
        for sec in ntheader.get_sections():
            if entrypoint >= int(sec.VirtualAddress)
               and entrypoint <= (int(sec.VirtualAddress
               ) + int(sec.Misc.VirtualSize)):
                return sec.Name, sec.Misc.VirtualSize,
                   sec.SizeOfRawData
    return '?', '?', '?'
```

```
#Check if the module is has the DLL flag set in the pe
   header. True if it does, False if not
def check_dll_flag(self, ntheader):
    if ntheader.FileHeader:
        if ntheader.FileHeader.Characteristics & 0x2000
             == 0:
            return False
        else:
            return True

#A faster way to get the number of exports the actual
   exported functions are not necessary
def get_num_exports(self, _ntheader, dllbase, addrspace
   ):
    if _ntheader:
        data_dir = _ntheader.OptionalHeader.
           DataDirectory[0]
        #invalid data directory; effectively no export
           directory exists
        if data_dir.VirtualAddress == 0 or data_dir.
           Size == 0:
```

```
            return '?'
        #values do not make sense; effectively no
          export directory exists
        if data_dir.VirtualAddress + data_dir.Size >
          _ntheader.OptionalHeader.SizeOfImage:
            return '?'

        expdir = obj.Object('_IMAGE_EXPORT_DIRECTORY',
                            offset = dllbase + data_dir
                              .VirtualAddress,
                            vm = addrspace)

        if expdir.valid(_ntheader):
            return int(expdir.NumberOfFunctions)

        return '?'

def get_imports(self, load, init, mem):
    imported_functions = []
    if load:
```

```
        for _, ordinal, function, name in load.imports
            ():
              imported_functions.append(str(name if name
                  else ordinal).lower())
        return imported_functions
    if init:
        for _, ordinal, function, name in init.imports
            ():
              imported_functions.append(str(name if name
                  else ordinal).lower())
        return imported_functions
    if mem:
        for _, ordinal, function, name in mem.imports()
            :
              imported_functions.append(str(name if name
                  else ordinal).lower())
        return imported_functions
    return None

def get_image_base(self, ntheader):
    if ntheader.OptionalHeader:
```

```
            return str(hex(ntheader.OptionalHeader.
                ImageBase)).rstrip('L')




if __name__ == "__main__":
    extractor = vae('Z:\\Malware Samples\\Memory Samples\\
        injectedwin7.vmem', 'Win7SP1x86')

    dl = DLLList()
    for dll in extractor.extract_dll_features():
        dl.append(dll)

    dl.write_to_csv('Z:\\Projects\\Dissertation\\Software
        for Appendices\\csv.csv')
```

APPENDIX C

DATA PREPROCESSING SOURCE CODE

C.1 Python Code for Preprocessing Dataset

```
import csv
import os
import collections
import sys



class PreProcesser():
    def __init__(self, input_filename, output_filename,
       columns):
        self.input_csv_filename = input_filename
        self.output_csv_filename = output_filename
        self.csv_headers = None
        self.csv_rows = None
        self.columns_to_remove = columns



    def process(self):

        with open(self.input_csv_filename, "rb") as
            csv_file:
```

```
csv_reader = csv.DictReader(csv_file, dialect='
   excel', quoting=csv.QUOTE_NONNUMERIC)

self.csv_headers = csv_reader.fieldnames

if not self.csv_headers or self.csv_headers ==
   []:
     return False

self.csv_rows = [row for row in csv_reader]

if not self.csv_rows or len(self.csv_rows) ==
   0:
     return False

self.total_entries = len(self.csv_rows)

for row in self.csv_rows:
     self.add_raw_virt_size_diff(row)
     self.add_load_path_is_temp(row)
     self.add_file_extension_is_dll(row)
```

```
            self.add_is_dkom_present(row)
            self.add_vad_path_ldr_path_differ(row)
            self.change_dll_base_to_TF(row)
            self.change_entrypoint_to_TF(row)
            self.change_load_count_to_int(row)
            self.change_PE_image_base_to_TF(row)
            self.change_process_name(row)

        headers_to_add = ['load_path_is_temp', '
           file_extension_is_dll', 'is_dkom_present', '
           vad_path_ldr_path_differ', '
           raw_virt_size_diff']
        for header in headers_to_add:
            if header not in self.csv_headers:
                self.csv_headers.insert(0, header)

        self.csv_headers = sorted(self.csv_headers, key
           =lambda k: k.lower())

    with open(self.output_csv_filename, 'wb') as
       csv_output:
```

```
        csv_writer = csv.DictWriter(csv_output, dialect
           ='excel', quoting=csv.QUOTE_NONNUMERIC,
           fieldnames=self.csv_headers)
        csv_writer.writeheader()
        for row in self.csv_rows:
            csv_writer.writerow(row)




#Changes the attribute dll_base from a memory address
   to a value of TRUE or FALSE
#   TRUE - The address is the default base address for
   DLLs (0x10000000)
#   FALSE - The address is something other than 0
   x10000000
def change_dll_base_to_TF(row):
    if row['dll_base'].lower() == 'TRUE' or row['
       dll_base'].lower() == 'FALSE':
        return
    if row['dll_base'].lower() == '0x10000000':
```

```
        row['dll_base'] = 'True'
    else:
        row['dll_base'] = 'False'


#Changes the attribute entrypoint_section_name from the
    name to TRUE or FALSE
#   TRUE - The entrypoint section name is set to .text
#   FALSE - The entrypoint section name is set to
   something other than .text
def change_entrypoint_to_TF(row):
    if row['entrypoint_section_name'].lower() == 'TRUE'
        or row['entrypoint_section_name'].lower() == '
      FALSE':
        return
    if row['entrypoint_section_name'].lower() == '.text
      ':
        row['entrypoint_section_name'] = 'True'
    else:
        row['entrypoint_section_name'] = 'False'
```

```
#Changes the load_count attribute to an integer
   representation
def change_load_count_to_int(row):
    if type(row['load_count']) is str:
        if row['load_count'].lower() != '?':
            row['load_count'] = int(row['load_count'],
               16)


#Changes the PE_image_base attribute to TRUE or FALSE
def change_PE_image_base_to_TF(row):
    if row['PE_image_base'].lower() == 'TRUE' or row['
       PE_image_base'].lower() == 'FALSE':
        return
    if row['PE_image_base'].lower() == '0x10000000':
        row['PE_image_base'] = 'True'
    else:
        row['PE_image_base'] = 'False'


#Changes the process_name attribute to other if it is
   not explorer.exe, svchost.exe, or taskhost.exe
def change_process_name(row):
```

```
    if row['process_name'].lower() != 'explorer.exe'
       and row['process_name'].lower() != 'svchost.exe'
       and row['process_name'].lower() != 'taskhost.exe
       ':
        row['process_name'] = 'other'


#Adds a new attribute load_path_is_temp
def add_load_path_is_temp(row):
    load_path = row['dll_mapped_path'].split('\\')
    row['load_path_is_temp'] = 'False'
    for d in load_path:
        if 'temp' == d.lower():
            row['load_path_is_temp'] = 'True'

#Adds a new attribute file_extension_is_dll
def add_file_extension_is_dll(row):
    if row['dll_mapped_path'].endswith('.dll'):
        row['file_extension_is_dll'] = 'True'
    else:
        row['file_extension_is_dll'] = 'False'
```

```
#Adds a new attribute is_dkom_present
def add_is_dkom_present(row):
    if row['dll_in_init'].lower() == 'false' or row['
       dll_in_load'].lower() == 'false' or row['
       dll_in_mem'].lower() == 'false':
        row['is_dkom_present'] = 'True'
    else:
        row['is_dkom_present'] = 'False'

#Adds a new attribute vad_path_ldr_path_differ
def add_vad_path_ldr_path_differ(row):
    if row['init_full_dll_name'].lower() == row['
       load_full_dll_name'].lower() and row['
       load_full_dll_name'].lower() == row['
       mem_full_dll_name'].lower():
        if 'c:' + row['dll_mapped_path'].lower() == row
           ['load_full_dll_name'].lower():
            row['vad_path_ldr_path_differ'] = 'False'
        else:
            row['vad_path_ldr_path_differ'] = 'True'
```

```
    else:
        row['vad_path_ldr_path_differ'] = '?'


#Adds a new attribute to measure size of difference
   between entrypoint_raw_size and
   entrypoint_virtual_size
#Can be {Neg, None, Low, High}
def add_raw_virt_size_diff(row):
    try:
        diff = float(row['entrypoint_virtual_size']) -
           float(row['entrypoint_raw_size'])
    except ValueError:
        row['raw_virt_size_diff'] = '?'
        return
    if diff < 0:
        row['raw_virt_size_diff'] = 'Neg'
        return
    if diff == 0:
        row['raw_virt_size_diff'] = 'None'
        return
    #If size is less than the size of a page
```

```
        if abs(diff) < 4096:
            row['raw_virt_size_diff'] = 'Low'
            return
        #If size is less than the size of a page
        if abs(diff) >= 4096:
            row['raw_virt_size_diff'] = 'High'
            return



    def remove_column(self):
        print "Removing", self.columns_to_remove, "from",
            str(self.input_csv_filename), "saving to", str(
            self.output_csv_filename)
        with open(self.input_csv_filename, "rb") as
            input_csv_file:
            csv_reader = csv.reader(input_csv_file, dialect
                ='excel', quoting=csv.QUOTE_NONNUMERIC)
            headers = csv_reader.next()
            rows_to_write = [row for row in csv_reader]

            for col in self.columns_to_remove:
```

```
    remove_column_index = -1
    try:
        remove_column_index = headers.index(col
            )
        headers.remove(col)
    except ValueError:
        print "Column (" + str(col) + ") does
            not exist in the input file:", str(
            self.input_csv_filename)
        continue
    for row in rows_to_write:
        row.pop(remove_column_index)

with open(self.output_csv_filename, "wb") as
    output_csv_file:
    csv_writer = csv.writer(output_csv_file,
        dialect='excel', quoting=csv.
        QUOTE_NONNUMERIC)
    csv_writer.writerow(headers)
    for row in rows_to_write:
        csv_writer.writerow(row)
```

```
#Build a dictionary mapping headers to their respective
    types (nominal, numeric)
def get_attribute_types(self):
    self.types = {}
    for header in self.csv_headers:
        types = set()
        for row in self.csv_rows:
            try:
                float(row[header])
                self.types[header] = 'numeric'
                break
            except:
                if type(row[header]) is str and row[
                   header] != '?':
                    types.add(row[header])
        if types:
            self.types[header] = sorted(list(types))
```

```
def convert_to_arff(self):
    arff_output_file = self.output_csv_filename.strip
        ('.csv') + '.arff'
    with open(self.input_csv_filename, "rb") as
        csv_file:
        csv_reader = csv.DictReader(csv_file, dialect='
            excel', quoting=csv.QUOTE_NONNUMERIC)
        self.csv_headers = csv_reader.fieldnames

        if not self.csv_headers or self.csv_headers ==
            []:
            return False

        self.csv_rows = [row for row in csv_reader]

    with open(arff_output_file, 'w') as
        output_arff_file:
        output_arff_file.write("@relation {0}\n\n".
            format(arff_output_file.split('\\')[-1].strip
            ('.arff')))
        self.get_attribute_types()
```

```
            for header in self.csv_headers:
                output_arff_file.write("@attribute {0} ".
                    format(header))
                if self.types[header] == 'numeric':
                    output_arff_file.write("numeric\n")
                else:
                    output_arff_file.write("{" + "{0}".
                        format(",".join(self.types[header]))
                        + "}\n")
            output_arff_file.write("\n@data\n")
            for row in self.csv_rows:
                output_arff_file.write("{0}".format(",".
                    join([str(row[header]) for header in self
                    .csv_headers])))
                output_arff_file.write("\n")


def get_arguments():
    import argparse

    parser = argparse.ArgumentParser()
```

```
    parser.add_argument("input_csv", help="Path to the
       input csv")
    parser.add_argument("output_csv", help="File to store
       processed data")
    #parser.add_argument("-c", help="Column to remove.
       Comma-separated if more than one", action="store_true
       ")

    arguments = parser.parse_args()

    return arguments

if __name__ == '__main__':
    args = get_arguments()
    columns = ['virtual_address', 'mem_full_dll_name', '
       malware_name', 'all_control_flags', 'dll_mapped_path
       ', \
                'init_full_dll_name', 'is_memory_private', '
                   load_full_dll_name', 'phys_page_addr', '
                   process_id', \
```

```
            'vad_cf_file', 'load_time', 'mem_position',
              'entrypoint_raw_size', '
              entrypoint_virtual_size' ]

m = PreProcesser(args.input_csv, args.output_csv,
   columns)
m.process()
m.input_csv_filename = args.output_csv
try:
    m.remove_column()
except:
    print "Could not remove columns"
m.convert_to_arff()
```