VISUALISATION IN MEMORY ANALYSIS

By

Paul Fowler

Supervisor(s): Peter Alexander

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
M.SC. IN APPLIED CYBER SECURITY
AT
INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN
DUBLIN, IRELAND
31 AUGUST 2018

Thesis Declaration Form

TO BE COMPLETED IN FULL BEFORE SUBMISSION TO SCHOOL OFFICE

Student Name	Paul Fowler
Student Number	B00108036
Assessment Title	Visualisation in Memory Analysis
Module Code	BN528
Module Title	MSc Research Project
Module Coordinator	Peter Alexander
Tutor (if applicable)	

A SIGNED COPY OF THIS FORM MUST ACCOMPANY ALL SUBMISSIONS FOR ASSESSMENT. STUDENTS SHOULD KEEP A COPY OF ALL WORK SUBMITTED.

Plagiarism: the unacknowledged inclusion of another person's writings or ideas or works, in any formally presented work (including essays, examinations, projects, laboratory reports or presentations). The penalties associated with plagiarism designed to impose sanctions that reflect the seriousness of Institute's commitment to academic integrity. Ensure that you have read the Institute's Plagiarism Policy and Procedures.

Declaration of Authorship

I declare that the work contained in this submission is my own work, and has not been taken from the work of others. Any sources cited have been acknowledged within the text of this submission.

I have read and understood the policy regarding plagiarism in the Institute of Technology Blanchardstown.

Signed..... Date .....

Abstract

The huge number of obfuscated malware threats that are introduced and detected every day has resulted in what seems like an endless cycle of attacks on individuals and organisations worldwide. Malware authors are constantly developing and evolving obfuscation techniques to evade anti-virus solutions, intrusion detection and other preventative systems. Previous research in the visualisation of malware using static, dynamic or reverse engineering showed that heavily obfuscated malware variants are not easily recognisable without first de-obfuscating the malware. To detect obfuscated malware, this study explores the detection of obfuscated malware families using visualisation in memory analysis.

A collection of some of the more popular malware family's variants are examined and visualised. The malware sample is executed, and a memory dump is captured. The executable is extracted and converted into a grayscale image. Features are extracted from the image and used to build a data model. Using machine pattern analysis techniques new malware variants can then be identified as been part of a malware family.

Results show that this method is a feasible approach to aid malware researchers to detect heavily obfuscated malware variants and to identify them as belonging to a particular malware family

Acknowledgements

I would like to thank Peter Alexander, my thesis supervisor for guiding me through this process, being available to answer any questions and pointing me in the right direction. I would like to acknowledge Stephen O'Shaughnessy for providing me with the idea for this thesis and for initial guidance. Finally, I would like to express my appreciation to my partner Leanne for her constant support and encouragement throughout the thesis and the entire course.

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

List of Tables viii

List of Figures x

Abbreviations xii

1 Introduction 1

1.1 Research Hypothesis 3

1.2 Research Questions 3

1.3 Document Road Map 4

1.4 Research Approach 4

2 Literature Review 6

2.1 Introduction 6

2.2 Obfuscation 7

2.2.1 Encryption 8
2.2.2 Polymorphic 8
2.2.3 Metamorphic 9
2.3 Obfuscation development 9
2.4 Visualisation 10
2.4.1 Static & Dynamic Analysis 10
2.4.2 Reverse Engineering 19
2.4.3 Behaviour Analysis 23
2.5 Memory Analysis 25
2.5.1 Trigger based memory analysis 25
2.5.2 Feature Memory Extraction 26
2.5.3 Fileless Malware 27
2.6 Conclusion 28

3 Design & Methodology **30**

3.1 Introduction 30
3.2 Restrictions & Limitations 31
3.3 Identify Data Type 32
3.3.1 Data Source 33
3.3.2 Identify Malware Families 33
3.3.3 Extracting Data 37
3.4 Lab Envoirnment 38
3.4.1 Cuckoo Sandbox 41
3.4.2 Collect trigger-based memory dumps 42

3.4.3 Naming Scheme . . . 43
3.5 Convert the dumps to grayscale images . . . 44
3.6 Extract features from images . . . 47
3.7 Weka . . . 48
3.7.1 Machine Learning Algorithms . . . 49
3.7.2 Pre-process Data . . . 49
3.7.3 Create data models using machine learning algorithms . . . 51
3.7.4 Measure to what degree new malware samples can be identified . . . 52
3.8 Test against the state-of-art . . . 52
3.8.1 Data Setup . . . 53
3.8.2 Accuracy Calculation . . . 56
3.9 Conclusion . . . 57

4 Implementation **58**

4.1 Experiment 1 . . . 58
4.1.1 Question . . . 58
4.1.2 Background . . . 58
4.1.3 Experiment 1.1 - Mirai . . . 59
4.1.4 Experiment 1.2 - Locky . . . 62
4.1.5 Summary . . . 66
4.2 Experiment 2 . . . 66
4.2.1 Question . . . 66
4.2.2 Background . . . 66
4.2.3 Preliminary Test Run . . . 67

4.2.4 Prepare Training and Test Models 70
4.2.5 Compare Training and Test Models 72
4.2.6 Comparing Classification Errors 74
4.2.7 Comparing Precision Variance 76
4.2.8 Summary 78
4.3 Experiment 3 79
4.3.1 Question 79
4.3.2 Background 79
4.3.3 Experiment 79
4.3.4 Summary 83
4.4 Experiment 4 83
4.4.1 Question 83
4.4.2 Background 84
4.4.3 Experiment 84
4.4.4 Summary 87

5 Discussion and Analysis **88**

6 Conclusion **92**

6.1 Future Work 93

Bibliography **95**

Appendices **103**

A Full Code of VirusTotal API vtcheck 103
B VirusTotal Hash check results 104

List of Tables

3.1 Visualisation of Packed and Obfuscated Samples 51

4.1 Mirai Samples Details 60

4.2 Mirai Static Visualisations 61

4.3 Mirai Memory Visualisations 61

4.4 Locky Sample Details 62

4.5 Locky Static Visualisations 63

4.6 Locky Memory Visualisations 64

4.7 Visualisation of de-obfuscation of Locky 65

4.8 Malware families 67

4.9 Explorer Injected Samples 69

4.10 Process Injection into explorer.exe 70

4.11 Training Samples Split 71

4.12 K-NN Test Result 73

4.13 K-NN Test Confusion Matrix 73

4.14 SVM Test Result 74

4.15 SVM Test Confusion Matrix 74

4.16 Visual difference 76

4.17 Ramnit Precision Variance 78

4.18 Trigger Based K-NN Test Result 81
4.19 Trigger Based K-NN Test Confusion Matrix 81
4.20 Trigger Based SVM Test Result 82
4.21 Trigger Based SVM Test Confusion Matrix 82
4.22 Locky and Trickster comparison 83
4.23 K-NN Accuracy Calculation 86
4.24 SVM Accuracy Calculation 86
4.25 Accuracy Comparison 87

List of Figures

2.1 Graphical representation of Adultbrowser malware using a treemap 11

2.2 Graphical representation of Adultbrowser malware using a thread graph . . 11

2.3 Byteplot Visualisation 12

2.4 Examples of grayscale images 13

2.5 Entropy Graph of Trojan-Dropper.Win32.Del 15

2.6 Regrun.rk Entropy Image 16

2.7 Regrun.yi Entropy Image 16

2.8 Entropy Pixel Image of Regrun family 16

2.9 Win32.FlyAgent.jt 17

2.10 Calc.exe 17

2.11 RGB Malware Visualisation of Backdoor.Win32.Hupogon family 18

2.12 Close-up of the MEW unpacking loop 20

2.13 Markov Byteplot: Packed and Unpacked versions of Beagle malware 21

2.14 Image Matrix of Win32.IstBar family 21

2.15 Similarity matrix: Win32.Dadobra vs Win32.Delf malware families 23

2.16 Malware behaviour image sample for 3 different types of malware 24

3.1 Procedure Overview 31

3.2 PE Sections of a File 32

3.3 Lab Setup 40
3.4 GIST Descriptor tool execution 47
3.5 Example of GIST.txt 48
3.6 Formated data for Weka 48
3.7 Find MD5 from Test set 54

4.1 bintoIm.py execution 60
4.2 Explorer plotted 68
4.3 Explorer plotted against Hupigon 68
4.4 Training Model in Weka 72
4.5 Visual classifier error 75
4.6 Ramnit PrecisionVariance 77
4.7 Trigger-Based Training Model in Weka 80
4.8 VirusTotal MD5 Results Check 85

Abbreviations

API	Application Programming Interface
DDOS	Distributed Denial of Service
DKOM	Direct Kernel Object Manipulation
FN	False Negative
FP	False Positive
IDS	Intrusion Detection System
IoT	Internet of things
K-NN	K - Nearest Neighbour
LCS	Lowest Common Subsequence
PE	Portable Executable
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
UPX	Ultimate Packer for Executables
VAD	Virtual Address Descriptors
VERA	Visualization of Executables for Reversing and Analysis
WEKA	Waitkato Environment for Knowledge Analysis

Chapter 1

Introduction

Due to the rapid proliferation of malware, there is an exponential increase in the number of new malware signatures released every year. In 2017 alone Symantec's study completed by Cleary et al. (2018) reported that there were 669,947,865 new malware variants. Even though 78% of that figure was that of the evolving Kotver Trojan variants, it has been proven that the number of new malware variants have been increasing year on year. This huge number alone shows the huge challenge it has become for anti-virus solution providers to keep on top of this challenging environment.

Antivirus solution providers currently use various methods to detect malware threats. Bazrafshan et al. (2013) discusses three methods currently used by anti-virus solution providers to detect malware. The signature-based approach is where a software solution uses a finger-print of the malware in order to detect any malicious payloads. These signatures are like rules that are used to identify different groups of known malware types. The author states that the weakness with this rule-based approach is that this process is unable to detect new malware variants. As a result, the signature-based approach has proven to be insufficient when solely used as new unique malware binaries are discovered every day.

The behaviour-based approach detects objects within programs and tries to determine the objects potential behaviour. Evaluating behaviour as the malware executes is called dynamic analysis. Determining a potential threat without executing but by assessing threat potential

by automated or manual means is called static analysis. Wagener and Dulaunoy (2008) discusses in more detail how behaviour analysis is used to determine how a sample runs by analysing and understanding malware activities relating to registry changes, file system changes, analysing events like system or API calls, execution history and network events. Bazrafshan discusses the weakness with behaviour approach in that if a malware sample has environment detection built in, it may be able to detect that it is in a sandbox environment and therefore it may not execute. The malware sample may also have snippets of code that are obfuscated so static analysis would also provide little information.

The third approach discussed by the Bazrafshan and used by anti-virus solution providers is a heuristic based approach. Heuristic scanning uses data mining methods, rules and algorithms to looks for commands, code patterns and grouped API functions which may indicate malicious intent. Ye et al. (2018) discusses in further detail how malware classifiers are built by extracting features like byte sequences, instruction sequences and API call sequences by applying reverse engineering techniques to captured binaries, email or network traffic. The reverse engineering process is often a manual process and time consuming. However the data gathered can be easily classified to enable new malware variant detection. Using multiple techniques like misuse detection and anomaly detection of the extracted features enables the detection of new malware variants with the same file features.

Most anti-virus solutions programs today use a mix of both signature and heuristic methods to detect malware. The weakness with the heuristic based approach is that if the malicious code is obfuscated successfully then it will evade detection. Using the heuristic approach also has a higher tendency to report false positives as this approach looks for a wide range of actions that may be used by legitimate programs. As a result, new malware detection techniques need to be developed to cope with the constantly changing malware eco system and to specifically deal with changing obfuscated malware.

Information visualisation has been used for some time in data mining and in more recent time visual analytics has been used in machine learning. Visualisation techniques for similarity detection of features in malware have been developed for static, dynamic, behavioural

and reverse engineering analysis. However, for heavily obfuscated malware this technique is known to give mixed results.

When a program or in particular malware, is executed it first needs to be unencrypted and de-obfuscated in memory in order to trigger sequences of simple machine-language instructions that allow the program to convert to a binary representation which a CPU can understand. As a result, a true picture of the program without obfuscation can be seen in memory. This study explores how visualising malware in memory can aid researchers to detect and identify malware families that are heavily obfuscated.

1.1 Research Hypothesis

Current malware visualisation research concentrated on static, dynamic or reverse engineering of malware. There have been various visualisation studies completed which have shown that heavily obfuscated malware variants are not easily identifiable.

The research hypothesis of this study is that by visualising malware in memory will enable unique features to be extracted which can facilitate the identification of new malware variants.

1.2 Research Questions

Question A: Can new malware variants be identified using computer vision pattern identity after extracting malware from a memory dump?

Question B: Does using visualisation of API trigger based analysis impove pattern identity of malware?

Question C: How does state of the art solutions stand up to computer vision model-based analysis?

1.3 Document Road Map

Chapter 1 An overall introduction to the study including the subject being researched

Chapter 2: Takes a look at the body of research on obfuscation, visualisation and memory analysis. This chapter explores the different visualisation techniques and methods used and their strengths and weaknesses are examined.

Chapter 3: Discusses the resaerch approach and design used in the study. This includes data gathering, envorinment setup and implmentation.

Chapter 4: Defines the experiments in this study and a runthrough of each. It also describes the technique and method chosen for each experiment.

Chapter 5: Discusses the final results.

Chapter 6: The conclusion of the study and related future work

1.4 Research Approach

The approach taken for this research thesis will be quantitative in nature. The basis of the study is built on previous research completed by Nataraj et al. (2011) on static visualisation of malware and Teller and Hayon (2014) on trigger-based memory analysis. The research approach in this study is designed to develop on the previous research with the aim to identify obfuscated malware variants using visualisation in memory analysis. The malware families used are some of the most popular identified by Symantec in 2017 and obtained from VirusTotal. Each of the malware variants used in the study use a broad range of obfuscation techniques where it is difficult to identify malware using other visualisation techniques. The data is analysed visually, GIST features are extracted and using pattern matching algorithms new samples are tested against trained models to identify new malware variants. The sample data set is limited in size due to the obtained small data set, the inability to control the obfuscation techniques used by malware variants and the time-consuming static analysis before or after analysis. The research approach is discussed further in Chapter 3.

There have been four different experiments designed with the intent of producing identifiable patterns of malware variants using visualisation of memory analysis and pattern matching techniques.

Experiment 1: Repeats the techniques used by Nataraj to identify if new malware variants can be identified from malware families using memory analysis visualisation.

Experiment 2: Using visualised data from experiment 1, create data models using GIST descriptor and using pattern matching analysis check to see if new malware samples can be identified.

Experiment 3: Using visualisation of API trigger-based memory analysis, determine if new malware variants can be identified using pattern matching analysis.

Experiment 4: Using pattern matching analysis, to measure how the created classification models compare to the current state of the art solutions.

Chapter 2

Literature Review

2.1 Introduction

Automated tools are designed to quickly evaluate malware in a sandbox environment. While the ability of automated tools to asses heavily obfuscated malware does offer enormous benefits to aid analysis, they do not offer as much of an in-depth analysis as a skilled human malware analyst. As a result, the process of analysing heavily obfuscated malware samples manually using static, dynamic or reverse engineering cannot be circumvented. Manual analysis of malware can be time consuming, so new techniques must be developed to aid malware analysis and detection.

Most anti-virus solutions use signature-based and heuristic based detection mechanisms to detect malware. To avoid the detection there are numerous obfuscation techniques used by malware authors to obscure malware from detection systems and manual analysis. While legally obfuscation is used to reduce the ability of reverse engineering programs in order to stop illegal reproduction of software, it is also used by malware authors to avoid detection.

There are numerous malware detection techniques and classification methods proposed to identify malware. These include graph-based, instruction sequence based, instruction frequency based, API call monitoring and behaviour-based methods. Each of these methods have difficulties when trying to detect new malware variants.

A method that has potential to aid malware detection is using visualisation to detect malware variants. Malware visualisation focuses on representing malware in a visual form. Research has shown that the developed techniques allow malware analysts to better understand malware graphically, by highlighting visual patterns in a form that may not be detected using static and dynamic analysis or reverse engineering methods. A survey completed by Wagner et al. (2015) on previous research of visualisation systems for malware analysis showed how existing visualisation techniques used on malware can generate similar images for malware variants from the same family. This occurs because the same code, functions, features and characteristics are often the same in these malware samples and as a result the samples can be detected visually.

This chapter will give an overview of how malware analysis can be obstructed when obfuscation techniques are implemented by malware authors. An analysis is completed on the various techniques used by researchers to accomplish visualisation of malware and on some methods currently used in machine learning in malware and memory analysis to identify malware.

2.2 Obfuscation

Protection techniques for software have been used for many years by software providers to protect intellectual property. Code obfuscation, software watermarking and fingerprinting, tamper proofing and birth marking are some of the techniques used to prevent malicious abuse of their intellectual property. These obfuscation techniques were used to reduce the ability to reverse engineer programs and to stop illegal reproduction of software. In recent times malware authors have used obfuscation methods and techniques to avoid detection by anti-virus solutions, intrusion detection systems and malware analysts. You and Yim (2010) discuss the various obfuscation techniques used by malware.

2.2.1 Encryption

One of the first methods used by malware authors to evade signature based anti-virus solutions was to use encryption. The authors discuss where encrypted malware evades signature-based detection by encrypting the body of code with a different encryption key every time it infects a system. This behaviour enables the malware to hide its signature as every time the program is executed the malware will have a different signature and therefore the signature-based solution can no longer detect the infection. The weakness of this approach was that the encryption program remained constant, so detection systems could be evolve to also scan for signatures of certain encryption code patterns in programs.

2.2.2 Polymorphic

To overcome the deficiency of encrypted malware, malware authors designed a method to evolve the decryptor from one generation to the next. The authors discuss the first attempt where the decryptor was changed only slightly and as a result both the malware and its decryptor could have its signature changed and thus thwarting anti-virus scanners temporarily. The weakness with this oligomorphic malware was that there were only a couple of hundred decryptors so eventually all the decryptors could be detected with signature-based scanners.

The authors discuss the next development in malware called polymorphic malware. This was where malware authors introduced obfuscation methods like dead-code insertion into the decryptors and therefore there could be an endless number of distinct decryptors. Even though polymorphic malware could evade signature-based systems, when the malware was executed the body of code remained constant. As a result, anti-virus solutions evolved and developed emulation techniques where the malware is executed in a simulated environment. Once the body of code is decrypted and loaded into memory the signature-based solutions could be applied once more.

2.2.3 Metamorphic

The authors discuss further development by malware authors where malware could essentially reorganise its entire code structure but yet function the same. In this case the malware is rewritten when it is executed so that a new variant of the malware is created for each infection which as a result circumvents signature-based detection. Some techniques include adding random useless instructions or loops to code, varying the lengths of no-operation instructions or reordering the functions. So as signature-based detection is becoming less important for new malware variants, the current challenge lies in how to detect heavily obfuscated malware variants.

2.3 Obfuscation development

Obfuscation development techniques are been driven by new attack vectors and the ability to obstruct analysis efforts. The longer an analysis takes the longer malicious activity goes undetected and therefore leads to the infection of many more machines. Cabaj et al. (2017) analysed the Locky ransomware variants. The authors compared Locky's earliest samples to its most recent. Initially Locky was a two-stage attack vector where the first stage included a macro written in seventeen lines of code in JavaScript, which was delivered via word or excel documents. The first stage called an unencrypted URL to download the second malicious stage which included the ransomware. The file size varied up to 1mb in size with the varying size usually due to various obfuscation methods being put in place as the malware developed over time. Obfuscated variable names, unicode encoding, concatenated code, dumb code are some of the obfuscated techniques that were used over time. The more recent versions had encrypted code in both first and second stage.

The second stage included an additional entry function that was not included in the exports table but would be called from the encrypted text from the first stage. This entry function was dynamically decrypted so it could only be detected via dynamic analysis. What the authors found was that the cycle between malware variants was decreasing. Out of over

5,900 variants of Locky detected there was sometimes 2 or 3 variants of Locky found in the same day. With these rapid changes they concluded that sometimes the significant obfuscation changes made to the malware variants made dynamic analysis not always viable. As a result the authors had to upgrade the analysis environment in order to be able to able to execute the analysis of the samples.

Locky is a good example of obfuscation development throughout its lifecycle. The authors found that despite the constant obfuscation changes made to various Locky variants, the core code for Locky remained similar over the course of its life. It is these similarities that the idea of visualisation of programs and therefore malware has been used in studies to help identify and detect malware.

2.4 Visualisation

2.4.1 Static & Dynamic Analysis

Trinius et al. (2009) proposed two visualisation techniques for malware behaviour. The authors work was based on Xia et al. (2008) visual analysis of program flow data. The authors began by executing the malware in CWSandbox. The authors grouped the detected API calls into specific functional groups and then used the gathered behaviour-based analysis for visualisation. The authors introduced two visualisation techniques called the Malware Tree Map and the Malware thread graph.

The Malware Tree map (Figure 2.1) technique was where malware behaviour is visualised in the form of nested rectangles where the size of the rectangle is determined by the proportion of API calls in that section of code. Each rectangle is further divided into the operation of the API. This behaviour-based technique obtains the API calls by executing the malware sample in a virtual environment.

The second technique also proposed by the authors is called malware thread graph (Figure 2.2). It works by plotting the activity for each thread in a graph. If a malware sample

has multiple threads, then multiple lines are plotted on the graph. Each thread can easily be examined to identify where the majority of operations are made. This technique also needs to be executed in a virtual environment to obtain the API calls. By being able to see the function of a thread it gives analysts the ability to be able to visualise unique behaviour of a certain family of malware by matching the thread patterns.

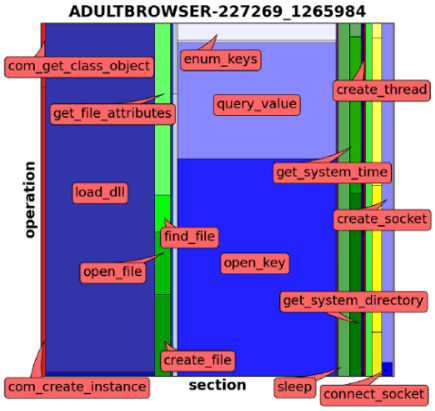


Figure 2.1: Graphical representation of Adultbrowser malware using a treemap

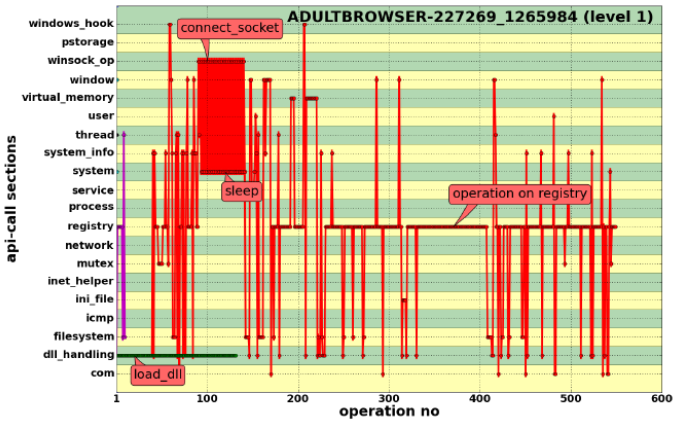


Figure 2.2: Graphical representation of Adultbrowser malware using a thread graph

Trinus et al. provided the first visual analysis of malware using dynamic analysis. The problem with both these behaviour-based techniques is that only one execution path of a binary can be analysed. These techniques do not have the ability to detect the presence of analysis environment detecting code. The malware can behave and execute in a different way and can therefore skew the results.

Conti et al. (2008) introduced texture based graphical techniques to support reverse engineering by human analysts of binary and data files using visualisation. The authors were able to visualise raw binary data such as text, image data or audio data as images and then classify the data using statistical structural features. The authors scanned every raw byte and converted each value into an image pixel.

The first byte in the file is located in the top left corner, coordinate (1,1), The next byte is

displayed at position (2,1) and so on. The program can be limited to a specific width of pixels and the height is then dependent on the size of the file. When the end of a line is reached, plotting begins on the next line until all bytes have been plotted. The colour of each pixel maps to the value of the byte displayed. The byte value 00 is black (0) and the byte value FF is white (255) with varying shades of grey in between. In Figure 2.3 there is a example of the first 4 bytes from the command prompt from a windows 7 x86_32 bit binary.

4D 5A 90 00

When these bytes are converted to decimal you get the values:

77, 90, 140, 00

Each byte is mapped to a shade between the values 0 and 255.

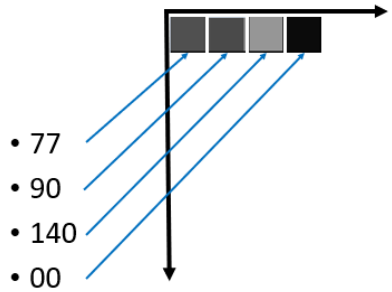


Figure 2.3: Byteplot Visualisation

These values are then plotted to a pixel and placed in the next available position in the file. The visual representation of a file showed that texture patterns could be identified. The idea and technique of using texture features in visualisation was one of the starting points for other researchers to start to look at visualisation as a tool that could potentially aid malware analysis.

Nataraj et al. (2011) built on the research from Conti et al. and found that malware families belonging to the same families had similar layouts and texture and that this similarity

could potentially be detected in a visual form. The authors proposed and developed a method for detecting malware visually using image processing techniques and pattern matching algorithms. The authors used Conti's techniques for converting binary files to images in order to convert malware samples to grey scale images (Figure 2.4). The authors then extracted texture features by computing a GIST descriptor for each image.

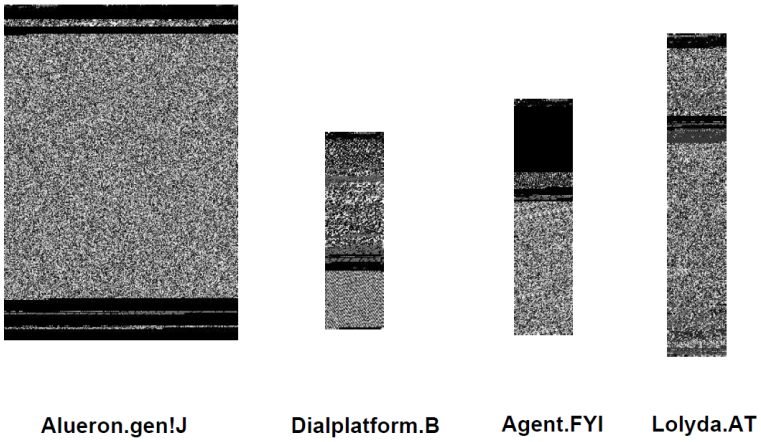


Figure 2.4: Examples of grayscale images

The GIST descriptor is based on a study completed by Oliva and Torralba (2001*b*) where calculated low dimensional image features are used for scene recognition and classification such as deciphering mountains, forests or a street. The calculated GIST global descriptor provides a statistical representation of the features in an image which can then easily be used for texture classification.

Using GIST to extract texture features allowed Nataraj to summarise the gradient information of each image which then enabled the authors to classify the images. Finally, the authors used K-Nearest Neighbour algorithm for feature pattern analysis. The results allowed the authors to observe malware families where there was a similar layout and texture classified via GIST classification method. This method allowed the authors to be able to also detect malware where only sections of code were encrypted.

The weakness with this approach was that a malware author who knows this technique

can easily obfuscate their malware by using a polymorphic engine which can add redundant code or null pointers to overcome this analysis. This study was a static analysis of unpacked malware or malware that used the same packer. As a result, if a malware family uses different packers for each variant then there will be no similarity found.

Kancherla and Mukkamala (2013) proposed a visualisation method that is more robust to code obfuscation and where it does not require the malware to be unpacked or decrypted before analysis. The authors took a similar approach to Nataraj by converting an executable to a grayscale image. While Nataraj solely worked with GIST based features and used K-Nearest Neighbour algorithim for feature analysis, the authors proposed extracting intensity, wavelet and gabor based features from images and used Support Vector Machine algorithm for feature analysis. The authors also reviewed various types of malware samples including malware variants that were packed with different packers. The authors were able to show that using intensity and wavelet feature detection showed a performance increase in detection of malware variants.

Han et al. (2015) proposed a method to visualise malware variants in entropy graphs. The proposal was to overcome time-consuming similarity calculations and to enable the detection of similarities between malware variants in an alternative way. The author took a similar approach to Nataraj where the binary files were converted to grayscale images. The grayscale images were then used to calculate entropy values of the binary files which are used to generate entropy graphs (Figure 2.5). Once the entropy graphs are generated for all samples a similarity measurement is calculated.

The similarity calculation was accurate for unpacked malware and malware variants could be detected. However, the weakness of this method was that packed binaries have high entropy values. This meant that classification and therefore identification of benign and malware variants are extremely difficult. When using entropy graphs for visualisation and a similarity calculation, all packed binary files will give similar results.

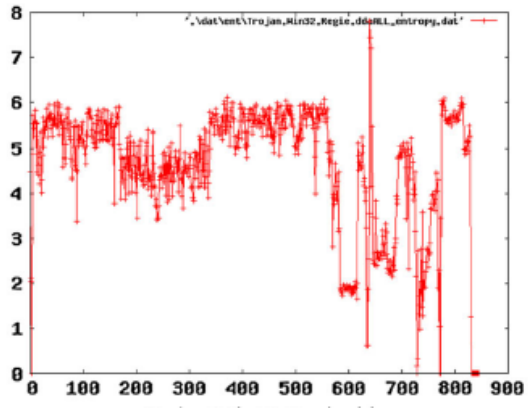


Figure 2.5: Entropy Graph of Trojan-Dropper.Win32.Del

Ren and Chen (2017) proposed a visualisation analysis method for investigating malware similarity. The authors further developed the techniques used by Nataraj in order to improve malware classification. According to the authors the malware visualisation classification used by Nataraj could be limited if each malware variant change uses function reordering or control flow modification. This is where the order of the sections of code and therefore the features extracted can have limited accuracy. The authors proposed a method that allowed them to overcome these limitations.

The authors begin by converting PE files into local entropy images by computing entropy values of blocks. In Figure 2.6 and Figure 2.7, examples of the Trojan.Win32.Regrun variants and the visual similarities can be seen after creation of the entropy images. The entropy values are then normalized, and the entropy images are converted into entropy pixel images (Figure 2.8) for easy visual similarity. The entropy pixel images are stored in a database where the similarities of the images and therefore malware variants are measured using the Jaccard index. Finally, a K-Nearest Neighbour pattern matching classifier is used to sort the samples according to the similarity value and the samples are assigned to the corresponding malware families based on the Jaccard similarity results.

This study was completed using static analysis. The study was designed to overcome a minimal amount of code obfuscation. Something as simple as using a different decryptor for

each variant would easily minimise the impact of the method.

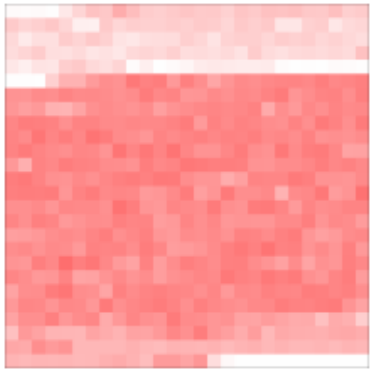


Figure 2.6: Regrun.rk Entropy Image

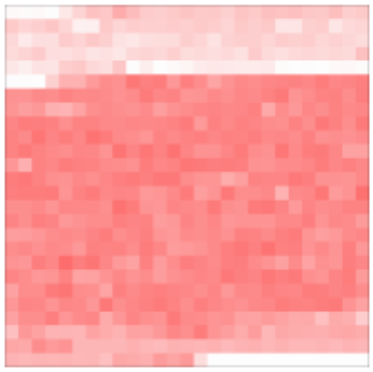


Figure 2.7: Regrun.yi Entropy Image

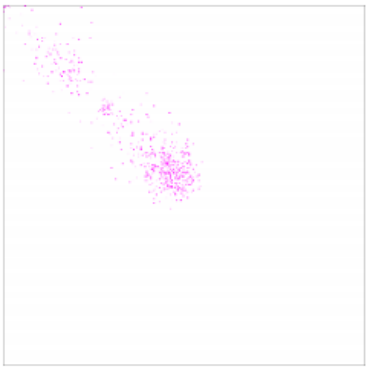


Figure 2.8: Entropy Pixel Image of Regrun family

Makandar and Patrot (2017) compared malware class feature recognition techniques. The authors used a pre-existing malware dataset that had already been converted to grayscale images. The authors compared Nataraj's feature extraction technique by comparing GIST, gabor, wavelet and discrete wavelet transform image feature detection to see if more accurate detection could be obtained. The authors then used K-Nearest Neighbour and Support Vector Machine for image classification. The authors showed that when using discrete wavelet transform to extract features from an image, it reduces the number of feature vectors for classification and therefore reduces complexity.

The authors presented improved accuracy when detecting malware variants, when using extracted discrete wavelet transform features and the support vector machine learning model. The weakness was that the dataset was static in that no new malware was used in the study so modern obfuscation techniques were not taken into account in the final results.

Raff et al. (2017) expanded beyond image recognition and into detecting spatial correlation behaviours by using a neural network. While the previous studies converted raw binaries into images, the author used a similar feature extraction procedure to Nataraj, but simply used the raw byte sequence of a binary as input into a neural network. The authors expanded the application of neural networks by training the network to be able to differentiate between

windows executables that are harmless and malicious. The authors work extended beyond image recognition and into detecting spatial correlation behaviours.

The advantage with this approach is that they do not concentrate on the executable header to gather important information. The disadvantage is that it uses batch-normalisation which means that the neural network is trained only for what has been put in previously so new malware variants will still be undetectable.

Hashemi and Hamzeh (2018) proposed another method for detecting malware using computer vision. The authors found that there was different behaviour and functionality between malware and benign files and that the difference was potentially detectable in micro patterns. The authors concentrated on differentiating malware and benign files by analysing these micro patterns in executable files using a method of textural image classification. The authors proposed a machine learning based method whereby visual features from executables are extracted using a modified local binary pattern matching texture operator. (Figure 2.9 & 2.10) The features detected are then used to train machine learning tools to be able to classify malware and benign executable files. The advantage of this approach is that the malware does not need to be executed or disassembled for analysis in order for this approach to work. The downfall of this approach is that the visual representation can be altered significantly by something as simple as encrypting the files.

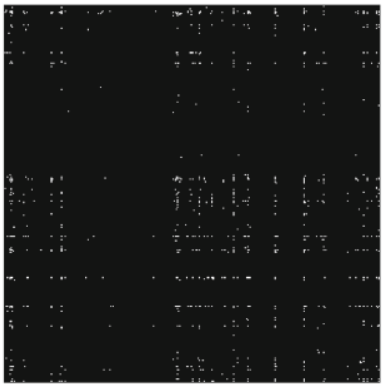


Figure 2.9: Win32.FlyAgent.jt

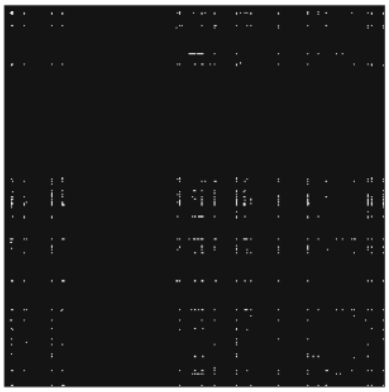


Figure 2.10: Calc.exe

Fu et al. (2018) further expanded on the research done by Nataraj. While Nataraj visualised malware in greyscale images and extracted image texture features of the malware families, the authors used RGB-coloured images and found some visualisation results that were different to Nataraj's findings. The author used a different method when visualising the malware by splitting the PE file up into sections based on the PE format. These sections were then visualised and then combined to create a RGB coloured image. Using this method, the author showed that the same family of malware may show several different images when the PE file is broke down, visualised and then combined (Figure 2.11). What they did find was that the data and code sections of the PE file malware families were identifiable based on RGB colour and texture. What they were also able to show was that some samples from different families can have the same texture as other malware families and that they could only be distinguished by colour.

The weakness with this method was that it proved to be a poor solution against packed or encrypted malware. The authors had to either unpack or decrypt the malware before completing analysis in order to be able to combine the separated sections later in the process. This meant quite a lot of manual effort to prepare the samples for this study.

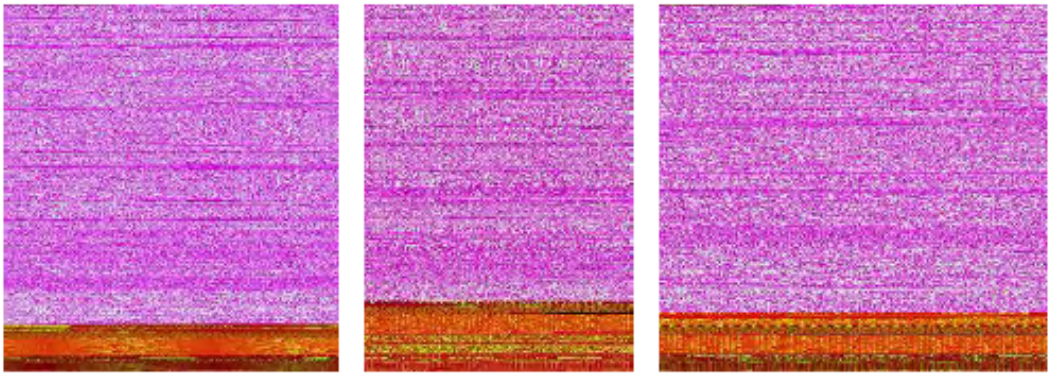


Figure 2.11: RGB Malware Visualisation of Backdoor.Win32.Hupogon family

As distributed denial of service (DDOS) attacks have become increasingly common against IoT devices and environment, Su et al. (2018) completed a study where the authors concentrated on just IoT malware. IoT devices often lack basic security monitoring

and security, as exposed by the recent Mirai and Brickerbot IoT botnets attacks. As part of a proposed solution the authors took a similar approach to Nataraj by converting malware DDOS binaries to grey scale images.

Using these grey scale images, they were able to classify the malware families by feeding into machine learning classifiers which were then able to detect the various malware families to a high degree of success. The authors found that using a neural network for malware detection, they could improve performance of image recognition and therefore speed up malware classification.

The advantage of using a neural network was the automatic feature extraction learning capability. This is where the neural network can automatically learn deep non-linear features of a malware family that is not easily picked up by a malware analyst. The issue with this method was that the training time needed for the neural network was quite long and the sample tested was quite limited. This method will not work where polymorphic or metamorphic obfuscation techniques are used as the malware sample changes itself with every variant.

2.4.2 Reverse Engineering

Quist and Liebrock (2009) proposed a visualisation method to aid reverse engineering of malware. Unlike the other methods and techniques discussed this was not used to classify data or discover interesting areas of a malware executable in order to determine their overall functionality. Instead it is aimed at assisting malware analysts to reduce the time needed to extract key features of an executable during reverse engineering. The technique developed was called Visualization of Executables for Reversing and Analysis (VERA). It was designed to differentiate code section entropy by covertly monitoring and tracking memory reads and writes of malware code sections. The resulting output data gathered was then used to build 2D and 3D visualisation of the memory read and writes by malware (Figure 2.12).

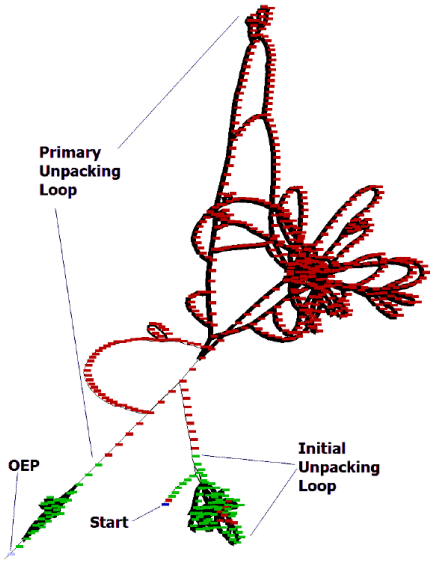


Figure 2.12: Close-up of the MEW unpacking loop

In order for this method to work the malware had to be de-obfuscated before reverse engineering or the code sections would not all be detected successfully. This required a lot of manual effort in the study as they were required to remove encryption and environmental awareness, as the code itself was the primary tool used to understanding the program execution.

Donahue et al. (2013) overall goal was to help in-experienced security professionals to easily extract underlying detail of a PE file and to be able to identify a packed portion of a PE file through visualisation. The authors proposed two techniques, an interactive hex editor and a visualisation technique in order to aid reverse engineering. The authors used the Markov Byte plot technique to identify a packed portion of a PE file (Figure 2.13). While the research seemed to be limited to just the UPX packer they were able to demonstrate the difference between a piece of malware where a portion of the code was packed and not packed and how this potential solution could aid security professionals in identifying malware.

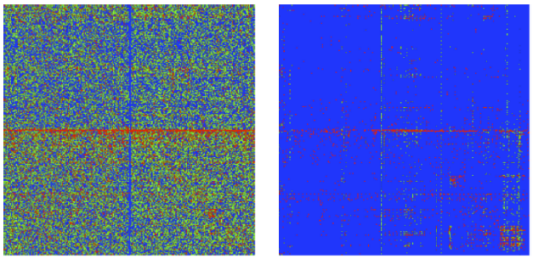


Figure 2.13: Markov Byteplot: Packed and Unpacked versions of Beagle malware

Han et al. (2014) proposed a visualisation method to convert binary files into images and also a similarity calculation method on the images. The author uses a combination of disassembly and dynamic execution analysis to extract opcode sequences from malware. Only well-known blocks of code related to suspicious behaviour that include call instructions like API's or library functions are used. The opcode sequences are extracted from these selected blocks and RGB coloured pixels are generated into an image matrix for that sample (Figure 2.14). Similarity calculations using the vectorized values of RGB pixels are then performed and image matrices on the malware family are generated.

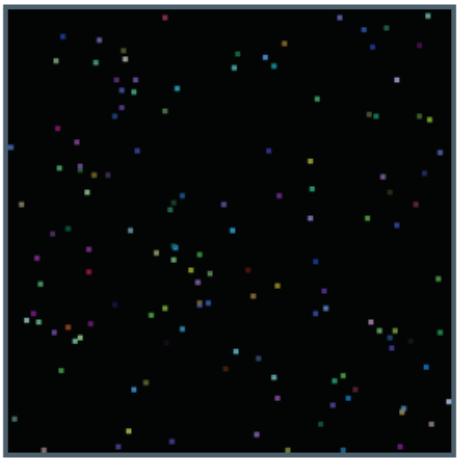


Figure 2.14: Image Matrix of Win32.IstBar family

By not using all code blocks it enables the author to decrease the number of code blocks that need to be analysed and therefore save analysis time. However due to the fact that both static and dynamic analysis of a malware sample has to be performed this means that the

initial time to gather opcode sequences can be significant especially for obfuscated code. In some cases due to environment detecting obfuscation or encrypted code sections neither static or dynamic disassembly is feasible. Also by preselecting blocks the author may miss some key code features for a particular malware family and therefore may not be able to detect the different variants.

Manavi and Hamzeh (2017) proposed a new method in the area of opcode visualisation. The authors begin by disassembling the executable to extract opcodes for both malware and benign files. A matrix is created for each file based on the frequency of binary combinations of opcodes. The matrices are then normalised and converted to grayscale images by using the same technique as Nataraj. The features of these images are extracted using GIST. These extracted features are then used to build a model using machine learning methods Support Vector Machine, K-Nearest Neighbour and ensemble. The data is then tested using this defined model and it is classified as either malware or benign.

While the opcodes analysis of malware and benign files did create suitable models that could detect some new malware samples, the weakness in this method was that the authors had to disassemble and reverse engineer the malware before analysis which required a lot of manual effort.

Venkatraman and Alazab (2017) designed a visualization classification technique of malware using similarity matrices. It's main aim was to provide faster detection and classification of malware. The authors adopted a malware feature-based technique in order to understand and compare operation code features extracted from related malware binaries. The authors used a combination of automated reverse engineering to disassemble malware samples and then used similarity mining to compute similarity between functions extracted from the samples.

A similarity matrix (Figure 2.15) is generated after each comparison completes and compared with a baseline of benign operation codes. The method is effective in identifying

malware visually due to difference in the behaviour patterns when comparing malware operation codes and benign operation codes. The method fell down in that some comparisons produced patterns that required manual intervention in order to investigate and analyse further. The method is not fully scalable because not all malware can be reverse engineered so therefore the operation codes could not be detected.

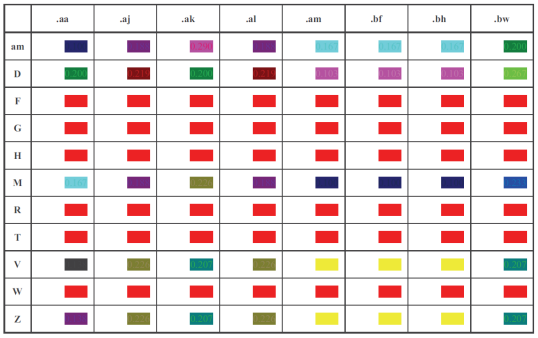


Figure 2.15: Similarity matrix: Win32.Dadobra vs Win32.Delf malware families

2.4.3 Behaviour Analysis

Lee et al. (2011) completed a study on visualisation of malicious code whereby they could identify new and existing variants of malware based on code pattern matching. The authors completed the study by extracting properties of malicious code from given samples already identified by anti-virus companies. The authors assigned binary values to certain code behaviour e.g. gathering personal information or sending an email. These binary values were plotted to visualise the code pattern in scatter plots.

By using scatter plots the authors could demonstrate the similarities between malware variants using these binary values and as a result that they could detect new malware variants based on the existing malware pattern matching. The weakness with this visualisation method is that scatter plots are not the easiest visualisation method to accurately read the results as the differences in marginal values are hard to decipher.

Shaid and Maarof (2014) proposed a new malware visualisation technique based on malware behaviour. The malware behaviour can only be determined when the sample is executed. To determine the behaviour all user-mode API calls are captured. Each captured API call is classified on a colour scale based on its level of maliciousness. Red being malicious and blue being benign. These captured API calls are then mapped to the colour scale which is then used to generate an image. By reviewing the generated colour image, a malware analyst can easily identify malicious or benign samples. (Figure 2.16)

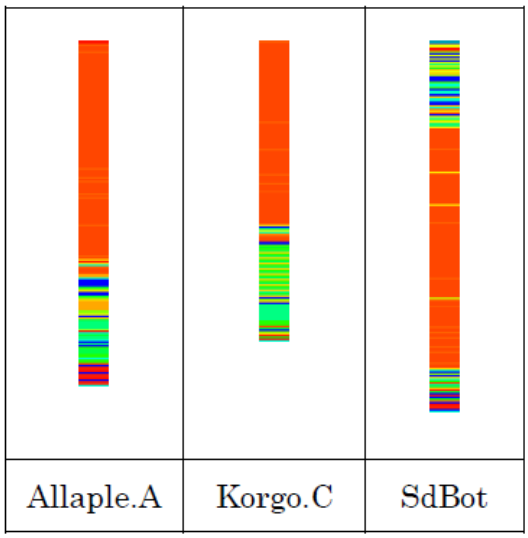


Figure 2.16: Malware behaviour image sample for 3 different types of malware

This technique can also be used to identify malware variants by highlighting malicious behaviour in malware samples. Using this technique, it is possible to detect lightly obfuscated malware, but the authors had skewed results when malware used the UPX packer or when the sample were heavily obfuscated.

2.5 Memory Analysis

2.5.1 Trigger based memory analysis

Teller and Hayon (2014) reviewed some of the limitations of the then current memory analysis techniques. There are numerous limitations that can hinder memory analysis. Memory analysis was a very manual effort, so it was quite time consuming. There are also anti-forensic methods and tools existing that can prevent memory dump acquisition or can be used to input fake objects into memory. One of the biggest limitations is that a piece of malware may have already executed and cleaned up memory before you take the snapshot, so you can miss any malware analysis in the memory dump. Finally interpreting the results of the output requires a deep knowledge of the underlying operating system functions.

The authors devised a memory analysis method to enhance automated memory analysis in order to boost malware detection rates. To overcome various obfuscation methods the authors developed trigger-based methods to take memory dumps. They developed API based, performance based and instrumentation-based triggers whereby they could wait for malicious behaviours to occur based on API call sequences, like process injection or performance thresholds being met. To begin with they created a clean non-malware infected memory dump baseline in order to compare the malware infected memory dumps. Their most fruitful findings came from API based triggers and they could then compare their results against a baseline in order to detect the malware.

While this approach shows some potential regarding detecting individual pieces of malware, there are still limitations as the malware can encrypt or pack sections of code which essentially leave the malware dump for encrypted static analysis. The trigger-based approach may be one method that will be worth further research for this study.

In order to overcome encryption, obfuscation and packed malware, Rathnayaka and Jamdagni (2017) proposed a modification to the methods used by Teller and Hayon. They integrated static analysis with memory forensic analysis in order to reveal malware hidden

behaviours. The static analysis included reverse engineering the samples to remove obfuscation for a smooth execution of the malware for memory analysis. This method does enable analysis of most malware by removing obfuscation, encryption and packing. However not every piece of malware can be successfully reverse engineered. This method requires extremely time consuming manual effort as it is not possible to fully automate the reverse engineering of malware.

2.5.2 Feature Memory Extraction

Aghaeikheirabady et al. (2014) study used static memory analysis and focused on detecting malware by comparing extracted features of malware from memory against already known data structure features in memory. The technique used extracted malware artefacts relating to virtual address descriptors, file object structures, registry changes and API function calls. The extracted features were classified and then compared against benign sample features and known malware features using machine learning algorithms some of which included Sequential Minimal Optimization, Random Forests and Decision Trees.

Virtual Address Descriptors (VAD) are vulnerable to direct kernel object manipulation (DKOM) attacks. As the authors used VAD as a feature in detecting memory structure it allows for manipulation of the final results and potential inaccuracy of feature detection.

Mosli et al. (2016) developed on Aghaeikheirabady et al. study by just using registry activity, API function calls and imported libraries as feature detection. Their aim was also to detect malware on a live system as it executed, and not using static memory analysis. They started by training models with extracted features for benign and malware samples. They classified the data and used a number of machine learning algorithms including Support Vector machine, K-Nearest Neighbour and Decision Tree. The authors obtained a high success rate of detection of between 93% and 96% detection depending on the algorithm used.

Both Mosli and Aghaeikheirabady studies were completed on very small data sets. To

improve the sample result accuracy for this technique a much larger data set of both benign and malware samples is needed. The other weakness with this method is if the malware authors use address obfuscation and in particular positional variation. This is where the code is not arranged in a standard data structure. The header of the binary remains the same but the rest of the file virtual address of the code and data can be rearranged throughout the binary. The randomisation of the code can be continuosly changed during execution time. This technique has traditionally been used legitimately to protect against memory exploits.

2.5.3 Fileless Malware

In recent years malware authors have been using the .NET frameworks powerful ability to interface with Windows operating system and the Microsoft product line. PowerShell modules are often whitelisted, so PowerShell scripts can be loaded into memory without ever leaving any file-based evidence. Pontiroli and Martinez (2015) completed a study on PowerShells ability to interface with the Microsoft product line. The authors found that an attacker can have the ability to attack an active directory server in order to attack other servers on the same network without leaving any file-based evidence behind.

Patten (2017) expanded on Ponitioli and Martinez research. The author concluded that due to the .Net Framework's ability to interface with the Windows operating system and Microsoft product line, that it's ability to be able to capture and detect fileless or memory only malware was essential.

The author discusses how some modern malware is designed to work solely in memory and how attacks exploit the .Net Framework. Fileless malware exclusively lives in RAM and it does not write any part of its activity to the file system. The author found that when using visualisation methods during malware static or dynamic analysis, it is not always possible to extract characteristic or signature features of a piece of malware if that malware is heavily obfuscated. However, by using memory analysis as the malware is executing, malware analysts can then use visualisation techniques to provide visual representations. The visualisations can then be used to aid malware analysts when investigating and comparing malware

samples.

2.6 Conclusion

There have been numerous attempts at visualising malware and using pattern matching algorithms in order to detect malware. Nataraj's research has been used as a baseline for multiple research projects. Visualisation of malware has been successfully concluded across static, dynamic, behaviour and reverse engineering and has proven that it has the potential to aid malware analysis.

There have been a number of ways that malware has been visualised including scanning every raw byte and converting it into an image pixel, using grouped API calls in behaviour analysis, calculating entropy values, analysing and calculating micro patterns or extracting opcode sequences. The current research has also shown various ways that features can be extracted from these images including by texture-based features or entropy values, and how features can be calculated based on GIST, Gabor, Intensity and Wavelet filters. Feature analysis models using the pattern matching algorithms K-Nearest Neighbour and Support Vector Machine have shown how machine learning algorithms can be used to detect malware and malware variants. However due to varying obfuscation techniques each study seems to have a weakness where certain obfuscated samples could not be detected. Nataraj's research which was on static samples only works on detecting un-obfuscated or similarly packed malware. Both Ren and Chen's or Hashemi and Hamzeh studies were limited as the research could be overcome by using simple encryption. The Quist and Liebrock or Venkatraman and Alazab research had to de-obfuscate or reverse engineer the samples before analysis.

Nataraj's research on static samples whereby the authors convert malware to grayscale images, extracts GIST features and used K-Nearest Neighbour pattern matching algorithm to detect malware samples showed promise of detecting un-obfuscated or similarly packed malware. However this method cannot detect when a different packer is used or metamorphic malware variants.

By combing Tomer and Adi's memory analysis research with Nataraj's visualisation study, it has the potential to improve the detectability of obfuscated malware variants by using visualisation of malware in memory analysis.

Chapter 3

Design & Methodology

3.1 Introduction

The approach taken for this research study is quantitative by nature. The basis for this study comes from Nataraj et al. (2011) static visualisation of malware and Teller and Hayon (2014) trigger-based memory analysis. Using a static analysis technique Nataraj managed to show that unpacked malware could be identified using visualisation techniques along with pattern matching algorithms. Tomer and Adi showed that by detecting API triggers in memory, that they could detect malware. By combining visualisation and memory analysis this study aims to show that visualising malware captured in memory can detect obfuscated malware variants. While Nataraj visualised malware statically, extracted GIST features and using K-Nearest Neighbour machine algorithm to identify patterns in malware variants, this study will visualise malware in memory, extract GIST features and use K-Nearest Neighbour and Support Vector Machine to identify patterns in malware variants. Experiment 1 will compare obfuscated visual static examples with samples executed in memory.

Figure 3.1 is an overview of the entire procedure. Malware families are identified based on the most recent and popular variations and obfuscation. Each sample is executed in the Cuckoo Malware Analysis Sandbox (2018). Static, dynamic and behavioural analysis is automatically performed using open source tools. Process memory dumps and a full operating system memory dump taken. The process is automatically extracted from the process

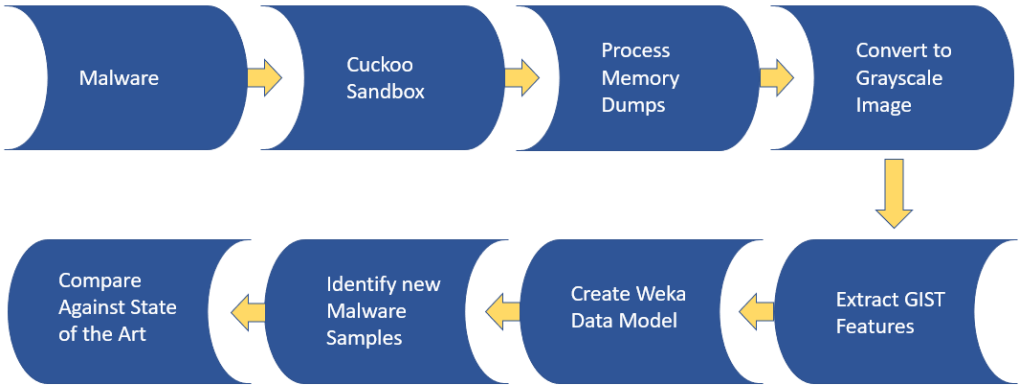


Figure 3.1: Procedure Overview

memory dump and can now be converted to a grayscale image. A Global GIST descriptor is calculated for each image and these values are used to create a data model using *WEKA* (2018) data mining tool. Test and training sets are created in order to determine how accurate this method is in detecting new malware variants. Finally, the last step is to compare the accuracy of this method with the current state of the art systems in place.

3.2 Restrictions & Limitations

The data set used is small considering the number of new malware variants detected every day. The malware data samples are restricted in that for some of the family's different variants are from the same month and are not spread out over time. As a result, these variants use the same obfuscation techniques to evade anti-virus solutions and once visualised from memory are quite easy to detect.

A number of malware samples detected the analysis environment. Some samples used specific environmental awareness where it specifically looked for the cuckoo sandbox python agent and killed the process. There were a small number of older malware samples that upon investigation depended on older visual c++ libraries and as a result did not execute. Due to the number of samples executed across two different cuckoo sandbox setups', it was

determined not to investigate every sample that could not be executed due to the time needed to analyse the sample using static, dynamic and reverse engineering tools as well as complete the post processing necessary for this project.

3.3 Identify Data Type

The Windows portable executable (PE) format is a well-documented data structure that includes all the information needed for the Windows loader program to manage an executable. In previous studies having this data structure has allowed researchers to be able to classify sections of a PE file visually. Conti was able to visualise binaries because they could identfiy sections of a file. Figure 3.2 give a breakdown of the section of the Trojan Dontovo.A taken from Nataraj's study. Nataraj also took this approach and as a result, for this study windows 32-bit executables will be used as they have a data structure that can be visualised and compared against Nataraj's approach easily.

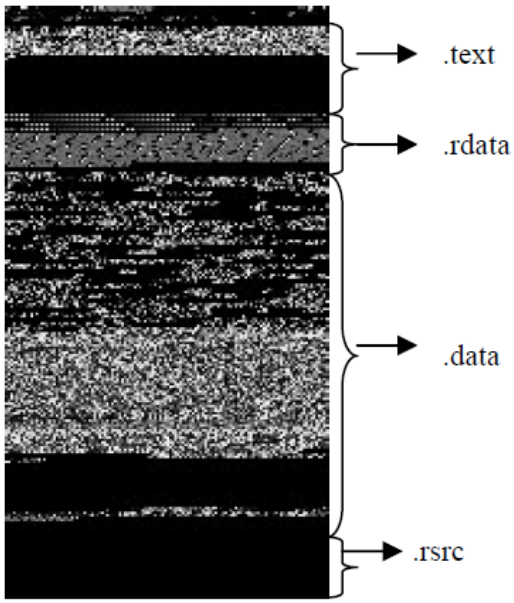


Figure 3.2: PE Sections of a File

3.3.1 Data Source

VirusTotal (2018) combines multiple vendor anti-virus solutions and website, domain and URL scanners to analyse files and URL's for malicious content. The detailed results are shared publicly and with anti-virus solution partners to improve the detection ability of their own systems. Millions of samples are uploaded and analysed every week by partners, malware analysts and the public. As a result, VirusTotal have a huge dataset of known malware.

The sample data used in this study was obtained from by gaining temporary academic access to a limited amount of malware samples from virustotal.com. The samples made available by VirusTotal were uploaded between October 2017 and December 2017. The CoinMiners were made available in a separate sample set from June 2018. Note that samples can be uploaded multiple times over a number of years as malware can exist in the wild for years even though it has been detected by anti-virus solutions. Therefore quite a number of samples were first detected many years prior and they are still being uploaded for analysis today as they still pose a threat. The VirusTotal sample data set included numerous families of malware and enough data to test the proof of concept in this study.

For the Trigger based analysis a small data set was assembled from the github "The Zoo repository assembled by ytisf (2018) and from the malware listings from malwaredb.com, Malekal (2017). Both repositories are renowned for making available malware samples for budding malware analysists. The reason for the two data sets is because of the late acquiring of data samples from VirusTotal and time necessary to post process the trigger-based analysis samples for a larger data set.

3.3.2 Identify Malware Families

The whole approach of this study is to detect obfuscated malware. So, the malware families chosen must be obfuscated. Also, a check to see if this approach can be used for multiple types of malware so some of the most popular malware variants over the last few years are being used in this study. According to Symantec Internet Security Threat report completed

Cleary et al. (2018), Locky was still one of the big ransomware families in 2017 and was shown in a study to have evolving obfuscation over the lifetime of its variants. The Mirai botnet was one of the biggest DDOS botnet attacks in 2016 and was still impacting IoT devices, routers and personal computers in 2017 due to its ability to infect multiple CPU architectures. Symantec reported that the most dominant banking trojan was Ramnit. Ramnit has been around in different variations since 2011 and has been used as a network worm to deliver other malicious packages. By the end of 2017 it was the number one banking trojan and could steal other information and setup backdoors into a compromised computer. There has also been a large growth in coin mining as Symantec reported an 80-fold increase in 2017. CoinMiners steal computational power and electricity to mine a currency. CoinMiners looking to avoid detection are known to tamper down their activities by not using a high percentage of CPU resources and therefore avoid detection. They are known to use various persistence mechanisims like registry manipulation so that if it is detected, it can re-install itself and re-infect a system. This study will also look at the Emotet and Hupigon malware variants. Hupigon shows a different kind of threat where it is known to use process hollowing in order to inject code into legitimate Windows processes. Emotet is a re-emerging banking trojan Both families have shown developing obfuscation over the years and still provide a significant threat to users.

Mirari

A study completed by Antonakakis et al. (2017) discusses where the Mirai botnet first came to prominence in September 2016 with DDoS attacks on French Internet service provider OVH, Krebs on Security blog and Dyn DNS service. The malware was designed to work across numerous CPU architecture like x86, ARM and MIPS. This enabled Mirai to infect a multitude of devices including IoT devices like camera's, infrastructure devices like routers and home computers. It was a two-stage attack where the 1st stage would detect the underlying architecture and then download a binary specific to that architecture. Over time various additional functionality and obfuscation techniques were added to variants of Mirai. This

included appending new passwords to the list, deleting the first stage binary, deleting itself once malware is running, obfuscating its process name and ID by altering it to a random value and killing competing malware. As a result, standard signature-based detection was not a reliable method for detecting the Mirai botnet variants. A further study completed by Said et al. (2017) showed that a small sample of Mirai variants eventually included dynamic string obfuscation where the strings are only decrypted at runtime which enabled it to circumvent detection by YARA rules at the time.

Locky

Locky ransomware was originally uncovered in 2016 where the earliest variants were delivered via email and a Microsoft word macro. Originally the macro was designed to download a encryption trojan which encrypted only files with certain extensions. As discussed in section 2.3, Cabaj et al, study on the Locky ransomware details the various new functionality and characteristics introduced with new variants. The obfuscation techniques included environment detection like checking for a debugger, checking for cursor movements and checking the parent process name. This counter analysis efforts meant that dynamic malware analysis or reverse engineering was made more difficult and with newer variants, new analysis environments and techniques were needed. McAfee (2017) threat report analysed Locky further. They found that even when the sample was executed successfully in a analysis environment, there was encryption, Unicode encoding, zero padding, XOR encryption, packing and obfuscated variable names used in different variants in help counter analysis efforts. Due to the various obfuscation techniques used Locky is a good example to test in this study.

Ramnit

Ramnit has been around since 2010 with expanding functionality and obfuscation. It has been used as a botnet and more recently as a banking trojan. In the Cleary et al. (2018) report, it as the top banking trojan in 2017. A study completed by Symantec (2015) showed its development of obfuscation techniques over time. it developed infection techniques like

infecting the master boot record, installing a service and modifying the registry to maintain persistence. It later contained a spy module to monitor websites for logon information. It had the ability to install a virtual network module and anonymous ftp server to allow remote connection. It has numerous obfuscation techniques including packing, debugging detection, emulation detection and host-based encryption.

Coin Miners

Both McAfee McAfee (2018) report and the Symantec threat reports by Cleary et al. (2018) shows a explosion in the detection of coin mining attacks. Due to the rise in the value of bitcoin, cybercriminals began to concentrate on crypto jacking or mining for crypto currency. Compared to other types of malware crypto jacking is a straight forward infection in that as all it has to do is infect a system in order to start to earn money. The crypto miners will use an algorithm to mine the currency and a separate protocol to provide privacy and anonymity. They will often change the power schemes on the infected machine in order to have maximum impact. The authors will often connect through private pools or proxies in order to hide wallet addresses. They maintain persistence by creating a task on start-up, adding the program to the Startup folder and creating registry entries. The sample taken for this study is a random sample taken from VirusTotal.

Hupigon

Hupigon has been around since 2007 in numerous forms. According to Comodo (2017) threat report Hupigon was still in the top 3 detected malware families in China. It is regarded as a backdoor program whereby using rootkit functionality it bypasses normal security mechanisms in order to control a program or computer. It works slightly differently to the other malware families in this study in that through numerous variants, it uses different windows process to infect a machine. It obfuscates itself by injecting legitimate windows process and often creates multiple process identities. It is sometime used alongside other malware where

it is dropped by the other malware onto the system to allow for further infection of the system. Over time it has developed multiple abilities including allowing access to webcams, logs keystrokes, steals passwords, send victim messages or to be used as part of a botnet.

Emotet

Symantec reported in their 2018 threat report that the banking trojan Emotet has remerged with a high number of new detections in the last quarter of 2017. Emotet was originally identified in 2014 and was thought to now be of minimal threat. New variants have since been identified from a large email spam campaign. The phishing email would download a word document which prompts the user to enable macros which then downloads the payload. The malware was originally an information stealer but now has added functionality which enables it to add infected devices to a botnet. It uses new obfuscation technique like dead code insertion, variable renaming and has advanced environmental awareness. The interesting thing with this family of malware is its re-emergence as a significant player in the banking trojan variants

3.3.3 Extracting Data

The VirusTotal supplied samples sets downloaded in zip files ranging from 2 GB to 50 GB. The malware samples are identified using the SHA 256 hash of the binary. An equivalent named json file includes a copy of the VirusTotal report including the hash values of the file and a list of anti-virus companies that detected the sample along with the identification name they gave to it. In order to identify malware family samples, the json files are searched for strings of the name of malware samples. To start with the Mirai family variants were searched for. 22 samples were discovered using a simple find command and extracted to a text file.

find . -type f -name '*.json' -exec grep –files-with-match -i 'Mirai' '{}' \; >Mirai.txt

The content of the text file was then copied from the VirusTotal sample set to a separate family folder.

xargs -a Mirai.txt cp -t /home/cuckoo/Downloads/Mirai/

The file was edited and the .json file extension was removed from all entries. The file was saved as Mira_Files.txt. Mira_Files.txt was copied back into the VirusTotal sample set directory and using xargs the malware samples are then copied over to the family folder.

xargs -a Mirai_Files.txt cp -t /home/cuckoo/Downloads/Mirai/

After identifying and extracting the malware a md5 hash check was executed. This was to chck for user error and to find any duplicated files to ensure that only unique malware samples are used in this study.

3.4 Lab Envoirnment

To run this study a malware analysis sandbox setup is needed that has the ability to take process memory dumps as well as full operating system memory dumps. There are numerous online sandbox environments available including ThreatExpert, CW Sandbox, Comodo Valkyrie, Joe Sandbox and Hybrid-Analysis. These solutions come with various functionality including Yara scanning, IDS detection, and VirusTotal scans integrated. Some of additional functionality is dependent on a billing model so this is charged for. One of the open source alternatives available currently is the Cuckoo Sandbox.

Tomer and Adi's study was completed on a customised version of the cuckoo sandbox. For experiment 3 the same process that Tomer and Adi developed will be replicated. The visualisation process of Conti and feature extraction and malware pattern analysis used by Nataraj will be replicated. As the Cuckoo Sandbox provides all the functionality needed for this study and as both process memory and trigger memory experiments are to be executed it is proposed to use the cuckoo sandbox for the automated malware analysis environment.

As malware can perform many malicious activities the analysis environment was designed to provide little opportunity to allow infection from the malware samples. I choose to use nested virtual machine environment. The test lab ran on a Windows 10 x86_64 laptop with 8 x CPU and 16 GB of RAM. For the analysis environment host machine, a ubuntu 16.04 virtual machine on VMware using 4 CPU's and 8GB of RAM was chosen. For the guest analysis environment, from within the ubuntu host virtual machine a Windows 7 x86_32-bit virtual machine on VirtualBox using 1 CPU and 1GB of RAM was chosen. Using 1 CPU for the guest analysis machine provided sufficient performance and only 1GB of RAM gave a performance improvement on taking memory snapshots as well as needing less space to store the memory dumps.

The cuckoo host analysis virtual machine had access to the local network via NAT. This was necessary for VirusTotal and anti-virus signature checking when running the cuckoo sandbox. Some of the malware samples analysed have the ability to spread automatically or conduct distributed denial of service attacks so the test lab must not enable an attack on others unknowingly. An isolated virtual network was chosen for the guest analysis machine so as not to allow the malware to connect to the internet or to connect to a real controlling host. The risk with this strategy is that the malware may not execute fully each time. Another concern is if a real connection was allowed an attacker may decide to change his behaviour if they see connections coming from a computer that they did not hack. As a result, the external IP address may become a target for future attacks.

The host ubuntu virtual machine was hardened with steps like enabling UFW firewall, disabling ssh, ftp and telnet kernel modules, disable port forwarding, strengthening user passwords and setting up a test cuckoo user with limited rights. The isolated network interface in Virtualbox was given a static IP address of 192.168.56.1. The cuckoo server could connect to the cuckoo agent via this interface over port 8000.

The guest machine had all windows security features turned off. The firewall was disabled, windows defender was disabled, automatic updates turned off, security centre is turned off and universal access control was disabled. There were a number of standard programs

installed including Adobe Reader, Java, .Net Framework, MS visual C++, Chrome, Firefox as well as a few other 3rd party apps like VLC player, Winrar and Notepad++. All new software was executed once and relevant dummy files were copied over to the Pictures, Documents, Download, Music and Videos folders. Internet browser history was also imported into Chrome, Firefox and Internet Explorer.

Python 2.7 and the pillow library was installed to enable the cuckoo agent to run and to also allow screenshots to be taken as any malware sample is executed. The cuckoo sandbox agent was copied into the Startup directory and was executed. A fixed static IP address of 192.168.56.10 is used to interface with the cuckoo sandbox server. (Figure 3.3)

A backup of the virtual machine is created and exported in .ova format. The export is stored on an external hard drive in case the virtual machine became corrupt and any snapshot cannot be restored. Finally, a snapshot of the Windows 7 analysis virtual machine was taken while it is running. This was used by the cuckoo sandbox as a restore point that it loads when a new sample is submitted. It then injects the malware into a running state.

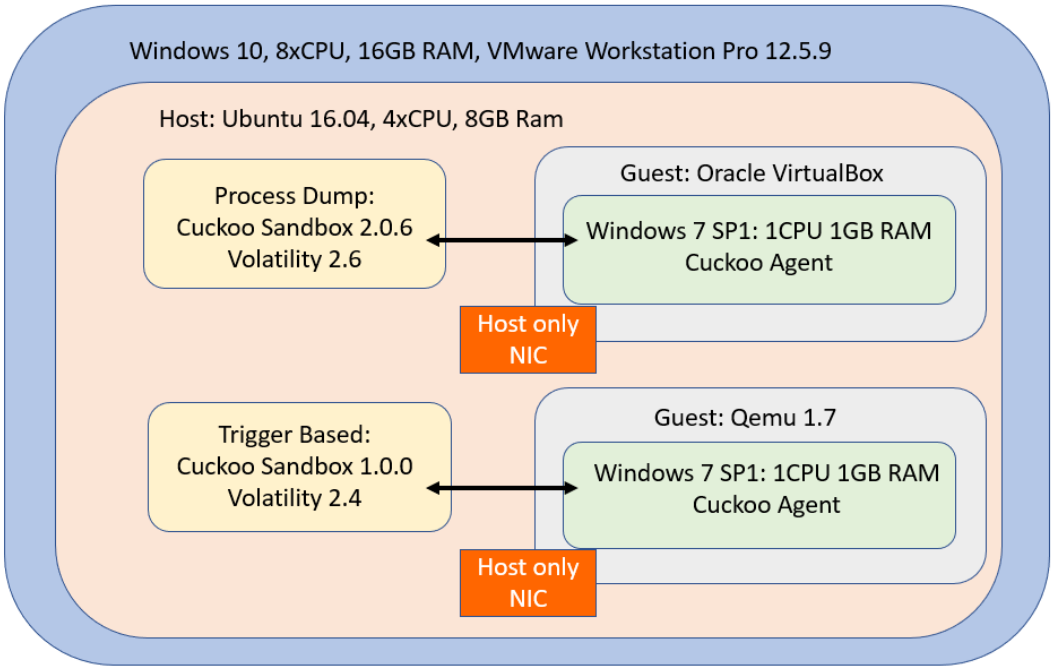


Figure 3.3: Lab Setup

3.4.1 Cuckoo Sandbox

The Cuckoo Sandbox is an open source automated malware analysis system based on python 2.7. It is used to execute and analyse malware and collect results that outline the behaviour of the file while it executes in an isolated operating system. The Cuckoo Sandbox has built in functionality to trace API calls performed by all processes that are spawned by the malware. It traces all files created, deleted or downloaded and registry key read, write and deletion's. It uses a number of open source programs to aid further analysis. The cuckoo sandbox has built in functionality to take full memory dumps of the virtual machine while it uses Volatility (2018) as a plugin to take process memory dumps of malware process's during execution. It uses tcpdump for network traffic analysis and python Pillow library module to take screenshots. Further functionality includes searching VirusTotal.com for antivirus signatures, Yara signature scanning and Snort IDS rule scanning. It has the ability to analyse multiple files types including windows executables, dll files and pdf documents. Virustotal.com and other analysis environments use the cuckoo sandbox as part of their malicious software detection infrastructure.

Cuckoo was configured with the web interface to allow for easily analysis of results and submission of samples. The Cuckoo host management server was configured to communicate with the guest machine over the VirtualBox host only virtual network interface. To begin the analysis Cuckoo opens a snapshot of guest virtual machine containing the preconfigured Windows 7 operating system. This provided a clean environment to run a sample every time. A cuckoo agent program runs in the background on this guest operating system. The Cuckoo agent also injected the malware sample into the guest operating system which allowed for communication with the cuckoo server, analysis of the samples and the gathering of results in real time.

Once the analysis was complete, a report containing static analysis like strings and VirusTotal signature search can be reviewed. Behaviour analysis of the sample can be analysed and spawned processes can be easily identified. Volatility is used as a plugin by Cuckoo to

create malicious file process memory dumps and spawned process memory dumps. Volatility also extracts the now de-obfuscated executable to a secure directory on the OS level. It is these extracted executables that are later used for visualisation. A full virtual machine memory dump is also stored at the end of analysis. For Experiment 3, the trigger-based memory experiment, only full operating system dumps are taken so the malicious process id and spawned processes must be identified and then extracted from the full memory dump.

3.4.2 Collect trigger-based memory dumps

As mentioned previously Volatility is used as a plugin for Cuckoo to automatically extract process memory dumps. Using the process id, volatility will extract the malicious process from the process memory dump leaving a copy of the malicious executable binary as it ran in memory.

For Experiment 3, a custom Cuckoo sandbox version 1.0 was developed by Tomer and Hayon Teller (2014). This Cuckoo sandbox version was developed before the integration of process memory dump functionality. As a result, for trigger-based analysis only full memory dumps are created automatically with an API trigger. There are also a limited number of API triggers in their custom Cuckoo Sandbox configuration. With this custom setup, memory dumps are created for every API trigger configured to be detected. There can be multiple memory dumps for every sample executed and as the memory dumps are 1 gigabyte is size the time it takes to run these samples is extensive and the storage space needed is significant for an analysis environment executed off a Windows laptop.

Once a memory dump has been created the malicious binary needs to be identified in the memory dump. To extract the malicious executable, the process ID must first be identified by the binary file name that was injected into the malware analysis environment. From there any spawned processes must be tracked by reviewing the child parent process field for the process id of the original malicious sample. To get a list of processes in the memory.dmp file the following command is executed.

vol.py –profile=Win7SP1x86_23418 -f memory.dmp pslist

Once the malicious process id and its spawned process are identified the processes can be extracted individually from the memory dump using the following volatility command.

vol.py –profile=Win7SP1x86_23418 -f memory.dmp procdump -D . -p + 1234

The process executables are now ready to be visualised. There will be multiple full memory dumps depending on the amount of triggered API's the sample uses.

3.4.3 Naming Scheme

The *CARO* (1991) naming standard is a malware naming outline first introduced as a means of creating a common virus naming standard. The use of this standard as a naming scheme was complicated by the fact that new platforms, anti-virus vendors, an increase of malicious programs and the advent of a variety of detection technologies, made it nearly impossible for all vendors to unify their scanning results and avoid inconsistent naming. As a result, anti-virus vendors started to use their own naming standards, some based on the CARO standard but with a naming mechanism that allows for unified scanning results amongst their own detection technologies. May of the top anti-virus solution providers use a version of the CARO standard. For this study the *Kaspersky* (2018) naming scheme was chosen. It uses a version of the CARO standard for easy recognition of malware sample. The Kaspersky labs naming scheme uses the following naming convention.

[Prefix:] Behaviour.Platform.Name[.Variant]

To give an idea of the structure of a real-life example with the naming convention

HEUR.Trojan-Ransom.Win32.Locky.a

The Prefix is optional, but it can be used to identify the subsystem that detected the malware. In the case of a heuristics engine detecting the malware then the "HEUR" tag is used to start the naming convention. Behaviour specifies what type of malware and what it does. For a Trojan-Ransomware example, it identifies the sample as ransomware. The

platform defines the environment that the file can be executed on i.e. Win32. The ”Name” is the official family name given to the detect file, in this case it is Locky. Finally, the variant is an optional parameter which describes a particular malware family variant.

For this study the Kaspersky naming scheme was used to name the malware variants once they have been visualised. Kaspersky has one of the most comprehensive naming schemes and from quickly looking at the naming convention you can quickly identify the malware family.

The VirusTotal API vtlite.py available on github and developed by Xen0ph0n (2015) was used to identify and name the visualised sample. This program works by submitting a MD5 hash to VirusTotal and sending back basic details like md5 hash, last scanned date, how many anti-virus solutions detected the sample and the name given to the sample by anti-virus solutions. As Kaspersky naming convention was used for naming the visualised samples then the identifying text for the samples was extracted from the output of this API . The section of code is discussed in section 3.6. To further distinguish samples locally, the first date submission recorded from VirusTotal.com of the sample name is appended to the name of the file in order to further identify malware variants in a malware family. If multiple variants from the same date were identified, then a further letter is appended at the end of the date in order to easily distinguish between samples when pre-processing the data.

3.5 Convert the dumps to grayscale images

Conti et al. study reported that they could visualise binary data by scanning every raw byte and converted each value into an image pixel. Nataraj’s study used this same method and was the first to visualise malware as grayscale.

For this study the same process is followed. A small python program was created called bintoIm.py based on the same functionality used by Nataraj. The program bintoIm.py is executed from the cuckoo sandbox memory dump location in order to visualise the extracted processes. Both bintoIm.py and vtlite.py are stored in /usr/local/sbin on the ubuntu cuckoo

sandbox virtual machine to allow for environment wide execution at administration level.

The bintoIm.py is written in Python 2.7 and uses the libraries numpy, os, array, PIL, subprocess and re. The program starts with the sample file input where it's opened as a read-only binary. The length in bytes of the file is calculated and the width is fixed at 256 bytes. A clean-up variable is created called "rem" in the unlikely event that there are odd number of bytes. An array is created with the unsigned character typecode "B" and its populated with the raw data from the input file using the "fromfile" numpy function. The array is reshaped based on the length calculation and fixed width. The array type is changed to uint 8 (unsinged integer 0 to 255) to later allow the python PIL module to process the array as an image. The VirusTotal API vtlite.py is executed against the original binary submitted to the cuckoo sandbox. This will download basic information including the identified name given to the sample by Kaspersky. The regular expression 'Kaspersky Detection:' is created and the proceeding malware identifier name is extracted from the output of vtlite. Finally, the array is saved as a jpeg image.

bintoIm.py

```
#Executed with Python 2.7
#!/usr/bin/python
import numpy, os, array, PIL, subprocess, re
from PIL import Image

#Opens file as readonly binary, calculates length, gives static width,
# byte cleanup variable is declared
filename = raw_input('Enter file here: ')
f = open(filename,'rb')
ln = os.path.getsize(filename)
width = 256
rem = ln%width

#Creates array, reads binary file as machine values, reshapes the array,
# changes array from int64 to unit8
a = array.array("B")
a.fromfile(f,ln-rem)
f.close()
g = numpy.reshape(a,(len(a)/width,width))
g = numpy.uint8(g)

#vtlite APi is ran, Kaspersky given name is extracted
var = os.popen('/usr/local/sbin/vtlite.py -s ../binary').read()
regex = re.compile('Kaspersky Detection: (.*?)\n')
m = regex.search(var)
b = m.group(1)
c = b.strip()

#Render and save the file as a jpeg
im = PIL.Image.fromarray(g)
im.save( c + ".jpeg")
print "Visualisation completed:" + c + ".jpeg created"
```

3.6 Extract features from images

To characterise and classify malware, Natarj's study extracted GIST global descriptors to calculate image-based texture feature's in malware. The GIST global descriptor is based on a studies completed by Oliva and Torralba (2001) Oliva and Torralba (2001*b*) where a statistical representation of the features in an image was used for scene recognition and texture classification.

There are numerous tools available that implement Oliva and Torralba study on calculating a GIST descriptor. The python library pyleargist can support GIST features and supports RGB images. However, the pyleargist library is no longer supported with the latest pillow libraries so its functionality with the older libraries is limited. There are numerous other tools developed by individuals using python and C++ libraries and available on github, but a lot of them only support the ppm image format. The tool chosen for GIST descriptor calculation is called the GIST-Global-Image-Descriptor Tool and was developed by Nrupatunga (2016). It is available on github under the GNU General Public license. It supports batch input of files and includes support for jpeg and png images. The tool is based on the matlab code Oliva and Torralba (2001*a*) made available. To check the accuracy of the tool the results were compared from small samples against my own python GIST descriptor script based on pyleargist. The calculation was the same but Nrupatunga C++ based program provided much improved performance during calculation.

To use the tool, all the images were copied into one input directory. An output location must also be given to use the tool. (Figure 3.4)

gist -i [input directory] -o [output directory]

```
D:\gist-tool>gist.exe -i Mirai -o Mirai_output
100% completed:: Total time passed in seconds: 3.64346
```

Figure 3.4: GIST Descriptor tool execution

The tool creates two files, a gist.txt file containing the gist descriptor values and a features.txt file containing the names of the samples. There are 512 GIST descriptor values calculated for each sample (Figure 3.5). In order to be able to upload the data into WEKA the data was formatted in a way that WEKA can understand. The txt file was loaded into excel and saved as a CSV file. Each row had 512 data values. The features file was also loaded into the same csv file in order to easily distinguish the samples apart. Finally the samples were manually classified into the Malware family groups that were used when searching for the data in VirusTotal json files (Figure 3.6).

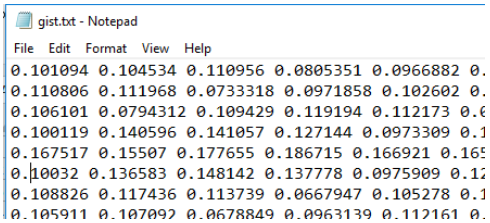


Figure 3.5: Example of GIST.txt

	A	B	C	D	E	F	
1	Classifier	Samples	Value1	Value2	Value3	Value4	Valu
2	Mirai	HEUR_Trojan.Win32.Mira_	0.101094	0.104534	0.110956	0.080535	0.0
3	Mirai	Packed.Win32.Krap.jc_18:	0.101094	0.104534	0.110953	0.081053	0.0
4	Mirai	Packed.Win32.Krap.jc_18:	0.101094	0.104534	0.110954	0.080493	0.0
5	Mirai	Trojan.Win32.Agent.icgh_	0.101094	0.104534	0.110955	0.080259	0.0
6	Mirai	Trojan.Win32.Agent.icgh_	0.101094	0.104534	0.110955	0.080717	0.0
7	Mirai	Trojan.Win32.Agent.icgh	0.101094	0.104534	0.110954	0.080574	0.0

Figure 3.6: Formated data for Weka

3.7 Weka

There are a number of data mining tools and programming libraries available that have machine learning functionality. Python and R are two of the most popular and have a huge number of libraries available for data exploration. They are often used for complicated big data mining tasks and are considered to perform excellent at scale. Waitkato Environment for Knowledge Analysis also known as *WEKA* (2018), is a java-based collection of machine learning algorithms designed for data mining. Weka provides a vast array of functionalities including pre-processing, classification, attribute selection and visualisation. It provides a gui into the word of data mining and has an easy learning curve for basic machine learning and data mining tasks. Two of the machine learning algorithms that are used in this study are K-Nearest Neighbour and Support Vector Machine, and both are supported by Weka. Weka also allows for user friendly visual comparison of samples using plot charts without

the need to write code to plot. Weka was chosen as the preferred data mining tool for this study because of the quick learning curve, online tutorials and supported functionality to aid the results.

3.7.1 Machine Learning Algorithms

The two machine learning algorithms used to compare results in this study are K-Nearest Neighbour (KNN) and Support Vector Machine (SVM). K-Nearest Neighbour is a simple but accurate machine learning algorithm. KNN works by plotting the data samples in a n-dimensional space based on the features given. As a result, every sample is represented by a plot point. When a data set is trained, and a test sample is executed against the training data set it plots the test data set and then searches for its ”K” nearest neighbours based on the distance measured from the training sample. So if ”K = 3” then it searches for the 3 closes samples and each sample votes on what class it is. The Support Vector Machine tries to identify an optimal separating hyperplane in order to decipher the maximum boundary between different classes. Support vectors refers to the points that are closes to the hyperplane. The boundary or margin, is calculated to sperate the data and the goal of SVM is to calculate the maximum margin possible between classes. Both KNN and SVM have been used in multiple studies already discussed in Chapter 2.

3.7.2 Pre-process Data

As data mining depends on the quality of the data, it is important to identify and process missing values, incomplete data, inconsistent data and outlier data. After creating the GIST descriptor calculations and creating a csv file with all malware samples, the data was imported into Weka and saved in Weka’s .arff format. The next step was to pre-process the data.

Incomplete Data

Pre-processing started by looking for obvious missing values or incomplete data. In the Locky variants there were 3 records where the GIST descriptor did not compute any other values other than 0. Upon reviewing these records and generated images, it was discovered that the image files that had been converted were 99% black pixels only. When the behaviour analysis was reviewed for each of the 3 samples it appeared that the process memory dump had been completed just before the process terminated. The terminated process was 2MB executable packed with upx. The file was successfully unpacked using static analysis tool RDG Packer. The nested file was now 1MB in size and using CFF Explorer is showed evidence of zero padding and encryption of the header's and code section. When reviewing the behaviour analysis report in Cuckoo it became evident that the process had not de-obfuscated at this stage. Therefore it was fully packed, zero padding obfuscated and encrypted as the process memory dump was being completed. As reverse engineering samples is out of the scope of this study the 3 samples were removed from the data set.

Outlier Data

An outlier value is where a value evidently deviates from all other values. To determine outlier data values in the data set in Weka, the filtering function interquartile range with a standard deviation value of 5 was executed. Out of a sample set of 372 visualisations, this highlighted 45 samples as being outliers. Upon reviewing these samples visually large portions of the file had black pixels (value 0), also known as zero padding. Using static analysis tools like PEView, CFF Explorer and PEid on the extracted processes showed that they were also packed. In Table 3.1 are 3 different malware visual examples of the outlier samples identified. Each of the extracted processes were packed with either UPX or Armadilio packing tools and showed clear evidence of zero padding obfuscation (Table 3.1) in the static analysis tools.

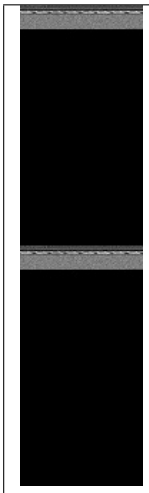
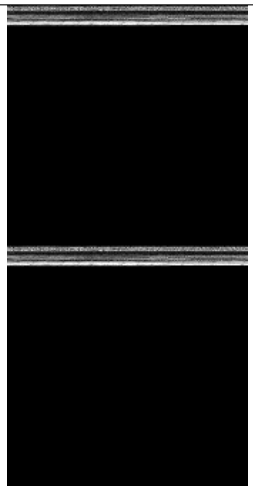
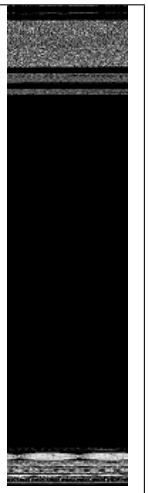
Locky	Hupigon	Cerber
		

Table 3.1: Visualisation of Packed and Obfuscated Samples

3.7.3 Create data models using machine learning algorithms

The idea of creating a data model is so that you can compare new data against an existing data model in order to make predictions on the new data. The current data structure of the CSV files will remain untouched and is used as the data structure for the data model. This includes one nominal field that identifies the malware class, one nominal field that identifies the sample name and 512 numeric fields that identify the GIST descriptor values for each image.

The data model is split into a training and test data sets. Most of the data is used for the training model and a smaller portion of the data is used as a test set. To find the optimal performance for the training and test data models the full data model was executed in Weka against the training algorithms K-Nearest Neighbour with K being 3, and Support Vector Machine. The percentage split module was used to gather the correctly classified instance percentage for various splits and the training model was decided on based on the results.

3.7.4 Measure to what degree new malware samples can be identified

The test model was compared against the training model in order to assess if the test data matches the training data. There are numerous calculations that are compared but the ones that are concentrated for this study are the correctly classified Instance, incorrectly classified instances, true positive rate, false positive rate, precision, recall and the confusion matrix.

Correctly classified instances are when the classifier correctly identifies an instance as belonging to a class. It is calculated as the average true positive rate for all classes. Incorrectly classified instance is when a classifier incorrectly identifies an instance as belonging to a class. True Positive Rate are the portion of actual positives that are identified. False positive rates are the percentage of data classified in the wrong class. Precision is calculated based on the percentage of instance belonging in a class and Recall is a calculation of how many instances of a class is actually belonging to that class. Finally, the confusion matrix is a method for summarizing the performance of the classification model algorithm. The confusion matrix gives an idea of where the classification model is making wrong decisions.

3.8 Test against the state-of-art

Aslan and Samit (2017) compared various static, dynamic, reverse engineering and online tools used to detect existing and unknown malware. They concluded that it is nearly impossible to detect malware using only one tool. The most promising results came from using a combination of static, dynamic, reverse engineering and online tools. These included using the combination of the Cuckoo sandbox and VirusTotal. Pandey and Mehtre (2014) completed a similar study but this concentrated on the performance of analysis tools. They concluded that VirusTotal was the most sensitive and accurate in the category of online scanners.

All test samples MD5 hashes from experiment 2 are uploaded to VirusTotal and a percentage of detected engines are worked out based on the positive results divided by the total

number of scanners executed for the detection. In order to detect how accurate, the findings are per family, the VirusTotal rating is compared to the final results from the test model.

3.8.1 Data Setup

For this experiment the only way to identify the hash value of a binary with a created visualisation image was to manually search for the image on the command line, identify the task it was created in and then run a md5sum of the binary for that task. This would be extremely time consuming to complete for the entire test set used in Experiment 2. In order to be able to compare the test sample results in batch mode against VirusTotal, a hash value for each test sample was identified. To combine the visualisation with its MD5 hash, the information stored in the cuckoo mysql database or in the reported json reports are needed. It was decided to use the cuckoo mysql database as the tools like mysql DB browser and excel can be used interchangeably with the data to quickly find the results. The cuckoo sandbox stores data relating to task identifier and hash of sample in multiple tables. It does not store the malware analysis information as this is stored in a bson and json format in the Linux file system.

To start a copy of the cuckoo mysql database was created and was worked on for this task. To match each cuckoo task id with that of the created image a simple find command was executed on the OS level. From the directory /home/cuckoo/.cuckoo/storage/analyses directory the following command was executed:

”find -name ‘*.jpeg’ >sample.txt”

This find command outputs of all jpeg files created, including the folder path (i.e. task id) to a file sample.txt. This file is imported into excel and sorted based on task id. The arbitrary data including the directory name ”memory” is removed to leave with only 2 columns, sample_id and name. Finally, the file is saved as a .csv file. The csv file is imported into the cuckoo sandbox database as a table samples_name.

The table tasks, samples and the new table samples_name was identified as being needed to extract the relevant MD5 hash for each sample. In order to extract matching md5 for each

task id and including the sample image and md5 a simple sql query is executed.

```
select samples_name.sample_id, samples.md5, samples_name.name
from tasks, samples, samples_name
where tasks.sample_id = samples.id and tasks.id = samples_name.sample_id
```

The output gives us the the task id, along with matching md5 and sample name of the image file. This new output is then reimported to create a new table called classMD5 which is used in the next sql query. The test batch of samples used in Experiment 2 is exported from Weka and saved as a csv file. The csv file is then imported into the Cuckoo DB as a new table test_batch. The test_batch table includes the classifier field, the sample_name field and 512 value fields. To find the matching md5 against the test data for the test_batch table set the following sql is executed. (Figure 3.7)

```
select classmd5.sample_name, ClassMD5.md5 from ClassMD5, test_batch
where ClassMD5.Classifier = test_batch.Classifier
and classmd5.sample_name = test_batch.sample_Name
```

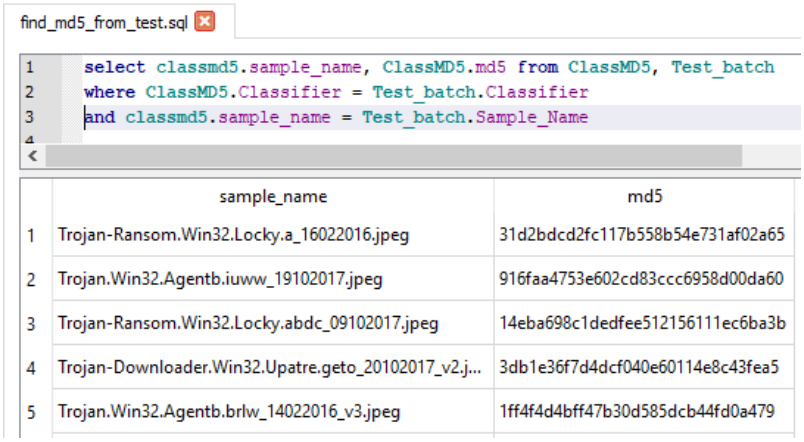


Figure 3.7: Find MD5 from Test set

The results are exported from the database in csv format and split into malware families based on the classifier field. Each malware family md5 hashes are grouped and exported to

a text file. The hash values in these text files are then uploaded using a VirusTotal API. The VirusTotal API vtcheck.py created by MalWerewolf (2015) is used to query VirusTotal.com for all reports of the hashes provided. The advantage of using this API was that it support batch upload of hashes in a file and it reports back whether the file is a positive match or not. The provided API code reported back the total number of anti virus systems in the command prompt output, which is a copy of the virustotal json report which is hard to decipher with all the detail. To enable the report to display both positive results and total number of systems scanned, the code was edited to include the total number of online anti-virus solutions that scanned the malware. A new line ”total = int(json_response.get(’total’))” was inserted at line 16 of the function VT_Request. The resulting string ”+ ’ / ’ + str(total)” was also appended at line 27 in the function VT_Request. The full code for the Virus Total API vtcheck.py is available in appendix A.

```
def VT_Request(key, hash, output):
    params = 'apikey': key, 'resource': hash
    url = requests.get('https://www.virustotal.com/vtapi/v2/file/report', params=params)
    json_response = url.json()
    print json_response
    response = int(json_response.get('response_code'))
    if response == 0:
        print hash + ' is not in Virus Total'
        file = open(output,'a')
        file.write(hash + ' is not in Virus Total')
        file.write('\n')
        file.close()
    elif response == 1:
        positives = int(json_response.get('positives'))
#Edit: added line ”total = int(json_response.get('total'))”
        total = int(json_response.get('total'))
```

```
        if positives == 0:
            print hash + ' is not malicious'
            file = open(output,'a')
            file.write(hash + ' is not malicious')
            file.write('\n')
            file.close()
        else:
            print hash + ' is malicious'
            file = open(output,'a')
  #Edit: Included total string at the end
            file.write(hash + ' is malicious. Hit Count:' + str(positives) + ' / ' + str(total))
            file.write('\n')
            file.close()
    else:
        print hash + ' could not be searched. Please try again later.'
```

3.8.2 Accuracy Calculation

In orer to compare samples from the test set 2 against the results from VirusTotal sample a accuracy percentage is calculated for both. Accuracy measures how close a measurement comes to a pre-existing value. It is often referred to as the closeness to truth. For the VirusTotal calculated data a accuracy measurement is calculated by divinding the positive results by the total number of systems scanned and multiplying that by 100.

$$Accuracy = \frac{Positve}{Total} x100$$

To calculate the accuracy from the test a different set of caluclations are used. The results from Weka give a overall accuracy results for the data set. In order to calculate the accuracy calcualtion for each family the following details are needed.

TP = true positives: number of examples predicted positive that are actually positive
FP = false positives: number of examples predicted positive that are actually negative
TN = true negatives: number of examples predicted negative that are actually negative
FN = false negatives: number of examples predicted negative that are actually positive

Weka calcualtes the True Positive (TP) and the False Positive (FP) and displayed the results in the output. The True Negative (TN) and False Negative (FN) can be calculated from this result. The total sum is always 1 in a calcualtion. If a True Positive is 0.75 then the False Negative is 0.25. If the False Positive is 0.33 then the True Negative is 0.66. The accuracy is then calculated for each malware family based on the following formula.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

3.9 Conclusion

The study was based on previous research completed by Nataraj et al. (2011) and Teller and Hayon (2014). VirusTotal was used to acquire the majority of malware samples used in this study. For the trigger based experiment the zoo and malwaredb were used as data sources. It was executed in a nested virtual environment using the cuckoo sandbox for the automated malware analysis. Volatility was used a plugin to extract process memory dumps of the malware executes. The Kaspersky Naming scheme was chosen for the visualisation samples in order to allow easier identification of the files once visualised. The extracted process was visualised in grayscale and then a GIST global descriptor was calculated for each sample. The samples were classified based the malware family and a data model was created. The data model was then split using Weka into training and test models. New malware variants could then be detected based on the results of this model. Finally, the results are compared against VirusTotal Anti-Virus online scanner solutions to detect how accuracy of the approach used on this study.

Chapter 4

Implementation

4.1 Experiment 1

4.1.1 Question

Can new malware variants be identified from malware families using memory analysis visualisation? This experiment attempted to measure how successfully visualising malware in memory has on detecting new variants of the same family.

4.1.2 Background

Nataraj et al. (2011) research on visualisation using static analysis was based on de-obfuscated samples or samples where the packer was the same. In this experiment the goal is to demonstrate how it is possible to identify malware variants by visualising variants using memory analysis and how this process can overcome some obfuscation. For the first experiment samples for the Mirai botnet on Windows 7 x86_32 are visualised for both static and memory analysis and then compared visually. The second experiment samples the heavily obfuscated Locky variants in an attempt to visually find a comparison.

4.1.3 Experiment 1.1 - Mirai

22 Mirai samples were collected from the VirusTotal malware sets provided dating October 20th, 2017 and December 5th, 2017. I used the first submission date provided via VirusTotal analysis reports as a gauge of how old the samples were. For visualisation the Mirai samples used range from a first submission of the 7th March 2016 to the 18th of October 2017. I manually checked each sample for a packer using PEid and RDG Packer. It was found that some Mirai samples had been packed with AsProtect versions ranging from 1.2 to 1.3.

In Nataraj's study they completed visualisation on samples that were not executed. Therefore, to compare static samples against in-memory samples, grayscale images of the binaries were created before execution. This would allow static visualisation to be compared to in-memory visualisations.

All samples are submitted in batches to the cuckoo sandbox for analysis with process memory dump and full memory dump enabled. The cuckoo sandbox using the volatility plugin extracts the process from memory and creates a process memory dump. It then extracts the process directly from that dump. Once the automated analysis is finished, a full review of the sample in Cuckoo's web front is completed. The static, behaviour and process memory analysis is reviewed to ensure successful execution of the sample. Once the analysis is complete, process memory dumps and the extracted process can be seen in the following location for each task.

/home/cuckoo/.cuckoo/storage/analyses/[task id]/memory.

After navigating in the terminal to the memory process location, the visualisation of the captured binary from memory files is started by executing bintoIm.py. As disucessed in section 3.5, the program converts the extracted executable to a grayscale image and extracts the given Kaspersky name for the sample and saves it as a jpeg to the memory directory. (Figure 4.1)

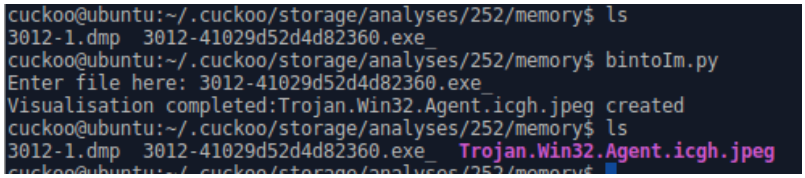


Figure 4.1: bintoIm.py execution

The VirusTotal report is manually reviewed online and the first submission date is taken from the details in the online report. This date is appended to the end of the file in order to make identification of the same variants with the same given identification name easier for this study. e.g. Trojan.Win32.Agent.icgh_18102017.jpeg. If multiple variants from the same day are identified then an additional letter is appended at the end of the date in order to allow each sample to be individually identified, e.g. Trojan.Win32.Agent.icgh_18102017b.jpeg. Visualisation of the original binary is also completed at this stage and the name is appended with the word ”STATIC”. In table 4.1 are details of 7 of the 22 Mirai samples used for this experiment. The table contains the unique md5 hashes and the Kaspersky given identification name.

Sample	MD5	Kapersky Given Name
Sample 1	16cfaaa1a875e25545249a0fa3585c74	Trojan.Win32.Agent.icgh
Sample 2	da961caacb748f1dac5399be32482545	HEUR:Trojan.Win32.Mira
Sample 3	a620e085dd01e11f31889a49dff028c2	Trojan.Win32.Agent.icgh
Sample 4	a349ac9a56dd3856ddd95e8121120637	Trojan.Win32.Agent.icgh
Sample 5	be2cbb05022cc8a61d70865f6a216dc0	Trojan.Win32.Agent.icgh
Sample 6	a5b33910ca458a11a219a626c462a75e	Trojan.Win32.Agent.icgh
Sample 7	bc7058a34345c7149a92b6cb766791c6	Packed.Win32.Krap.jc

Table 4.1: Mirai Samples Details

Table 4.2 contains the static visualisation of the original Mirai binaries. The samples can be identified by the MD5 from table 4.1. By reviewing the static samples there is clear indication of zero padding, potential packing and encryption of the samples. What is most notable is that each sample looks different. Therefore, using Nataraj’s technique for detecting obfuscated Mirai variants would not work as each sample is visually different.

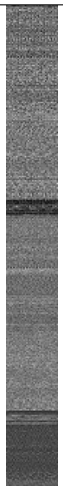
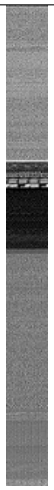
Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7
						

Table 4.2: Mirai Static Visualisations

Table 4.3 contins the visualisation of Mirai binaries taken in memory. Each sample taken from memory are visually similar. All 22 Mirai samples taken were visually similar and could easily be identifed.

Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7
						

Table 4.3: Mirai Memory Visualisations

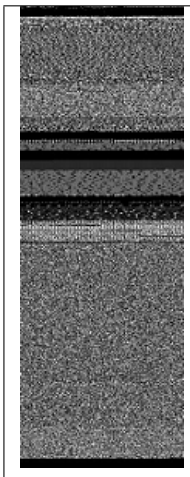
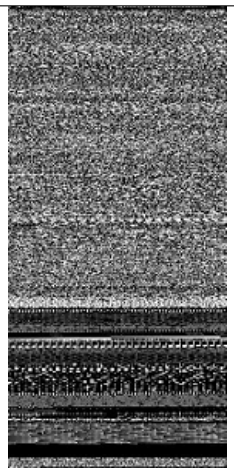
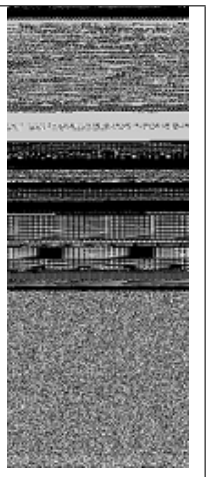
4.1.4 Experiment 1.2 - Locky

There were 37 Locky malware variants obtained from the VirusTotal data set with a total of 48 visualisations created. The samples range in first submission date from February 2016 up until October 2017. The interesting thing with Locky was the changing obfuscation techniques used in multiple variants over time. The earliest samples of Locky had only basic obfuscation and could be reverse engineered. The newest variants use advanced obfuscation techniques and are therefore difficult to detect. Table 4.4 contains the detailed identification of the 7 samples including MD5 and Kaspersky given that are discussed in this section.

Sample	MD5	Kapersky Given Name
Sample 1	bafb370b3e76d1a0b3c52ec1ad57b5eb	Trojan-Ransom.Win32.Locky.d
Sample 2	baf9e357a90ccf15bcf875719b01ebbc	HEUR:Trojan.Win32.Locky
Sample 3	0c0d798add0c4e450a1c8bf3824ea989	HEUR:TrojanWin32Locky
Sample 4	0c4bcd747e26da130ea483ab0ad89996	BackdoorWin32Andrommxie
Sample 5	e7e5628f67cb2fa99a829c5a044226a4	HEUR:TrojanWin32Locky
Sample 6	230606dd8b0d62e2a8a04ef61b2d8707	HEUR:TrojanWin32Locky
Sample 7	6a8eb2a593079817edf74a48c48fec5e	HEUR:TrojanWin32Locky

Table 4.4: Locky Sample Details

Table 4.5 contains the static visualisation of Locky variants. The samples can be identified by the MD5 from table 4.4. By reviewing the static samples there is indication of multiple types of obfuscation. Each sample is visually different so, using Nataraj's technique for detecting obfuscated Mirai variants would not work as each sample is visually different.

Sample 1	Sample 2	Sample 3
		

Sample 4	Sample 5	Sample 6	Sample 7
			

Table 4.5: Locky Static Visualisations

It is clear that from the visualisations that Nataraj's static visualisation would not be able to detect these variants. Table 4.6 are the samples taken from memory. Sample 1 was a Locky variant first submitted to VirusTotal in Febuarary 2016. This variant was fully de-obfuscated when captured in memory so any variant of this sample could be detected in much the same way that the Marai variants can be detected. Sample 2, 3 and 4 were first submitted in September and November 2016 and April 2017. While each of the visualisations are different there are some visual similarities that can be seen. This is where code obfuscation is seen in the captured memory visualisation.

Sample 5, 6 and 7 are from August, September and October 2017. These 3 samples use different obfuscation methods compared to the older samples. What is observed in the visualisation is encrypted code in memory. While the visualisations are very similar due to the advanced obfuscation methods used, it is not possible to see the code in this visualised format. According to the VirusTotal analysis reports for these samples use RSA encryption. The code is encrypted and re-encrypted as it executes. Therefore, any memory sample taken can only see the code block that has been de-obfuscated as everything else is still encrypted. However, for the purpose of visualisation similar variants can be detected.

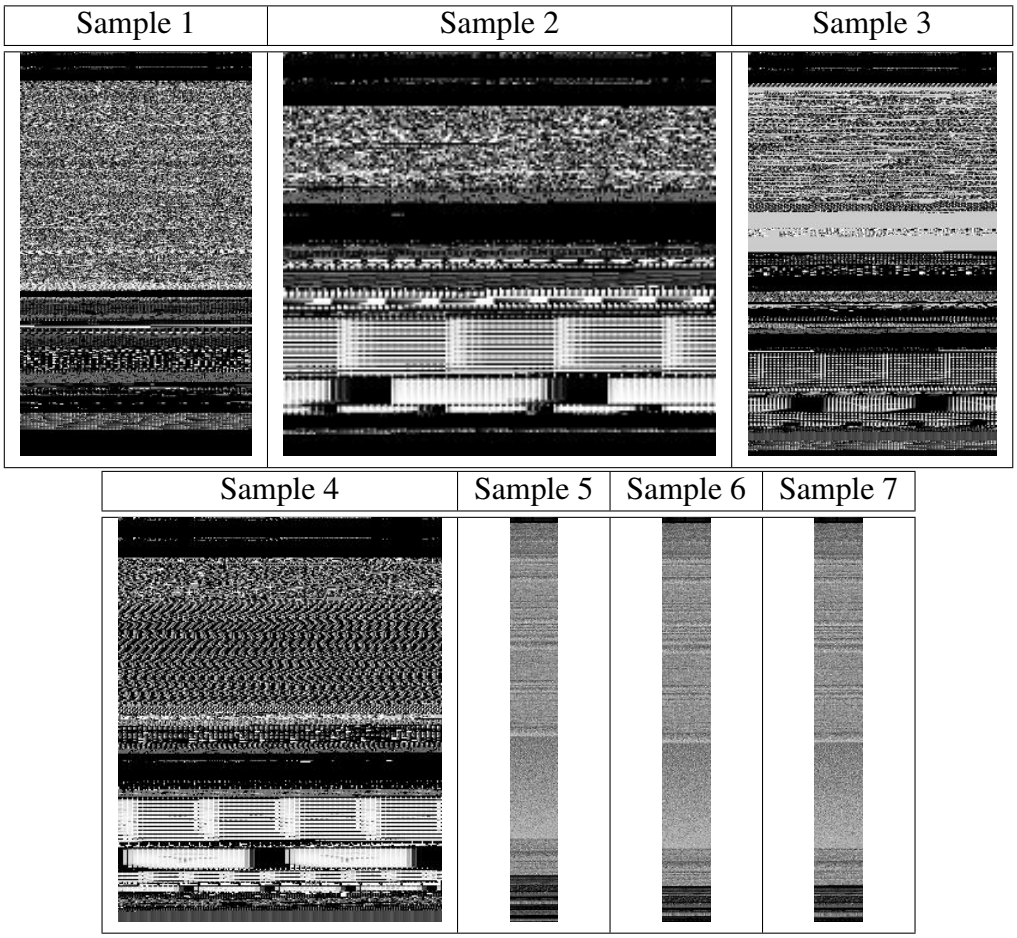


Table 4.6: Locky Memory Visualisations

Visualisation of obfuscation

For some Locky variants there were numerous process memory dumps created. This was determined by the volatility baseline capture which highlights the changes during the analysis of the malware variant. Table 4.7 contains the visualisation of a Locky variant process as it was executed in memory. The first process dump showed a packed executable with zero padding with a size of 4mb. This sample visualisation was originally picked up as an outlier value when the data set was examined using pattern matching analysis of the data set. The 2nd process dump showed a removal of the zero padding, but the file is still packed with a size of 400KB. The 3rd process dump showed the true visualisation of the process where the file is unpacked with a size of 80KB. The texture features in the file are clearly visible and will enable pattern analysis using machine learning algorithms. As a result, this Locky variant can be detected using the 3rd process dump visualisation and this finding helped with easily identifying outlier samples that can be removed from the data set for later experiments.

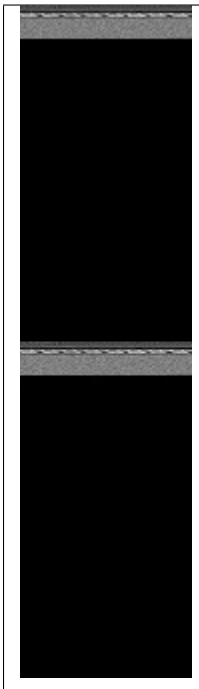
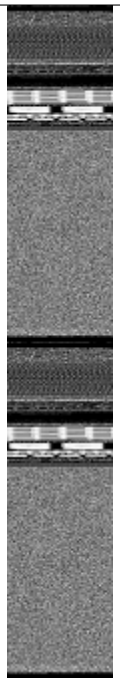
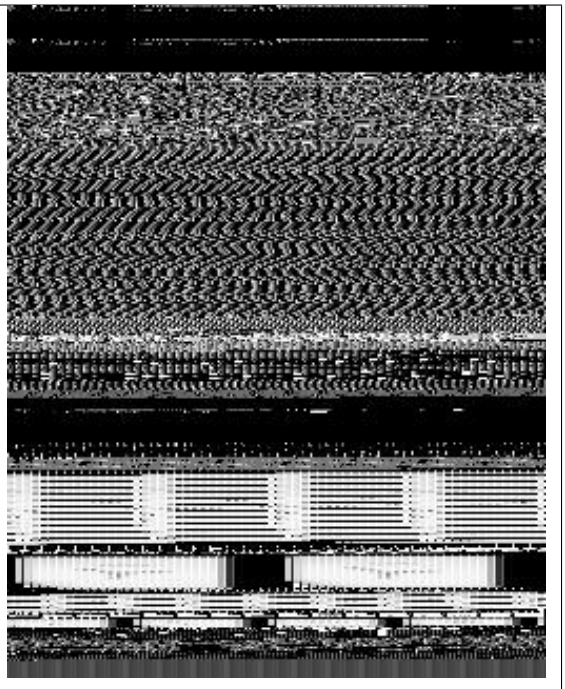
Zero Padding	Packed	De-obfuscated
		

Table 4.7: Visualisation of de-obfuscation of Locky

4.1.5 Summary

The first experiment showed the advantages of visualising memory analysis over static analysis. The experiment showed how features can be visibly identified and therefore used in pattern matching analysis. As a further result of the finding around the data set, it was determined that the outlier samples identified as packed with zero padding could be removed from the data set, as these malware samples can be identified using the other visualisation process memory dumps already in the data set. It was decided because of the large benign data set that benign values were also removed from the proceeding experiments. Legitimate programs like explorer, calc, cmd, powershell and dwm are used by malware as part of the attack vector. They are also used as diversionary tactics my malware authors to obscure or delay reverse engineering. By keeping benign processes in the data set, it will result in a high number of false positives and mis-classification of malware. As a result benign samples are removed for the proceeding experiments.

4.2 Experiment 2

4.2.1 Question

Using pattern matching analysis, can new malware variants be detected? This experiment attempted to measure how using pattern matching analysis in visualisation can be used to detect new malware variants.

4.2.2 Background

Nataraj et al. (2011) research on visualisation of static binaries used extracted GIST features and the K-NN classification algorithm in pattern matching analysis in order to detect new malware variants. The goal of this experiment is to measure to what degree new malware variants can be found using pattern matching analysis. This experiment used K-Nearest Neighbour and Support Vector Machine (SVM) on visualisation of memory samples and

then compared the results to see if an improved identification accuracy can be obtained when using SVM.

Some malware samples created multiple process dumps and therefore multiple images, so, for some samples there were multiple images to help identify that sample. Table 4.8 displays the total number of samples and total number of visualisations used in this experiment.

Malware Family	Total Samples	Total Visualisations
Mirai	22	22
Locky	37	48
Emotet	16	55
Hupigon	17	144
Ramnit	12	27
CoinMiner	16	30
Benign	39	42
Total	159	368

Table 4.8: Malware families

Each image could be identified individually by the Kaspersky naming scheme, the first submission data to VirusTotal and an additional optional alphabetical character in the name of the file. Each image was grouped in a folder by malware family. Using the GIST-Global-Image-Descriptor-Tool each malware family had its GIST descriptor calculation completed. The resulting data were imported into a csv file which included the malware family identified as a class, the jpeg image file name and the 512 GIST descriptor calculations. The csv format was used in order to import the sample data set into Weka.

4.2.3 Preliminary Test Run

Several preliminary test runs were executed using K-NN and SVM algorithms with cross validation set to 10 against all the data to check how aligned it was. These preliminary checks showed that a high number of benign samples were matching with malware samples. Upon further investigation of the Cuckoo sandbox behaviour analysis results and the visualisation samples taken, it became evident that windows processes that were identified as benign were

being used by some of the malware samples. Figure 4.2 shows explorer.exe plotted against itself. The plot is a straight diagonal line. When explorer.exe data was plotted against one of the matching Hupigon data sets (Figure 4.3) it became evident that there was a close match between the malicious and legitimate binary.

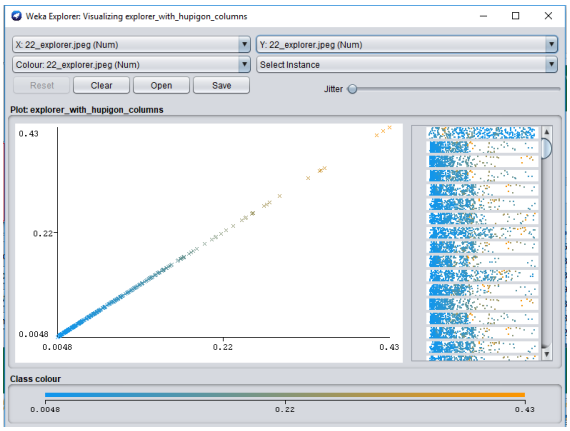


Figure 4.2: Explorer plotted

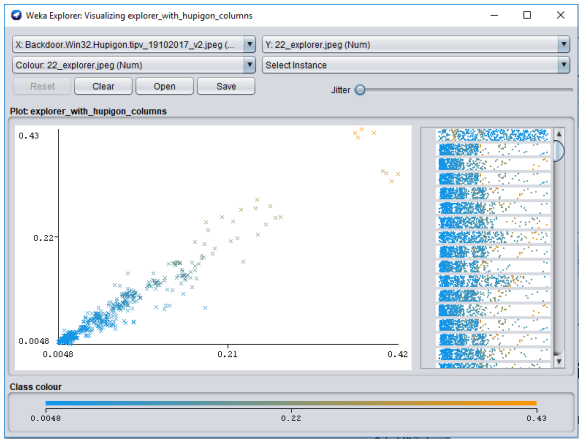


Figure 4.3: Explorer plotted against Hupigon

Taking into account the information already known about Hupigon and the behaviour analysis report available from the cuckoo sandbox it became evident that explorer.exe had in fact been injected by the hupigon malware binary. A further investigation into benign samples matching malware was completed.

Visualisation of Code Injection

When reviewing other malware samples, it was found that they too took advantage of process hollowing in explorer.exe and other benign samples. When the data is loaded into Weka it became visually evident the similarities amongst binaries. When reviewing the extracted images, it became visually evident the large section of code injected into explorer.exe. This behaviour was confirmed for some samples when the behaviour analysis was reviewed. In Table 4.9 are 4 different malware families where different variants injected code into the explorer.exe.

Sample	MD5	Name
Explorer	8b88ebbb05a0e56b7dcc708498c02b3e	Windows 7 32 bit explorer.exe
CoinMiner	a8696d104b010865a7687eb0e92a1904	Trojan.Win32.Miner.sxvj
Fareit	56f7acee738aba5b917c46edc18ed34c	Trojan.Win32.VBKrypt.nrww
Hupigon	b2b886ebf88cf2b0fdb4a39fbb8a7276	Backdoor.Win32.Hupigon.tipv
Locky	0bd30fca55a734b29218d45d7dab1a04	Trojan-Ransom.Win32.Cryptodef.cls

Table 4.9: Explorer Injected Samples

Upon comparing benign samples to the malware samples, it became evident of that some windows processes had been injected by the malware sample. Explorer is often given trusted status by anti-virus solutions, so the process hollowing obfuscation allows an illegitimate process to masquerade as a legitimate process used can allow malware full system, registry or network access.

When comparing explorer.exe samples against the injected explorer.exe samples it was visually evident where in explorer.exe a large section of the code was injected (Table 4.10). By comparing the samples in a hex editor, I was able to compare samples and extract the large section of injected code. Viewing the file in the hex editor it was evident that the code was encoded or encrypted.

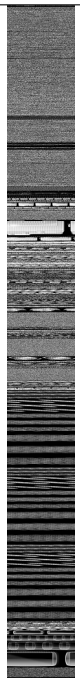
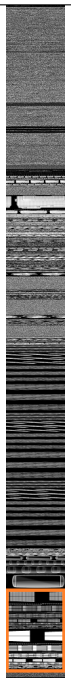
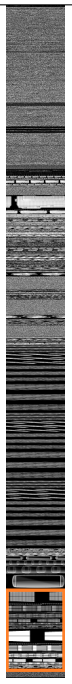
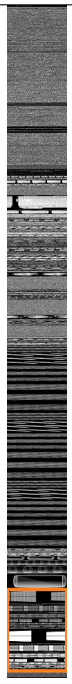
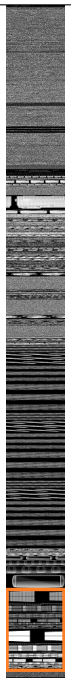
Explorer	CoinMiner	Fareit	Hupigon	Locky	Injected Code
					

Table 4.10: Process Injection into explorer.exe

4.2.4 Prepare Training and Test Models

The full data model is split into a training set and a test set. Most of the data was used in the training model and a smaller portion of the data was used in the test set. To find the optimal performance for the training and test data models, the percentage module was executed to test the split of the data. The algorithms K-Nearest Neighbour with K being 3, and Support Vector Machine was used to measure the accuracy. The percentage split module was executed where the split was changed in multiples of 5 and executed in order to identify the optimal performance of the training and test split. The correctly classified instances results are displayed in table 4.11. It was found that with K-NN set to 3 that the optimal training result was with a 75/25 split. The optimal training result for SVM was with a 90/10 split but the difference between the 75/25 split is only 0.0478. As the difference between K-NN 75/25 and 90/10 is 1.2971, It was decided to use the 75/25 split for the training test

model split.

Training Split	K-Nearest Neighbour = 3	Support Vector Machine
65 / 35	84	89
70 / 30	84.8837	91.8605
75 / 25	87.5	93.0556
80 / 20	85.9449	91.2281
85 / 15	86.0465	93.0233
90 / 10	86.2029	93.1034
95 / 05	85.7143	92.8571

Table 4.11: Training Samples Split

The target was to split the data into a 75% and 25% split. Weka supports stratified sampling which enables the even distribution of samples across the training and test set. The data was split using the pre-processing filter weka.filters.supervised.instance.stratifiedRemoveFolds. The parameters number of folds selected was set to 2 and the seed was set to 10. This meant that the data sample was split into 2 equal parts and the random number for shuffling the dataset is set to 10. One section of data was selected and split again using the same method. This yielded a 25%-25% split. The data from one of the 25% split was merged into what was left over from the 50% split which then yielded a 75%-25% stratified split. The training model (Figure 4.4) had a total of 245 instances while the test model had 82 instances.

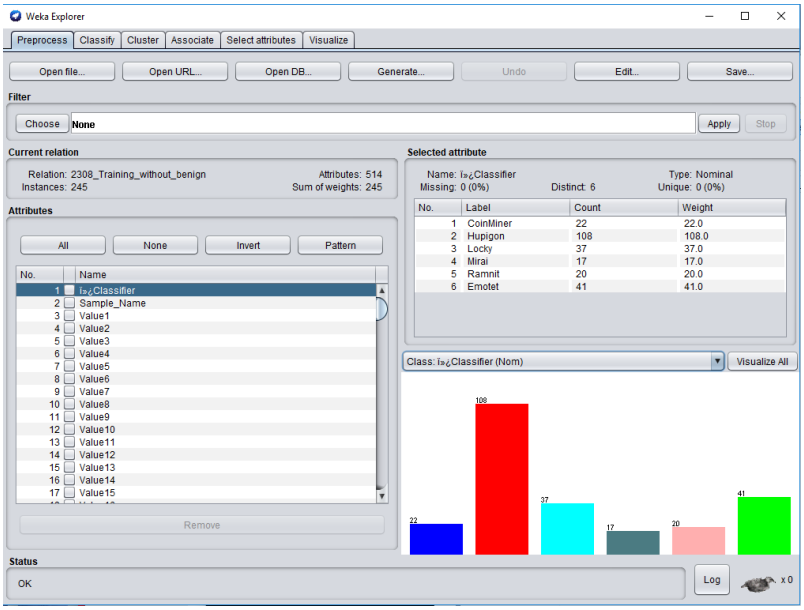


Figure 4.4: Training Model in Weka

4.2.5 Compare Training and Test Models

The entire data model had 327 samples with 245 being assigned to the training model and 82 being assigned to the test model. Using the classification functionality of Weka the training set calculation is executed using K-Nearest Neighbour classification, with K = 3. The test set is executed against the training model with K = 3.

When the test model was executed against the training model the correctly classified instance value were 84.1463 % or 69 instances. This is equivalent to the weighted average True Positive Rate. The incorrectly classified instances were 15.8537% or 13 instances. The detailed accuracy results are detailed in table 4.12. It showed some encouraging results from some malware families including Mirai and Ramni which were classified at 100% True Positive.

Malware Family	TP Rate	FP Rate	Precision	Recall
CoinMiner	0.625	0.014	0.833	0.625
Hupigon	0.972	0.109	0.875	0.9721
Locky	0.583	0.014	0.875	0.583
Mirai	1.000	0.000	1.000	1.000
Ramnit	1.000	0.0.67	0.583	1.000
Emotet	0.714	0.015	0.909	0.714

Table 4.12: K-NN Test Result

Table 4.13 contains the confusion matrix for this sample test set. A Confusion matrix is used to measure the performance of the classification model algorithm. The results show 3 CoinMiner, 1 Hupigon, and 1 Locky sample being classified as Ramnit malware. This seems to identify the issue of common patterns used amongst different malware. Since the dataset is biased towards the Hupigon class which had a large sample set it was expected that the Hupigon class may confuse some of the other class's, but this was not the case as all bar 1 Hupigon sample was identified as a Huigon variant.

Classified As –>	a	b	c	d	e	f
a = CoinMiner	5	0	0	0	3	0
b= Hupigon	0	35	0	0	1	0
c = Locky	0	3	7	0	1	1
d= Mirai	0	0	0	5	0	0
e = Ramnit	0	0	0	0	7	0
f = Emotet	1	2	1	0	0	10

Table 4.13: K-NN Test Confusion Matrix

The Support Vector Machine (SVM) algorithm was then executed against the 245 samples in the training set in order to set a baseline. The test set of 82 samples was once again executed against the training set. The correctly classified instance value was 89.0244 % or 73 instances. The incorrectly classified instances were 10.9756% or 9 instances. The detailed accuracy results are detailed in table 4.14. The results seemed improved when compared to the K-NN. It showed some True Postive rate of 100% for Hupigon, Mirai and Ramnit.

It showed improved scores for all malware families except for Locky which got the same result.

Malware Family	TP Rate	FP Rate	Precision	Recall
CoinMiner	0.750	0.000	1.000	0.750
Hupigon	1.000	0.043	0.947	1.000
Locky	0.583	0.029	0.778	0.583
Mirai	1.000	0.000	1.000	1.000
Ramnit	1.000	0.0.27	0.778	1.000
Emotet	0.890	0.033	0.891	0.890

Table 4.14: SVM Test Result

Table 4.15 contains the confusion matrix for the SVM sample test set. It clearly shows a improved performance as less malware variants are being identified in other malware classes. The main noticeable difference is that 3 Locky variants are being identified as Emotet where as when using K-NN 3 Locky variants were identified as Hupigon.

Classified As –>	a	b	c	d	e	f
a = CoinMiner	6	0	1	0	1	0
b= Hupigon	0	36	0	0	0	0
c = Locky	0	1	7	0	1	3
d= Mirai	0	0	0	5	0	0
e = Ramnit	0	0	0	0	7	0
f = Emotet	0	1	1	0	0	12

Table 4.15: SVM Test Confusion Matrix

4.2.6 Comparing Classification Errors

Using the Weka classifier visualiser enables the visualisation of classification errors. There were 3 Locky samples that were incorrectly classified as Hupigon. In order to investigate the classification error, the visualise classifier error functionality of Weka was used. Figure 4.5 plots all the classification errors in a graph for K-NN classification. The error highlighted is one Locky class that is classified as a Hupigon with the prediction margin of -0.1333. The sample name allows for the identification of the jpeg image for visual comparison.

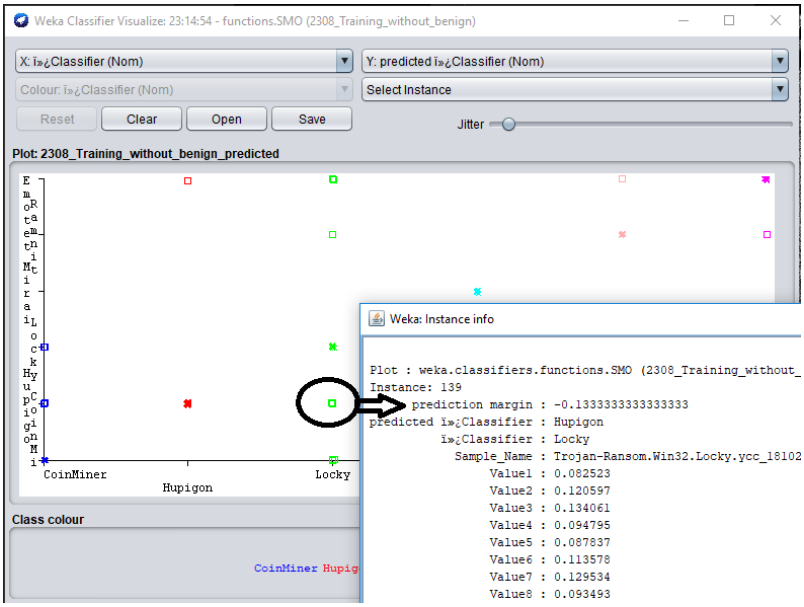


Figure 4.5: Visual classifier error

In table 4.16 the Hupigon and Locky sample can be compared side by side. When comparing the 2 samples visually you can see some visual comparison, but not an exact match. However, when you take into account the prediction margin which is 13%, this is quite a high False Negative margin so perhaps with a larger data set this classifier error would not occur.

When investigating the SVM confusion matrix there are 3 Locky samples that are classified as Emotet. when using the Weka visualised classifier error functionality, it was found that the prediction margin was -0.047. This prediction margin is 4.7% and a lot closer than the previous example. Again, the samples can be identified by the image name and they can be compared visually. In table 4.16 visually these samples have similar features. When reviewing further information regarding the samples both samples were submitted to VirusTotal in 2015 so are quite old. It is possible that these old malware variants shared some code, packers or encoding techniques.

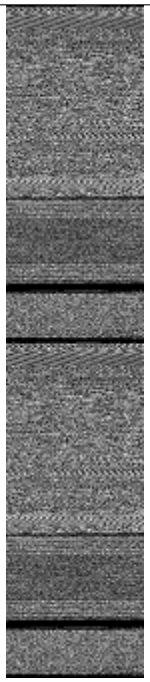
K-NN Hupigen	K-NN Locky	SVM Locky	SVM Emotet
			

Table 4.16: Visual difference

4.2.7 Comparing Precision Variance

When reviewing the data, precision variance of the Ramnit sample stood out. It had a True Positive rate of 100% for K-NN and SVM however there was a variance in the precision calculation where for K-NN it was 58.3% and for SVM it was 77.8%. Upon reviewing the visual error classifier it was distinguished that Ramnit had some samples that were classified correctly but had a close match to another malware sample. (Figure 4.6)

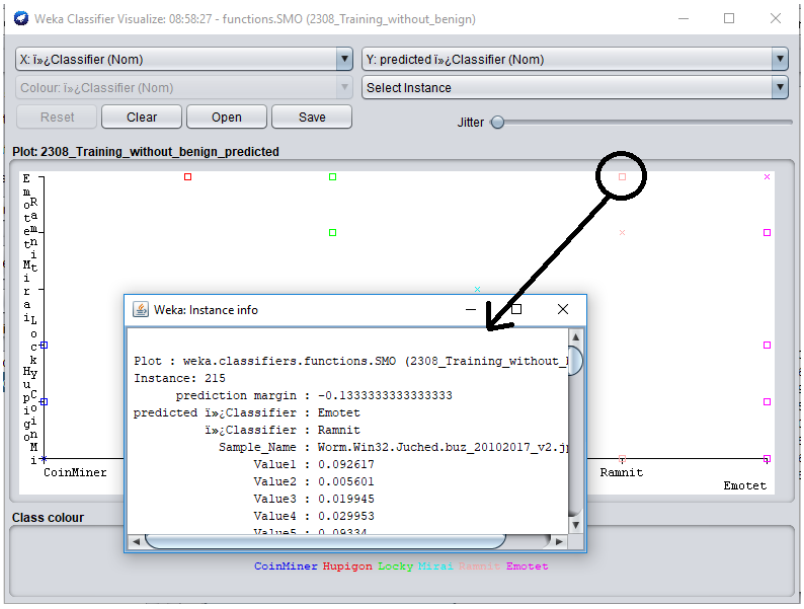


Figure 4.6: Ramnit PrecisionVariance

Upon reviewing the data further and in particular comparing an Emotet sample against a Ramnit sample the discrepancy became clear. The visual samples identified may have looked different but each sample was extracted from different sized samples. Table 4.17 contains the Ramnit and Emotet samples identified with a precision error. The Ramnit sample was created from a captured process memory dump with a size of 311KB. The Emotet sample was captured from a process memory dump measuring 33KB. As a result, it is feasible to suggest that the pattern matching algorithms precision measurement picked up the similarity between all of the Emotet sample and only some of the Ramnit sample. The Ramnit sample could feasibly contain the Emotet sample.

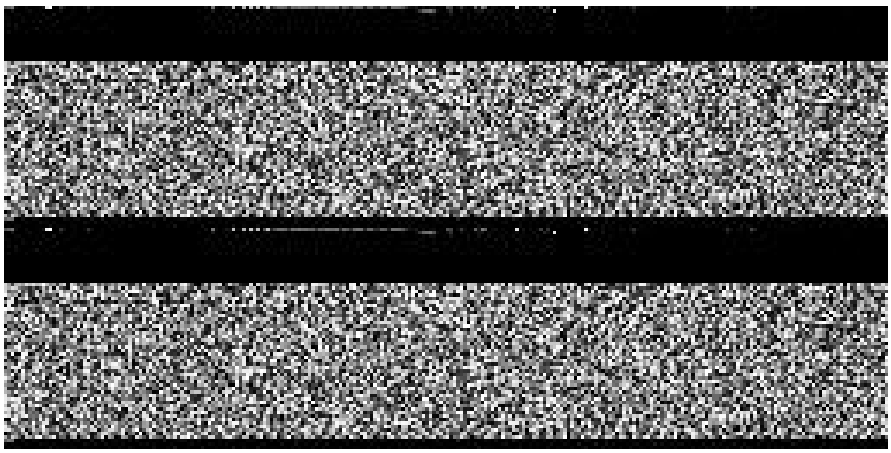
K-NN Ramnit	K-NN Emotet
	

Table 4.17: Ramnit Precision Variance

4.2.8 Summary

The results show that the GIST global descriptor features are sufficiently descriptive for the purpose of classification of malware in memory. Using GIST features with pattern analysis shows the ability to identify and group malware variants and identify malware families. Using the visualising samples along with pattern analysis allowed for a deeper understanding and interpretation of the results. The SVM algorithm proved to be the better performer of the 2 algorithms.

4.3 Experiment 3

4.3.1 Question

Using visualisation of API trigger-based memory analysis, determine if new malware variants can be identified using pattern matching analysis. This experiment attempted to measure how by visualising captured API trigger memory dumps and using pattern matching analysis, new malware variants can be detected.

4.3.2 Background

A combination of the visualisation and pattern matching analysis research by Nataraj et al. (2011) and Teller and Hayon (2014) API trigger based memory dumps method is used in this experiment. The goal is to measure if new malware variants can be detected using visualised API trigger memory dumps in pattern matching analysis. This experiment also compares the performance of KNN and SVM in pattern matching analysis. This experiment used a smaller data set and included the Cerber, Kovter and Trickster malware variants in the place of Mirai, Hupigon and Ramnit.

4.3.3 Experiment

All samples were submitted and executed in the cuckoo sandbox environment. This experiment was executed on an older customised version for cuckoo sandbox 1.00 and volatility 2.04. As each memory dump was 1 Gigabyte in size it would take some time for all trigger-based API memory dumps to be created. Multiple memory dumps were created including one before analysis and one after analysis. That meant that a minimum of 3 memory dumps and sometimes up to 10 memory dumps were taken for each analysis. As a result, this process was time consuming and took up a lot of limited file storage space. Each trigger-based API memory dump that is created must be searched for the submitted binary file and any child process before the process can be extracted from the memory dumps.

vol.py -profile=Win7SP1x86 23418 -f memory.dmp pslist

Once the processes are identified then the process is manually extracted from memory

vol.py -profile=Win7SP1x86 23418 -f memory.dmp procdump -D . -p + 1234

The visualisation process is executed on all extracted executables and the malware families are stored in separate folders. Using the GIST-Global-Image-Descriptor-Tool each malware family had its GIST descriptor calculation completed. The resulting data were imported into a csv file in order to import the sample data set into Weka.

The full data model was a smaller set of 102 samples. Using the same procedure from section 4.2.4 Prepare Training and Test Models, the data set was split with a 75% training and 25% test split which left 78 samples (Figure 4.7) in the training model and 24 samples in the test model set.

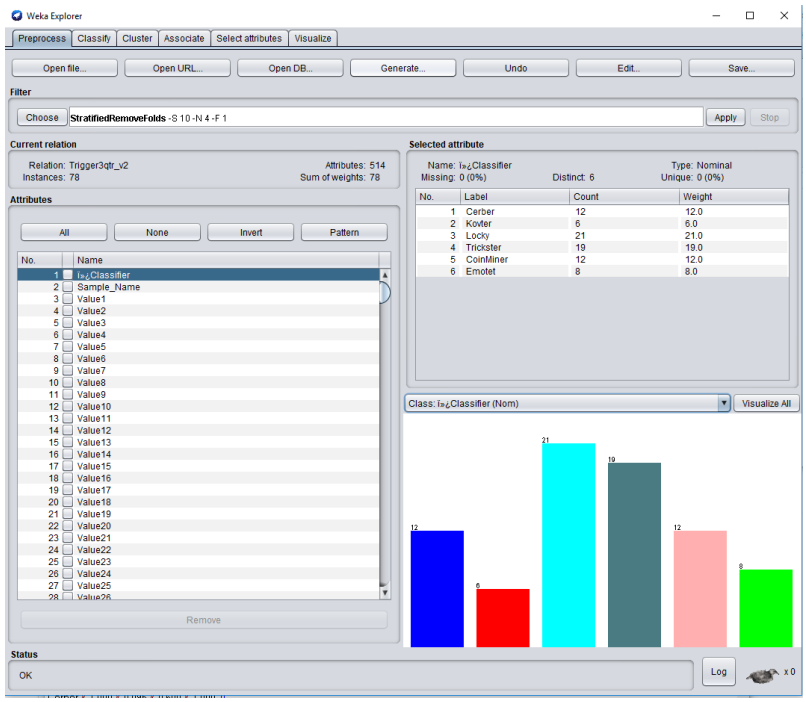


Figure 4.7: Trigger-Based Training Model in Weka

Using the classification functionality of Weka, the training set calculation is executed using K-Nearest Neighbour classification and with K = 3. The test set is also executed against

the training model with K = 3. The results showed the correctly classified instance value was 79.1667 % or 19 instances. This is equivalent to the weighted average True Positive Rate. The incorrectly classified instances were 20.8333% or 5 instances. The detailed accuracy results are detailed in table 4.18.

Malware Family	TP Rate	FP Rate	Precision	Recall
Cerber	1.000	0.095	0.600	1.000
Kovter	1.000	0.000	1.000	1.000
Locky	0.857	0.176	0.667	0.857
Trickster	0.500	0.000	1.000	0.500
CoinMiner	0.750	0.000	1.000	0.750
Emotet	1.000	0.000	1.000	1.000
Weighted Avg.	0.792	0.063	0.853	0.792

Table 4.18: Trigger Based K-NN Test Result

Table 4.19 contains the confusion matrix on the test sample set when using K-NN. The performance is good across all the classes except for the Trickster class where 2 samples get incorrectly classified as Locky and 1 sample as Cerber.

Confusion matrix

Classified As –>	a	b	c	d	e	f
a = Cerber	3	0	0	0	0	0
b = Kovter	0	2	0	0	0	0
c = Locky	1	0	6	0	0	0
d = Trickster	1	0	2	3	0	0
e = CoinMiner	0	0	1	0	3	0
f = Emotet	0	0	0	0	0	2

Table 4.19: Trigger Based K-NN Test Confusion Matrix

The Support Vector Machine (SVM) algorithm was then executed against the sample training and test samples sets. This showed the correctly classified instances as 87.5% or 21 instances and incorrectly classified instances as 12.5% or 3 instances. Table 4.20 shows

the detailed accuracy results for SVM algorithm executed against API trigger-based memory visualisation.

Malware Family	TP Rate	FP Rate	Precision	Recall
Cerber	1.000	0.048	0.750	1.000
Kovter	1.000	0.000	1.000	1.000
Locky	0.857	0.000	1.000	0.857
Trickster	0.667	0.000	1.000	0.667
CoinMiner	1.000	0.100	0.667	1.000
Emotet	1.000	0.000	1.000	1.000
Weighted Avg.	0.875	0.023	0.913	0.875

Table 4.20: Trigger Based SVM Test Result

Table 4.21 contains the confusion matrix on the test sample set when using SVM. It shows a improved accuracy for CoinMiner and a small increase in detection for Trickster. The performance is generally good except for Trickster which had 2 samples as being incorrectly classified as Locky.

Classified As –>	a	b	c	d	e	f
a = Cerber	3	0	0	0	0	0
b = Kovter	0	2	0	0	0	0
c = Locky	1	0	6	0	0	0
d = Trickster	0	0	0	4	2	0
e = CoinMiner	0	0	0	0	4	0
f = Emotet	0	0	0	0	0	2

Table 4.21: Trigger Based SVM Test Confusion Matrix

In the confusion matrix for K-NN it became evident that 2 Trickster variants were incorrectly classified as Locky samples. Table 4.22 contains samples of the visualisations of the Locky and Trickster samples that were matching. While there are minimal features matching here, this mismatch has most probably occurred because of the small data set.

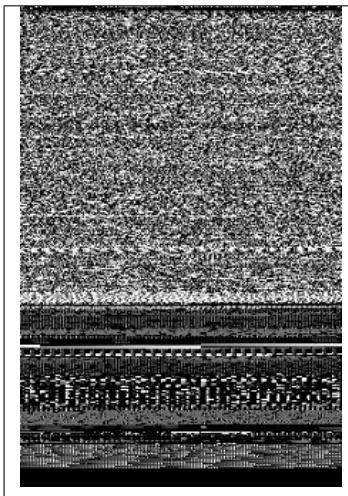
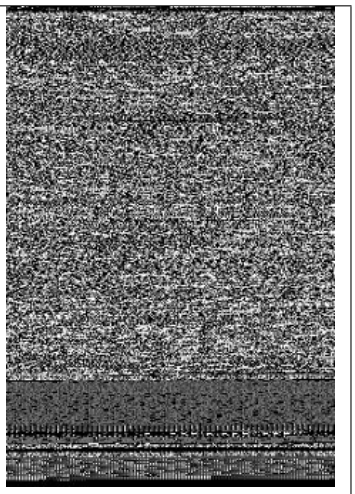
Locky	Trickster
	

Table 4.22: Locky and Trickster comparison

4.3.4 Summary

The results showed a promising 79.2% accuracy for K-NN and 87.5% accuracy for SVM classification. This experiment was executed on a smaller data sample which seemed to affect the overall results. While the process did seem quite accurate for some malware variants and families, it did not perform as well as experiment 2. API triggers in the custom sandbox were limited to 20 which also hindered this experiment. File storage and time taken to execute a sample were the biggest drawbacks in this experiment.

4.4 Experiment 4

4.4.1 Question

Using pattern matching analysis, measure how the created classification models from experiment 2 compare to the current state of the art solutions.

4.4.2 Background

This experiment is dependent on the results from experiment 2. The concept is to measure how accurate the results are in experiment 2 when compared against the positive results identified by VirusTotal reports. While the cuckoo sandbox did report back the VirusTotal positive results, it was discovered that it did not include all anti-virus solutions in the results.

1. Get the MD5 hash for each sample in the test set from experiment 2.
2. Upload each MD5 using vtcheck VirusTotal API to get the positive score
3. Calculate percentage accuracy the based on the positive rate divided by the number of detection engines used.
4. Compare results for each family against the results from experiment 2.

4.4.3 Experiment

The MD5 hash of each sample from the test set is recorded in a text file containing only the hashes of that family of malware. So, in the example explained, the Hupigon.txt file contains the md5 hash for that family of malware that were part of the test set in experiment 2. The VirusTotal API vtcheck.py outputs the report to the terminal and only sends the final positive hit count to an output results file. The code of the API was edited to include the total number of online scanners in the results file. In order to use this API a VirusTotal API key must also be given at the command prompt.

```
python vtcheck.py -i ./Hupigon.txt -o result1.txt
-k 3c93eddea6dc39812558ee5bd0946d38791e9536c367544d159b716903dd2258
```

Figure 4.8 shows the hashes of the samples from the test set of the Hupigon malware family. All the results show positive malicious hits against the total number of anti-virus solutions that scanned the file.

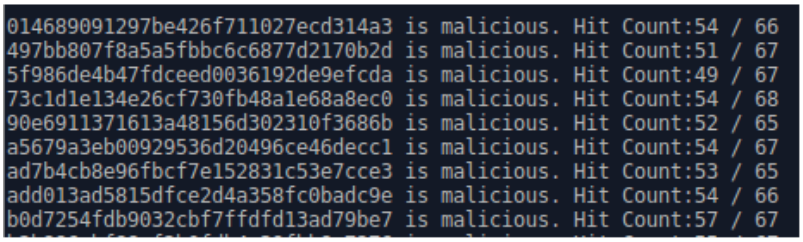


Figure 4.8: VirusTotal MD5 Results Check

The results are entered into the excel spreadsheet and an accuracy percentage is calculated. The accuracy is calculated by dividing the total number of positive by the total number of scanners. Each sample is calculated and then the average accuracy for a family is calculated. The following formula is used for the calculation.

$$Accuracy = \frac{Positve}{Total} x100$$

For a Hupigon variant with a positive hit of 54 out of 66 scanners that ran this is a accuracy rating of 81.8%. The average is then calculated for the family which results in 93.15%. The full list of results are in appendix B.

To calculate the accuracy from the test set executed in Weka with K-NN and SVM the following formula is used.

TP Rate = True Positive Rate

FP Rate = False Positive Rate

TN Rate = True Negative Rate

FN Rate = False Negative Rate

Accuracy is calculated using the formula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

So in an example taken from table 4.23, The CoinMiner family, a True Positive Rate is 0.625, then the True Negative is 0.375. The False Positive Rate is 0.014 so the False Negative Rate is 0.986.

In table 4.23 shows the breakdown of calculations for the K-nearest Neighbour where K = 3. The TP Rate and FP Rate results were taken from the given Weka output when the test set was executed against the training set. The TN Rate and FN Rate are calculated and then the accuracy percentage is calculated in order to later compare against the same sample accuracy calculation from SVM and from VirusTotal.

Malware Family	TP Rate	FP Rate	TN Rate	FN Rate	Accuracy
CoinMiner	0.625	0.014	0.986	0.375	80.55%
Hupigon	0.972	0.109	0.891	0.028	93.15%
Locky	0.583	0.014	0.986	0.417	78.22%
Mirai	1.000	0.000	1.000	0.000	100%
Ramnit	1.000	0.067	0.933	0.000	96.67%
Emotet	0.714	0.015	0.985	0.286	84.95%

Table 4.23: K-NN Accuracy Calculation

In table 4.24 shows the breakdown of calculations for Support Vector Machine. The TP Rate and FP Rate results was taken from the given Weka output. The TN Rate and FN Rate are calculated and then the accuracy percentage is calculated in order to later compare against the same samples accuracy calculation from K-NN and from VirusTotal.

Malware Family	TP Rate	FP Rate	TN Rate	FN Rate	Accuracy
CoinMiner	0.750	0.000	1.000	0.250	87.5%
Hupigon	1.000	0.043	0.957	0.000	97.85%
Locky	0.583	0.029	0.971	0.417	77.7%
Mirai	1.000	0.000	1.000	0.000	100%
Ramnit	1.000	0.027	0.973	0.000	98.87%
Emotet	0.857	0.033	0.967	0.143	90.65%

Table 4.24: SVM Accuracy Calculation

Table 4.25 shows a accuracy comparison of the pattern matching algorithms K-NN, SVM and a accuracy percentage calculated per family from VirusTotal.

Malware Family	K-NN	SVM	VirusTotal
CoinMiner	80.55%	87.5%	72.70%
Hupigon	93.15%	97.85%	80.12%
Locky	78.22%	77.7%	82.42%
Mirai	100%	100%	86.66%
Ramnit	96.67%	98.87%	90.12%
Emotet	84.95%	90.65%	78.07%

Table 4.25: Accuracy Comparison

4.4.4 Summary

The performance is good across all classes. The Support Vector Machine classification algorithm performs best overall. It does fall down when it comes to the heavily obfuscated Locky family where both K-NN and VirusTotal perform better. Both K-NN and SVM showed a high accuracy when detecting Hupigon, Mirai and Ramnit families. For long running malware families like Emotet and Locky where the variants have significantly changed obfuscation and functionality over time the accuracy remains very good. The VirusTotal online scanners are generally given the latest updates from the partners. The low scores may be impacted by poor performing online scanners with old signature or heuristic databases. So, this accuracy score does not reflect any one solution. Overall with this small data set it both K-NN and SVM outperforms the combination of anti-virus solutions. SVM has shown to be the more accurate of the two classification models.

Chapter 5

Discussion and Analysis

As stated in the research hypothesis, the current malware visualisation research concentrated on static, dynamic and reverse engineering of malware. These visualisation studies showed that heavily obfuscated malware variants are not easily identifiable. The research hypothesis of this study stated that by visualising malware in memory, it will facilitate features to be extracted and identified which can enable the identification of new malware variants. The question posed included, can new malware variants be identified using computer vision pattern identity after extracting malware from a memory dump? Does using visualisation of API trigger-based analysis improve pattern identity of malware? and How does state of the art solutions stand up to computer vision model-based analysis?

The results of this study demonstrated that obfuscated malware can be detected using visualisation in memory analysis. Machine learning pattern matching algorithms are currently used in heuristic engines and have shown in previous visualisation research to be of benefit when identifying malware using visualisation techniques. For obfuscated malware like the Mirai botnet variants, they showed the potential to be identified visually due to the similarity in the captured process memory dumps. The benefit to this would aid beginner to intermediate skilled malware analysts to identify a new variant of a malware family. The visualisation of code injection and de-obfuscation of a sample could also aid reverse engineers to reduce the time needed to identify key features in a sample.

In experiment 1 a comparison of visualisation in static analysis and visualisation of memory analysis was completed. Nataraj et al. research was completed on de-obfuscated samples or samples that used the same packer. Their results were positive for these types of malware. Unfortunately, malware authors are becoming more and more creative with their obfuscation techniques. Nataraj's approach on modern obfuscated malware clearly showed that static analysis on obfuscated samples is not viable. The Mirai botnet sample were clearly different when visualised statically. When the samples were executed in the cuckoo sandbox environment and the process memory dumps were captured and visualised it became evident that the now de obfuscated malware variants had similar code patterns.

With the Locky variants it became evident of malware authors developing obfuscation techniques over time. The earliest samples were completely de-obfuscated quickly and as new variants appeared different obfuscation techniques were used. The most recent samples used encryption and re-encryption as the process executed. This means that reverse engineering is hindered, and signature and heuristic detection become problematic. However, visualisation still detect the same new variants despite the encryption obfuscation techniques used as there was similarities in the visualisation. The shortcoming with the approach for heavily obfuscated malware like the most recent version of Locky, is that at least one variant must be detected first in order for the other variants to be detected using visualisation. The results showed visually how Locky variants has evolved obfuscation techniques over time. This has made detecting new malware variants extremely difficult.

In Experiment 2, pattern matching analysis using K-NN and SVM on process memory dumps showed good performance across all classes. The K-NN average True Positive rate was 84.15% while the SVM average True Positive rate was 89.02%. There was 100% accuracy for Mirai and Ramnit in both K-NN and SVM test results. However, the positive predictive value (precision) for Ramnit in K-NN was 58.3% and 77.8% in SVM. For SVM, even though it had a 100 True Positive rate the precision calculation showed a discrepancy. Upon reviewing the Weka classifier error visualiser, it showed a small margin of 1.13% percent discrepancy between a Ramnit variant and a Emotet Variant. So even though the sample

was correctly classified it is correct in highlighting the variance in precision as the Emotet sample could realistically be incorporated in its entirety in the Ramnit sample.

Hupigon despite being the largest test set had 97.2% True Positive rate for K-NN and 100% for SVM. CoinMiner showed an improved accuracy rate in SVM with 75% True Positive rate while in K-NN it was only 62.5%. Locky had the same True positive rate of 58.3% for both K-NN and SVM.

By reviewing the confusion matrix, it became easier to detect where the conflicting data points were. Both the K-NN and the SVM confusion matrix for process memory dumps showed that 4 out of the 11 test Locky samples were incorrectly classified. Upon closer inspection and using the Weka classifier error visualiser it showed that perhaps a larger data set of Locky samples would help improve the detection ratio as the false negative margin was at 13%. What the confusion matrix also showed was the similarity between some malware families. The SVM algorithm which performed better overall detected similarity between some Locky variants and an Emotet variant. The predictive margin was 4.7% and the samples visually had similar features. This can demonstrate similar code, obfuscation techniques and numerous other features that are shared amongst malware families. The results also show that the GIST global descriptor features are sufficiently descriptive for the purpose of classification of malware in memory. When using GIST features with pattern analysis, it demonstrated the ability to identify and group malware variants and identify malware families.

Experiment 3 was executed on a custom version 1.0 cuckoo sandbox. It had a much smaller data set when compared to the previous experiment. For the K-NN algorithm the correctly classified instance value was 79.2% while for SVM it showed an improved performance of 87.5%. The Trickster malware family proved to have the lowest correctly classified instance value at only 50% for K-NN and 66.7% for SVM. In the confusion matrix for K-NN it is clear that 2 Trickster variants are classified as Locky. Upon further investigation it was discovered that while there are some similar features there are not enough to state that the sample is should be classified as a Locky variant. This result is most probably caused by the small data set.

Experiment 4 measured how accurate the results from Experiment 2 are when compared to VirusTotal positive detection. This experiment was completed using a VirusTotal API to upload the MD5 hashes of each sample in a malware family and obtain a percentage accuracy score for each malware family. An average accuracy for each family was calculated. Both K-NN and SVM performed really well. Support Vector Machine showed to be the most accurate overall. It did fall down on the heavily obfuscated Locky variant but performed better than K-NN and VirusTotal on all other samples.

When comparing the results of K-NN and SVM performance across all experiments it is clear the SVM performs better. What the results show is that visualisations of memory analysis have the potential to aid malware detection and malware analysis. The comparison against the VirusTotal shows that the accuracy even with a small data set, is at least on a par or slightly better than some current anti-virus solutions. Along with other detection and analysis tools, this study has shown then visualisation of memory analysis can at the very least be used to aid malware analysis.

Chapter 6

Conclusion

Previous visualisation studies demonstrated that obfuscated malware variants are not easily identifiable. This study demonstrated that new obfuscated malware variants and malware families can be identified using visualisation of memory analysis. Obfuscated malware family types including botnets, banking, ransomware, coin miners and backdoors were identified in order to have a broad range of data samples to test against. Each sample was executed in a automated malware analysis sandbox. Process memory dumps were extracted during execution and then visualised into a grayscale image. Global GIST descriptors were calculated and then used to create training and test models. Finally, classification algorithms K-NN and SVM were executed against the models to calculate a

The study also that static visualisation samples proved insufficient when detecting obfuscated malware families. Visualising the malware variants from memory showed potential to visually be able to identify texture features. However, as this is time consuming and a manual task machine learning pattern analysis proved to aid this process of analysis. The results of this study confirmed that GIST global descriptor features are sufficient when classifying malware in memory using machine learning pattern analysis. Some obfuscation techniques like string obfuscation used by the Mirai botnet to overcome Yara rules, are bypassed as the malware sample is captured in memory where the sample is de-obfuscated. Locky proved to be the hardest to detect as it was the one family with huge obfuscation changes over time. It was however demonstrated that once one close variant of a newly obfuscation technique

within the Locky malware family was detected, then other variants could then be detected based on pattern analysis. The study was limited with a small data set but the results demonstrated to be large enough for a proof of concept study.

Support Vector Machine classification demonstrated to be the more accurate of the two algorithms tested in this study. It also showed a higher accuracy rating when comparing the same test sample set against the collection of VirusTotal online scanners. Overall this approach proved that visualisation of memory analysis has a potential to detect new malware variants and aid malware detection in existing variants. Visualisation can also aid malware analysis during reverse engineering or behavioural analysis.

6.1 Future Work

There is huge potential for future work in this area. By repeating this study but by using RGB coloured images allows for more feature detection than that of the limited view by using grayscale. Fu et al. (2018) had already completed the study using RGB on unpacked and decrypted samples but it proved to be weak against packed or encrypted malware. By capturing the malware process in memory and visualising using RGB will give the potential for thousand of features to be extracted. This may make the pattern analysis runtime longer but with large data sets it should make it more accurate. As already discussed, previous research was completed on manually de-obfuscated samples. Using memory analysis on multiple visualisation techniques like entropy graphs or entropy blocks could prove beneficial. Entropy graphs and Entropy blocks proved to be very accurate on de-obfuscated samples so improved results on this study is possible with this method.

Further development of the trigger-based API memory analysis is worth considering. For the experiment in this study the software was old and limited in functionality. As behaviour analysis of new malware variants includes detecting suspicious patterns, visual analysis could be automated to aid this analysis process. A cuckoo plugin for trigger-based API process memory dumps would be one way to get a visualisation of a full end to end execution

of a piece of malware. This has the potential to greatly aid malware analysist and reverse engineering samples.

CoinMiners have exploded onto the malware scene in the last few years. Web browsers are the target here where some JavaScript can be loaded into a user's browsing session on a website that there visiting and then used to mine a crypto currency. Java does not a fixed structure like the PE structure. JavaScript still needs to be de-obfuscated in memory in order to execute. As a result, a visualised sample can be taken and used for pattern analysis. As shown with the PE format, this has its limitations but once a pattern is discovered then variants can be identified.

Yara signature scans can be bypassed by many malware samples. However, Yara scans in memory as malware executes have shown to have improved accuracy. Incorporating Yara into scanning for a percentage of known extracted GIST features could have the potential for a live visualisation memory analysis program.

Bibliography

Aghaeikheirabady, M., Farshchi, S. M. R., Mashhad, I. and Shirazi, H. (2014), 'A new approach to malware detection by comparative analysis of data structures in a memory image', *2014 International Congress on Technology, Communication and Knowledge (ICTCK), Technology, Communication and Knowledge (ICTCK), 2014 International Congress on* p. 1.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.7033519&site=eds-live&custid=ns122646*

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L. and Kallitsis, M. (2017), Understanding the mirai botnet, *in* 'USENIX Security Symposium', pp. 1092–1110.

Aslan, O. and Samit, R. (2017), 'Investigation of possibilities to detect malware using existing tools', *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), Computer Systems and Applications (AICCSA), 2017 IEEE/ACS 14th International Conference on, AICCSA* p. 1277.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.8308437&site=eds-live&custid=ns122646*

Bazrafshan, Z., Hashemi, H., Fard, S. M. H. and Hamzeh, A. (2013), 'A survey on heuristic malware detection techniques', *The 5th Conference on Information and Knowledge Technology, Information and Knowledge Technology (IKT), 2013 5th Conference on* p. 113.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edseee&AN=edseee.6620049&site=eds-live*

Cabaj, K., Gawkowski, P., Grochowski, K., Nowikowski, A. and Zórawski, P. . (2017), 'The

impact of malware evolution on the analysis methods and infrastructure', *2017 Federated Conference on Computer Science and Information Systems (FedCSIS), Computer Science and Information Systems (FedCSIS), 2017 Federated Conference on* p. 549.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee &AN=edseee .8104598&site=eds-live&custid=ns122646*

CARO (1991).
URL: *http://www.caro.org/articles/naming.html*

Cleary, G., Corpin, M., Cox, O., Lau, H., Nahorney, B., O'Brien, D., O'Gorman, B., Power, J.-P., Wallace, S., Wood, P. and Wueest, C. (2018), Internet security threat report: Volume 23, Technical report, Symantec.
URL: *https://resource.elq.symantec.com/e/f2*

Comodo (2017), 'Global malware report 2017'.
URL: *https://www.comodo.com/ctrlquarterlyreport/2017summary/Comodo_2017Report.pdf*

Conti, G., Dean, E., Sinda, M. and Sangster, B. (2008), *Visual reverse engineering of binary and data files*, Visualization for Computer Security, Springer, pp. 1–17.

Donahue, J., Paturi, A. and Mukkamala, S. (2013), 'Visualization techniques for efficient malware detection', *2013 IEEE International Conference on Intelligence and Security Informatics, Intelligence and Security Informatics (ISI), 2013 IEEE International Conference on* p. 289.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee &AN=edseee.6578845&site=eds-live&custid=ns122646*

Fu, J., Xue, J., Yong, W., Liu, Z. and Shan, C. (2018), 'Malware visualization for fine-grained classification', *IEEE Access, Access, IEEE* p. 14510.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee &AN=edseee.8290767&site=eds-live&custid=ns122646*

Han, K., Kang, B. and Eul, G. I. (2014), 'Malware analysis using visualized image matrices', *The Scientific World Journal, Vol 2014 (2014)* .

URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edsdoj&AN=edsdoj.0d278388d1d6415085dc10ee00f8d3ce&site=eds-live&custid=ns122646*

Han, K., Lim, J., Kang, B. and Im, E. (2015), 'Malware analysis using visualized images and entropy graphs', *International Journal of Information Security* **14**(1), 1–14.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=bth&AN=100351864&site=eds-live&custid=ns122646*

Hashemi, H. and Hamzeh, A. (2018), 'Visual malware detection using local malicious pattern', *Journal of Computer Virology and Hacking Techniques* pp. 1–14.

Kancherla, K. and Mukkamala, S. (2013), 'Image visualisation based malware detection', *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), Computational Intelligence in Cyber Security (CICS), 2013 IEEE Symposium on* p. 40.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.6597204&site=eds-live&custid=ns122646*

Kaspersky (2018).
URL: *https://encyclopedia.kaspersky.com/knowledge/rules-for-naming/*

Lee, D., Song, I. S., Kim, K. J. and hyeon Jeong, J. (2011), 'A study on malicious codes pattern analysis using visualization', *2011 International Conference on Information Science and Applications, Information Science and Applications (ICISA), 2011 International Conference on* p. 1.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.5772330&site=eds-live&custid=ns122646*

Makandar, A. and Patrot, A. (2017), 'Malware class recognition using image processing techniques', *2017 International Conference on Data Management, Analytics and Innovation (ICDMAI), Data Management, Analytics and Innovation (ICDMAI), 2017 International Conference on* p. 76.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.8073489&site=eds-live&custid=ns122646*

Malekal (2017), 'Malware list - malekal.com'.
URL: *http://malwaredb.malekal.com/*

MalWerewolf (2015), 'Virus total lookup script'.
URL: *https://github.com/MalWerewolf/vtlookup*

Manavi, F. and Hamzeh, A. (2017), 'A new method for malware detection using opcode visualization', *2017 Artificial Intelligence and Signal Processing Conference (AISP), Artificial Intelligence and Signal Processing Conference (AISP), 2017* p. 96.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.8324117&site=eds-live&custid=ns122646*

McAfee (2017), 'Mcafee threat report 2017'.
URL: *https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf*

McAfee (2018), 'Mcafee labs threat report june 2018'.
URL: *https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2018.pdf*

Mosli, R., Li, R., Yuan, B. and Yin, P. (2016), 'Automated malware detection using artifacts in forensic memory images', *2016 IEEE Symposium on Technologies for Homeland Security (HST), Technologies for Homeland Security (HST), 2016 IEEE Symposium on* p. 1.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.7568881&site=eds-live&custid=ns122646*

Nataraj, L., Karthikeyan, S., Jacob, G. and Manjunath, B. S. (2011), 'Malware images: Visualization and automatic classification'.
URL: *https://www.researchgate.net/publication/228811247*

Nrupatunga (2016), 'Gist-global-image-descriptor'.
URL: *https://github.com/nrupatunga/GIST-global-Image-Descripor*

Oliva, A. and Torralba, A. (2001*a*), 'Gist descriptor (matlab code)'.
URL: *http://people.csail.mit.edu/torralba/code/spatialenvelope/*

Oliva, A. and Torralba, A. (2001*b*), 'Modeling the shape of the scene: A holistic representation of the spatial envelope', *International journal of computer vision* **42**(3), 145–175.

Pandey, S. K. and Mehtre, B. M. (2014), 'Performance of malware detection tools: A comparison', *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on* p. 1811.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.7019422&site=eds-live&custid=ns122646*

Patten, D. (2017), 'The evolution to fileless malware', *Retrieved from* .
URL: *http://www.infosecwriters.com/Papers/DPatten_Fileless.pdf*

Pontiroli, S. M. and Martinez, F. R. (2015), The tao of .net and powershell malware analysis, *in* 'Virus Bulletin Conference'. URL: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07202147/Pontiroli_Martinez-VB2015-2.pdf.

Quist, D. A. and Liebrock, L. M. (2009), 'Visualizing compiled executables for malware analysis', *2009 6th International Workshop on Visualization for Cyber Security, Visualization for Cyber Security, 2009.VizSec 2009.6th International Workshop on* p. 27.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edseee&AN=edseee.5375539&site=eds-live*

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B. and Nicholas, C. (2017), 'Malware detection by eating a whole exe'. TY: GEN; ID: Accession Number: edsarx.1710.09435; Accession Number: edsarx.1710.09435; Publication Type: Working Paper; Publication Date: 20171025.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edsarx&AN=edsarx.1710.09435&site=eds-live*

Rathnayaka, M. C. and Jamdagni, D. A. (2017), 'An efficient approach for advanced malware analysis using memory forensic technique', *2017 IEEE Trustcom/BigDataSE/ICESS, Trustcom/BigDataSE/ICESS, 2017 IEEE, TRUSTCOM-BIGDATASE-ICESS* p. 1145.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edseee&AN=edseee.8029568&site=eds-live*

Ren, Z. and Chen, G. (2017), 'Entropyvis: Malware classification', *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), 2017 10th International Congress on* p. 1.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edseee&AN=edseee.8302000&site=eds-live&custid=ns122646*

Said, N. B., Biondi, F., Bontchev, V., Decourbe, O., Given-Wilson, T., Legay, A. and Quilbeuf, J. (2017), 'Detection of mirai by syntactic and semantic analysis'.

Sandbox, C. (2018), 'Cuckoo malware analysis sandbox'.
URL: *https://cuckoosandbox.org/*

Shaid, S. Z. M. and Maarof, M. A. (2014), 'Malware behaviour visualization', *Journal Tknologi, Eissn 2180–3722* pp. 25–33.
URL: *https://www.researchgate.net/publication/273311319_Malware_Behaviour_Visualization*

Su, J., Vargas, D. V., Prasad, S., Sgandurra, D., Feng, Y. and Sakurai, K. (2018), 'Lightweight classification of iot malware based on image recognition'. TY: GEN; ID: Accession Number: edsarx.1802.03714; Accession Number: edsarx.1802.03714; Publication Type: Working Paper; Publication Date: 20180211.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edsarx&AN=edsarx.1802.03714&site=eds-live*

Symantec (2015), 'Security response: W32.ramnit analysis'.
URL: *https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/w32-ramnit-analysis-15-en.pdf*

Teller, T. (2014), 'Automated malware memory analysis'.
URL: *https://github.com/djteller/MemoryAnalysis*

Teller, T. and Hayon, A. (2014), 'Enhancing automated malware analysis machines with memory analysis '.

URL: *https://www.blackhat.com/docs/us-14/materials/arsenal/us-14-Teller-Automated-Memory-Analysis-WP.pdf*

Trinius, P., Holzy, T., Gobel, J. and Freiling, F. C. (2009), 'Visual analysis of malware behavior using treemaps and thread graphs', *2009 6th International Workshop on Visualization for Cyber Security, Visualization for Cyber Security, 2009.VizSec 2009.6th International Workshop on* p. 33.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edseee&AN=edseee.5375540&site=eds-live*

Venkatraman, D. S. and Alazab, D. M. (2017), 'Classification of malware using visualisation of similarity matrices', *2017 Cybersecurity and Cyberforensics Conference (CCC), Cybersecurity and Cyberforensics Conference (CCC), 2017, CCC* p. 3.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edseee&AN=edseee.8252893&site=eds-live*

VirusTotal (2018).
URL: *https://www.virustotal.com*

Volatility (2018), 'Volatility foundation'.
URL: *https://www.volatilityfoundation.org/*

Wagener, G. and Dulaunoy, A. (2008), 'Malware behaviour analysis', *Journal in computer virology* **4**(4), 279–287.

Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D. A., Aigner, W., Borgo, R., Ganovelli, F. and Viola, I. (2015), A survey of visualization systems for malware analysis, *in* 'EG Conference on Visualization (EuroVis)-STARs', pp. 105–125.

WEKA (2018).
URL: *https://www.cs.waikato.ac.nz/ml/weka/*

Xen0ph0n (2015), 'Virustotal_api_tool'.
URL: *https://github.com/Xen0ph0n/VirusTotal_API_Tool*

Xia, Y., Fairbanks, K. and Owen, H. (2008), *Visual analysis of program flow data with data propagation*, Visualization for Computer Security, Springer, pp. 26–35.

Ye, Y., Li, T., Adjeroh, D. and Iyengar, S. S. (2018), 'A survey on malware detection using data mining techniques', *ACM Computing Surveys* **1**(3), 41.
URL: *http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=edsgao&AN=edsgcl.543780151&site=eds-live&custid=ns122646*

You, I. and Yim, K. (2010), Malware obfuscation techniques: A brief survey, *in* 'Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on', IEEE, pp. 297–300.

ytisf (2018), 'The zoo malware db'.
URL: *https://github.com/ytisf/theZoo*

Appendices

A Full Code of VirusTotal API vtcheck

```
"""
--------------------------------------------------------------------------------
Copyright 2015 Destruct_Icon
   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at
       http://www.apache.org/licenses/LICENSE-2.0
   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
--------------------------------------------------------------------------------
http://www.malwerewolf.com/
Author: Destruct_Icon, nanoSpl0it
Version: 1.0
Summary: Queries Virus Total for all reports of the hashes provided.
"""
import requests
import argparse
import os
import time

def checkkey(kee):
        try:
                if len(kee) == 64:
                        return kee
                else:
                        print "There is something wrong with your key. Not 64 Alpha Numeric characters."
                        exit()
        except Exception, e:
                        print e

def checkhash(hsh):
        try:
                if len(hsh) == 32:
                        return hsh
                elif len() == 40:
                        return hsh
                elif len(hsh) == 64:
                        return hsh
                else:
                        print "The Hash input does not appear valid."
                        exit()
        except Exception, e:
                        print e
```

```
def fileexists(filepath):
        try:
                if os.path.isfile(filepath):
                        return filepath
                else:
                        print "There is no file at:" + filepath
                        exit()
        except Exception, e:
                        print e

def main():
        parser = argparse.ArgumentParser(description="Query hashes against Virus Total.")
        parser.add_argument('-i', '--input', type=fileexists, required=False, help='Input File Location EX:
        parser.add_argument('-o', '--output', required=True, help='Output File Location EX: /Desktop/Somewh
        parser.add_argument('-H', '--hash', type=checkhash, required=False, help='Single Hash EX: d41d8cd98
        parser.add_argument('-k', '--key', type=checkkey, required=True, help='VT API Key EX: ASDFADSFDSFASI
        parser.add_argument('-u', '--unlimited', action='store_const', const=1, required=False, help='Chang
        args = parser.parse_args()

        #Run for a single hash + key
        if args.hash and args.key:
                file = open(args.output,'w+')
                file.write('Below is the identified malicious file.\n\n')
                file.close()
                VT_Request(args.key, args.hash.rstrip(), args.output)
        #Run for an input file + key
        elif args.input and args.key:
                file = open(args.output,'w+')
                file.write('Below are the identified malicious files.\n\n')
                file.close()
                with open(args.input) as o:
                        for line in o.readlines():
                                VT_Request(args.key, line.rstrip(), args.output)
                                if args.unlimited == 1:
                                        time.sleep(1)
                                else:
                                        time.sleep(26)
```

```
def VT_Request(key, hash, output):
        params = {'apikey': key, 'resource': hash}
        url = requests.get('https://www.virustotal.com/vtapi/v2/file/report', params=params)
        json_response = url.json()
        print json_response
        response = int(json_response.get('response_code'))
        if response == 0:
                print hash + ' is not in Virus Total'
                file = open(output,'a')
                file.write(hash + ' is not in Virus Total')
                file.write('\n')
                file.close()
        elif response == 1:
                positives = int(json_response.get('positives'))
#Edit: added line   "total = int(json_response.get('total'))"
                total = int(json_response.get('total'))
                if positives == 0:
                        print hash + ' is not malicious'
                        file = open(output,'a')
                        file.write(hash + ' is not malicious')
                        file.write('\n')
                        file.close()
                else:
                        print hash + ' is malicious'
                        file = open(output,'a')
#Edit: Included total string at the end
                        file.write(hash + ' is malicious. Hit Count:' + str(positives) + ' / ' + str(total)
                        file.write('\n')
                        file.close()
        else:
                print hash + ' could not be searched. Please try again later.'
# execute the program
if __name__ == '__main__':
        main()
```

B VirusTotal Hash check results

Class	MD5	Positive	Total	Percentage
CoinMiner	143830249aa144318e2d2237bd2815ff	51	68	75.00%
CoinMiner	a3f66f70643c692893bd1f84f67183a4	49	68	72.06%
CoinMiner	a4ed97a732bb0bd9671c0093322ab4e2	50	68	73.53%
CoinMiner	a5c9234f0811fe23c42e9d3b4e318f98	49	68	72.06%
CoinMiner	a6301bff7e66044f149653df5b50cfc8	51	67	76.12%
CoinMiner	b3688c24e3e41001965eab636e90900f	51	68	75.00%
CoinMiner	e55e3b9593d2d0f7c6ec678c0e5217dc	43	66	65.15%

CoinMiner	Average			72.70%
Emotet	07dc32112fe5ec562a29079db09469cb	51	64	79.69%
Emotet	16f514f1bcfbff2a895a5740d2bf8b6c	47	57	82.46%
Emotet	37aea76dbe9448430032438710b2d5d0	44	55	80.00%
Emotet	3bae3d4487825298d6de471ea2f56974	57	68	83.82%
Emotet	46a4d1884a770a63bda619613e0701e1	49	67	73.13%
Emotet	46cae9e115be7db19983bd7bb5532963	42	55	76.36%
Emotet	69462df559efa0d5d45b481d804f9d62	41	55	74.55%
Emotet	afcc2e54a18cea16912069d2c75716b7	43	57	75.44%
Emotet	b5d004945ac73c941b4c0313012471de	44	57	77.19%
Emotet	Average			78.07%
Hupigon	014689091297be426f711027ecd314a3	54	66	81.82%
Hupigon	497bb807f8a5a5fbbc6c6877d2170b2d	51	67	76.12%
Hupigon	5f986de4b47fdceed0036192de9efcda	49	67	73.13%
Hupigon	73c1d1e134e26cf730fb48a1e68a8ec0	54	68	79.41%
Hupigon	90e6911371613a48156d302310f3686b	52	65	80.00%
Hupigon	a5679a3eb00929536d20496ce46decc1	54	67	80.60%
Hupigon	ad7b4cb8e96fbcf7e152831c53e7cce3	53	65	81.54%
Hupigon	add013ad5815dfce2d4a358fc0badc9e	54	66	81.82%
Hupigon	b0d7254fdb9032cbf7ffdfd13ad79be7	57	67	85.07%
Hupigon	b2b886ebf88cf2b0fdb4a39fbb8a7276	55	67	82.09%
Hupigon	b7e153cae080279b33751397d75f1217	55	67	82.09%
Hupigon	bd29346f12dd65a5e1a1c5b1fa2c76eb	53	67	79.10%
Hupigon	dd5d93f282223f3114fce703d8e55212	52	66	78.79%
Hupigon	Average			80.12%
Locky	0265f31968e56500218d87b3a97fa5d5	52	68	76.47%
Locky	0bd30fca55a734b29218d45d7dab1a04	60	68	88.24%
Locky	0c0d798add0c4e450a1c8bf3824ea989	56	67	83.58%
Locky	14eba698c1dedfee512156111ec6ba3b	59	68	86.76%
Locky	1ff4f4d4bff47b30d585dcb44fd0a479	54	67	80.60%
Locky	31d2bdcd2fc117b558b54e731af02a65	56	67	83.58%

Locky	370546a376345688e5ad3ef836e2ea78	56	68	82.35%
Locky	3db1e36f7d4dcf040e60114e8c43fea5	50	69	72.46%
Locky	4a4eee2416f0cb4ac0de4a658ec168ba	55	68	80.88%
Locky	4f03e360be488a3811d40c113292bc01	55	67	82.09%
Locky	916faa4753e602cd83ccc6958d00da60	54	68	79.41%
Locky	aba2d86ed17f587eb6d57e6c75f64f05	63	68	92.65%
Locky	Average			82.42%
Mirai	a06b6c71e8852c710cbfc71255f43018	60	67	89.55%
Mirai	a4227486fcacbe561f0f5a8a5f7847b6	58	68	85.29%
Mirai	a620e085dd01e11f31889a49dff028c2	59	66	89.39%
Mirai	b4151f69192845f047b61aced0e0dda0	50	63	79.37%
Mirai	ba7ef832f3ffe418176e384dfe2f035f	61	68	89.71%
Mirai	Average			86.66%
Ramnit	704aff508a74f2ddb27a4d61322df5d7	59	66	89.39%
Ramnit	76e1b01e950154773e09fd8cee739d04	61	67	91.04%
Ramnit	b0e2dde3393ba667e3bb266a449bf2f3	59	67	88.06%
Ramnit	d80be38df6f84734e90d5e3bf125217d	61	67	91.04%
Ramnit	fb49af392cc70f830b1fea3559e6adc8	61	67	91.04%
Ramnit	Average			90.12%