KING'S College LONDON

University of London

Deep Learning Image-based Malware Detection and Adversarial Variants

Final Project Report

BSc Computer Science with a Year Abroad

Author: Jordan Joe Watson

Supervisor: Dr Jose M. Such

Student ID: 1616076

April 23, 2020

Abstract

In recent years Deep Learning methods have been applied to Malware Detection by representing files as images. Additionally, there has been research into creating Adversarial Variants of Malware to bypass these Deep Learning Detection methods. The aim of this report is to (1) demonstrate two new methods for Deep Learning Image-based Detection of Malware, (2) create Adversarial Variants of Malware using File Infection techniques and finally (3) to propose a new method for Deep Learning Image-based Detection of Malware that will achieve a high accuracy whilst also being invulnerable to File Infection attacks.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Jordan Joe Watson

April 23, 2020

Acknowledgements

I would like to thank my Supervisor and Tutor, Dr. Jose M. Such for his guidance and support throughout my Degree, this research project, and for advising me in Adversarial Variants. I would also like to thank Professor Lorenzo Cavallaro for his help in understanding the PE and ELF file structure and Adrian Salazar Gomez for his help in Image Classification using different Convolutional Neural Networks structures. Thank you to Virus Total [1] for providing me with the Malware dataset used for this project. The author acknowledges use of the research computing facility at King's College London, Rosalind (https://rosalind.kcl.ac.uk). Thank you to my family.

Contents

Chapter 1

Introduction

With the recent and rapid advancement of technology, Computer Systems are now integrated into every part of daily life. Alongside this, the rise of Malware - malicious Software - has also risen dramatically. There are now approximately 350,000 new malicious files being identified every day, this has increased from 65 Million to over 1 Billion in the past decade [2]. Although Detection methods have been researched and implemented, Malware is constantly being refined to find new ways of thwarting Detection methods. The Detection methods therefore need to be constantly refined to try and stay one step ahead of the offensive techniques, continuing the continuous cat and mouse game.

Previous approaches to Malware Detection have involved Signature-based, Behaviour-based, Heuristic-based and Model Checking-based methods of Detection. Recently, newer methods have been proposed including Deep Learning-based Detection. In the area of Deep Learning-based detection, researchers have been demonstrating ways in which Malware Detection can be approached by representing files as images and analysing them using Machine Learning methods. Following on from this, researchers have been looking into how these Malware Detectors can be tricked by creating Adversarial Variants of the Malware.

At the beginning of this year, an IEEE report claimed that Deep Learning-based Malware Detection is not resistant to obfuscation [3]. This report is a study into new methods of Deep Learning-based Malware Detection and Adversarial Variants using Image Representations of files, that ultimately aims to test this claim.

1.1 Report Aims

The aim of this report is to firstly present two new methods for Malware Detection using images that is an improvement on previous approaches. Secondly to demonstrate that these classifiers are invulnerable to previous approaches of Adversarial Variants. Furthermore, to create Adversarial Variants that are succesful against one of the classifiers and demonstrate that the other classifier is invulnerable to a these attacks. Finally, to propose a new method for Malware Detection that will achieve a high accuracy whilst also being invulnerable to the Adversarial Variants discussed in this report.

1.2 Report Structure

As the aim of this report is to demonstrate new methods for Malware Detection, Adversarial Variants and a new Proposal for Malware Detection, the main body of this report will be broken down into multiple sections.

The first section will lay out the fundamental details, background and context required to fully understand the research. This includes defintions, review of related works, basic explanation of the PE32 File Format and the Axioms that the following research is built upon.

The second section will discuss the dataset used, including a discussion on different datasets, how the dataset was cleaned, Data Preparation of converting Malware to images and finally an analysis of the dataset.

Next, the third section introduces the first main aspect of the research. This section looks into Detection of Malware using Convolutional Neural Networks, Convolutional Neural Network architecture and two new methods of Malware Detection with an evaluation of the successes and failures of each approach.

After demonstrating new methods of Malware Detection, the next section discusses how to create Adversarial Variants using File Infection methods for these Classifiers to evaluate the successes and failures of the new approaches.

Finally, after demonstrating the new methods of Detection and Adversarial Variants, a proposal is given for a Malware Detection method that aims to achieve high accuracy whilst being invulnerable to the Adversarial Variants demonstrated in the previous section.

Chapter 2

Background

2.1 Definitions

Malware 1 *Software that is designed to cause damage or gain unauthorized access to a Computer System.*

Goodware 1 *Software that does not cause damage and has no malicious intent.*

Malware Detection 1 *The act of deciding if Software is malign or benign.*

Malware Classification 1 *The act of categorizing Malware into a Malware family/type.*

Adversarial Variant 1 *Inputs to a Machine Learning Model that aims to cause the Classifier to output an incorrect classification.*

2.2 Related Work

Due to this report studying aspects of both Malware Detection and Adversarial Variants, the related work of these areas are discussed seperately. Some of the related work is for Malware Classification, only the results obtained in Detection research is discussed as Malware Classification results are not comparable, however the methods and ideas used are still relevant.

2.2.1 Malware Detection and Classification

Nataraj et al. [4] researched methods for classifying Malware into different families using K-Nearest Neighbors. The dataset was collected from the University of California Santa Barbara Vision Research Lab [5]. The Malware files were read as a vector of 8 bit unsigned integers

and organised into a 2D array, this can then be visualised as a greyscale image. The width is determined by the file size and the height varies depending on the file size and width. GIST Features are extracted from the images for classification.

Kancherla and Mukkamala [6] achieved a 95% Detection Accuracy using Support Vector Machines and greyscale image representations of Malware for Malware Detection. The Goodware dataset was collected from the Windows XP, Vista, 7 and NT Operating Systems as well as Sourceforge. The Malware was collected from Offensive Computing. The dataset contained 12,000 Goodware samples and 25,000 Malware Samples. The files were then converted to images using similar methods to [4] and the images are created using a fixed width depending on the size of the file. Gabor Wavelet based features are extracted and used as features.

Saxe and Berlin of Invincea Labs [7] achieved a 95% Detection Rate using a feed forward Neural Network and a dataset taken from Invincea's own database containing 431,926 binaries, using features mostly extracted from Metadata.

Makandar and Patrot [8] used Support Vector Machines and K-Nearest Neighbors for Malware Classification. The dataset was taken from the Malimg dataset [9]. Files were converted to greyscale images using methods previously mentioned, with the difference that they were resized to 64x64 images. Features were extracted using Gabor wavelet, GIST and Discrete Wavelet Transformations.

Zhu et al. [10] created a new approach of Malware Detection for Android Malware called DeepFlow. They were able to achieve an F1 score of 95.05% on 3000 Benign files and 8000 Malign files taken from the Google Play Store. Features were extracted from APK files using taint analysis with FlowDroid.

Su, Vargas and Prasad [11] used Convolutional Neural Networks inspired by ImageNet and VGG for Malware Detection. The Malware dataset used was taken from IoTPOT containing 500 Malware samples and the Goodware from Ubuntu 16.03, the files were converted to images by reading a byte and writing it as a greyscale pixel before resizing the images to 64x64. They achieved as accuracy of 94%.

Ye et al. [12] created DeepAM, a Malware Detection system, using AutoEncoders, Restricted Boltzmann Machines, using API Features extracted from Windows PE files. The dataset was from Comodo Cloud Security Centre of 4500 Benign and 4500 Malign files achieving a 98.2% Accuracy.

Singh et al. [13] researched Malware Classification using Images with a dataset of over 60,000 Malware Samples taken from VirusTotal [1], VirusShare and MalShare. Differing from

previous approaches, RGB representations were used. They approached this in similar ways as previous greyscale method by considering a byte as a pixel, however they used the upper and lower nibbles of a byte as indeces into a 2D colour map to use as the RGB colour. The files are fixed to a width of 384 bytes and the length of the files determines the height. After converting the files to images, they were resized but to 32x32 images instead of 64x64 as others have done [11]. They experimented with Convolutional Networks and used a ResNet-50 for the best results.

He and Kim [14] experiment with different approaches to Malware Detection using Images and Deep Learning. The dataset used was from a Korea University study. They use typical greyscale approaches as previously defined as well as a new method for converting to RGB images that involves using three consequitive bytes to represent an RGB value. The width is fixed to 1920 bytes, and to ensure that the images work with a fixed input they use a Spatial Pyramid Pooling layer with a ResNet-50.

Khormali et al. [15] used a Convolutional Neural Network with Goodware dataset of 1000 files taken from the Windows 10 Operating System and Malware dataset of 10,678 files taken from the Maligm [9] Dataset achieving a 99.12% Accuracy. The files were converted to greyscale images similar to previous approaches, using a fixed width and rescaling the images.

As all of the results obtained from different studies on Malware Detection use different Classifiers, different Datasets and different approaches to preparing data it makes it very difficult to draw conclusions between the results. Therefore the benefits and drawbacks of Datasets, preparing data and Classifiers will be discussed in Chapters 3 and 4.

2.2.2 Adversarial Variants

Szegedy et al. [16] showed that imperceptibly small perturbations of images can cause a Classifier to misclassify the image. Goodfellow et al. [17] continued on with this idea and proposed the Fast Gradient Sign Method (FGSM), which influenced many approaches of Adversarial Variants from hereon.

Papernot et al. [18] discuss different approaches to creating Adversarial Variants such as Confidence Reduction, Misclassification, Targeted Misclassification and Source/Target Misclassification.

Papernot et al. [19] presented a Mathematical Formalization for an Oracle for Adversarial Variants with the premise that a minimally altered version of any input should be made for creating Adversarial Variants and created Adversarial Variants using a Substitute Network.

Anderson [20] proposed multiple methods in which Adversarial Variants for Malware could be created. This includes adding functions to the Import Address Table, manipulating section names, creating new unused sections, appending bytes to the end of sections, creating a new entry point, manipulate signature, manipulate debugging information, packing/unpacking files, modifying the header checksum and appending bytes to the overlay.

Xu, Qi and Evans [21] used Genetic Programming techniques to perform Stochastic Manipulations to create Adversarial Variants. This was applied to PDF Malware, using PDFrate and Hidost as Classifiers. They used Cuckoo Sandbox as an Oracle to ensure the behaviour remains the same before and after the modification.

Dang, Huang and Chang [22] researched creating Adversarial Variants of PDF Malware by Morphing the Malware by inserting, deleting or replacing objects and using PDFrate and Hidost as Classifiers and Cuckoo Sandbox to ensure the behaviour remains the same. They also described a method for creating Adversarial Variants that includes a Morpher M that either inserts, deletes or replaces objects in the PDF file and a Tester T is used to check if the file maintains its malicious property, e.g. Cuckoo Sandbox.

Some potential issues with [21] and [22] are that by using Cuckoo Sandbox to ensure the Malicious property remains, it can only check if behaviour is the same and not the successes or failures. For example a system call may be called with an incorrect argument, or shellcode may be modified but the system call remains the same causing the Malware not to execute correctly but to still have the same behaviour. However as mentioned in [21], if there was a solution to this problem then Malware Detection would not be an area of further study. Cuckoo Sandbox is also computationally expensive and requires Dynamic Analysis.

Hu and Tan [23] propose a generative Adversarial Network. They create Adversarial Variants by representing the file as a list of 160 System Level API calls and create Variants by adding API calls to the vector. They don't remove API calls as that could damage the file. The limitations of this is that the features need to be extracted dynamically and the Adversarial Variants are a theoretical representation instead of actually altering the Malware. The approaches discussed in this report don't involve Dynamic Analysis, however the results from adding System Calls to a feature array to create Adversarial Variants could be utilized to discover what code could be used to infect a binary for creating Adversarial Variants.

Al-Dujaili et al. [24] create Adversarial Variants inspired by FGSM method described previously in [17]. The problem with this is that it doesn't take into account the changes to the underlying PE File Format and results in a corrupted file. The fundamental issue is that the

Malware Adversarial Variants are created in the same way as Image Adversarial Variants, this flaw will be discussed in this report.

Anderson et al. [25] defined three categories for attacking Machine Learning Classifiers for Malware. (1) Direct Gradient-based Attacks in which the model must be fully differentiable and the structure and weights must be known by the attacker. This allows the attacker to query the model directly to gain insight into how to bypass it. (2) White-box attacks against models that report a maliciousness score. The attacker has no knowledge about the model structure, but has unlimited access to probe the model and may be able to use heuristics that take advantage of the revealed scores to seach for evasive variants. (3) Binary black-box attacks. The attacker has no knowledge about the model, but has unlimited access to probe the model. The output is binary, either 1 (malign) or 0 (benign).

Kreuk et al. [26] created Adversarial Variants by adding small modifications to binaries by adding code in unused sections of a file or appending bytes to the end of a file. A potential limitation here is that they don't discuss any alteration of the PE File Format apart from appending/inserting data, so these approaches may not work when using metadata from the PE File to decide where to extract data from.

Kolosnjaji et al. [27] proposes an idea for appending a few bytes at the end of the file to ensure the behaviour of the file doesn't change. One issue with this approach is that they use a fixed feature length, and add bytes to the end of the file to create Adversarial Variants. If a file is equal or larger than the length of features then this attack may not be useful.

Liu et al. [28] uses Convolutional Neural Networks, Support Vector Machines and Random Forest to test the effectiveness of their Adversarial Variants. They develop a novel Adversarial Texture Malware Preturbation Attack (ATMPA) to create Adversarial Variants using FGSM [17] and a C&W attack-based method [29]. The dataset is taken from the Malimg [9] Malware dataset and Goodware from the Windows XP and 7 Operating Systems. The limitation of this work is that it approaches Malware Adversarial Variants from the perspective of Image Adversarial Variants and the Adversarial Variants don't maintain their executable nature and are therefore corrupted.

Khormali et al. [15] proposed an approach to generate Adversarial Variants called COPYCAT that uses two approaches. (1) AE padding and (2) Sample Injection. AE Padding works by appending an Adversarial Variant Image to the end of the original file image, this doesn't represent a real change to the Malware and wouldn't be applicable in a real scenario, also studying a file through PE metadata may also make this attack useless as will be discussed in

this report. Sample Injection works by injecting a sample from the targeted class into an unreachable area of the new binary. This will ensure executibility of the original sample. However changes to the PE File Format were not explicitly mentioned this method could potentially be invalid like previous ideas. Other approaches to creating Adversarial Variants similar to [18] were also presented, such as Confidence Reduction, Untargeted Misclassification and Targeted Misclassification.

There has been much research into creating Adversarial Variants for Malware Detection over the past decade, from describing different approaches, formalizations on how to create succesful Variants and many unsuccesful attempts. Because of this, some have claimed that Deep Learning-based approaches are not resistant to Obfuscation [3]. Chapter 5 will examine these approaches in greater detail to summarize the successes and failures of previous approaches and finally how to succesfully create Adversarial Variants as well as creating succesful Variants.

2.3 Prerequisites

The following is a basic overview of the PE File Format, vulnerabilities within the PE File Format and Axioms that this report is built upon.

2.3.1 PE File Structure

The research presented in this paper is based on Software for the Windows x86 Operating System. Therefore the dataset is limited to x86 PE (Portable Executable) Files [30].

The PE File Format is made up of multiple sections. The DOS Header, DOS Stub, COFF Header, Optional Header, Section Table and Sections.

The important information required for this report is that the sections hold the information used in the executable, e.g. a *.text* section contains the code for the application, the *.data* section holds the initialized data, *.idata* is used for imports used by the file. The sections present in a PE file are defined in the Section Table, each section has its own Section Header, each of size 40 bytes. The Section Header includes information such as the name of the section, e.g. *.text*, the size of the section and virtual address. The Optional Header contains the *FileAlignment*, this means that the first section will be aligned to the first multiple of the file alignment that occurs after the end of the final section header. The Optional Header also contains the *AddressOfEntryPoint* which is a memory address where the program starts

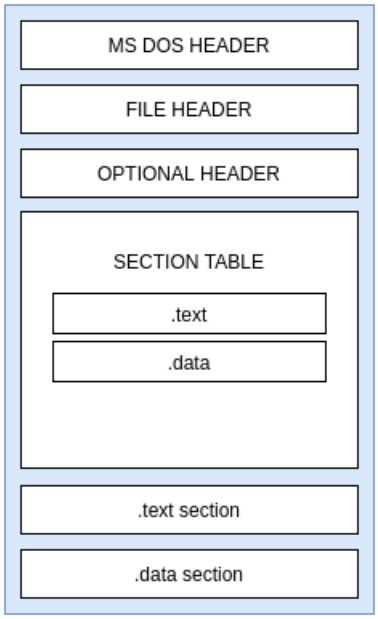


Figure 2.1: PE File Format Diagram

executing instructions. Although the research presented here is for PE files, it is also relevant to any file structure that can be broken down into sections such as ELF files for the Linux Operating System.

2.3.2 PE File Infection

Common techniques used by malicious actors to hide malicious code is to hide it in a benign file, by injecting it into a code cave [31], or to add a new section to the file [32]. This opportunity presents itself in PE files because of the empty space at the end of a section or the empty space between the last section header and beginning of the first section, both due to the file alignment respectively. Figure 2.2 illustrates file infection by inserting a new section at the end of the file and the corresponding section header.

This allows the attacker to inject shellcode into the new section and change the *AddressOfEntryPoint* to the new section, that when ran will execute and end with jumping back to the original *AddressOfEntryPoint* located in the Optional Header.

This also presents a greater vulnerability in ELF files as the Section Table is at the end of the file, meaning that an attacker can inject files regardless of the space available because of the file alignment.

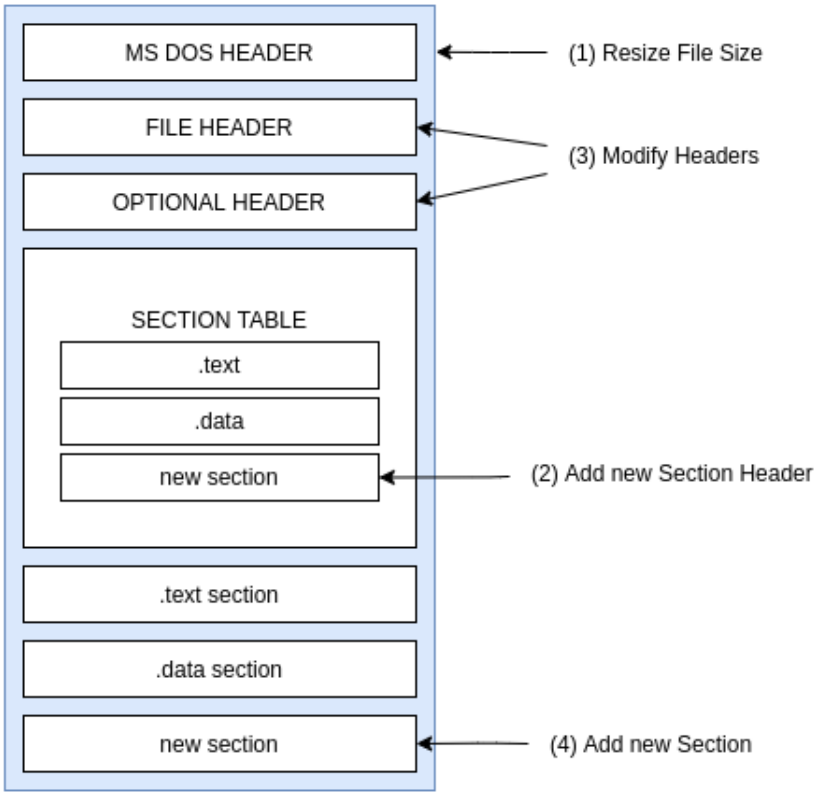


Figure 2.2: PE File Infection

The important ideas to take from this is that not every section in a file is neccesarily malicious. In the example of File Infection of a benign file, the *AddressOfEntryPoint* in the Optional Header could be modified to point to a malicious section that has been added to the file [32]. Although the file could now be considered as Malware if one was to compare the original file with the infected file, all of the sections except for the newly added section remain the same. Therefore those unaffected sections from Malware could be considered as benign.

Another reason for Malware potentially containing a Goodware section is that during Compilation, the Compiler may generate an empty *.tls* or *.bss* section. These sections can exist in both malign and benign files, implying that when analysing from a non-dynamic point of view, these are not malicious.

One issue when analysing Malware is that due to obfuscation techniques, just because a section has the name of *.data* does not mean that it will be a *.data* section.

2.3.3 Axioms

Following on from these ideas on the PE File Format, the research presented in this paper assumes the five following axioms are fundamentally true, and often refers back to them to

justify certain approaches and results.

I Software that is classified as benign must contain no sections that are malicious.

II A section from Malware may or may not be malicious.

III Software that is classified as malicious must contain at least one section that is malicious.

IV A section from Goodware can be trusted to behave as described by its name.

V A section from Malware cannot be trusted to behave as described by its name.

Chapter 3

Dataset

3.1 Datasets

Much of the research discussed in Chapter 2 is split between Malware Classification and Malware Detection, with the majority of research studying the former. One possible reason for this is that a high quality dataset for Malware Detection is hard to find. Malware datasets are available in great quantities from sources such as Virus Total [1] and MalShare [33] and Internal Company/Educational Resources [5]. Microsoft Malware Classification Challenge Dataset (BIG 2015) [9] and Drebin [34]. There are other datasets such as EMBER [35] and ClaMP [36] that contain malign and benign samples, however the data consists only of features extracted from the files and not the files themselves, making full Image-based analysis impossible. Finding a high quality large Goodware Dataset is challenging task, other researchers have used files from Sourceforge, Windows and Linux Operating Systems, online Software Respositories and creating variants of their own Software.

3.1.1 Online Repositories

The benefit of collecting Goodware from online repositories is that there are often a wide variety of files. The drawbacks are that it is time consuming to download all of the files, secondly you can't be certain if they are Goodware so they need to be scanned by Virus Total [1] and thirdly the Software is often old and limited in functionality so it doesn't represent modern Software.

3.1.2 Operating System

The benefits of using files from Operating Systems is the assurance that the files are Goodware and don't need to be checked by Virus Total. The limitation to this is that there is a very small amount of data, additionally the data is limited to applications for a specific purpose and size.

3.1.3 Creating own Dataset

Others have created a dataset by writing programs of sorting algorithms and used genetic evolutions to create many different variations on these. The problem here is that in the very unlikely example that one of these variations changes into a malicious piece of code, or a corrupted non functioning executable, this will be damaging to the dataset.

3.1.4 Evaluation

Regardless of which dataset is chosen there will be benefits and drawbacks. The best option would be to select files from each of these options. Files from the Windows Operating System were chosen as the Goodware dataset because it was a trustworthy dataset of well written and up to date Software. The Malware Dataset was gathered from 2018 and 2019 Windows EXE Files from Virus Total.

3.2 Data Cleaning

The following steps were taken to ensure the Goodware and Malware Datasets were as clean as possible. Both Datasets were filtered to only include 32 bit Executables. After filtering the Goodware dataset to only include files that were 32 bit executables, the Goodware dataset was made up of 774 files. This determined that the Malware Dataset should have only 774 files in total.

As Malware is often packed to obfuscate the file and make analysis difficult for Detection methods, the Malware dataset was required to be unpacked to match the Goodware Dataset. First, the Malware Dataset was filtered to include only Malware that was packed by UPX [37]. This is easily identifiable by checking the section names are in the format of UPX0, UPX1, to UPXN. The UPX Packer also has the ability to unpack executables packed with UPX. The UPX filtered Malware was then further filtered to Malware that had been succesfully unpacked by UPX. After being succesfully unpacked, the dataset was then filtered again to ensure that every file was still a 32 bit PE File that contained no UPX sections. The minimum filesize of

Goodware was 9KB and maximum was 9.9MB, so the final filtering for the Malware Datasets were filtered to a similar file size range.

As the data was taken from two Malware datasets (Virus Total 2018 and 2019 Windows Executables). After filtering the data as mentioned above, 387 files were chosen from each at random. This approach to cleaning the data ensured that the file sizes were in the same range, the data was unpacked and all files were x86 PE files.

3.3 Data Analysis

The following is an analysis on the final dataset.

Dataset		
	Goodware	Malware
File Count	774	774
Minimum File Size (bytes)	9886	9216
Maximum File Size (bytes)	9949880	9901384
Mean File Size (bytes)	982109.33	277277.06
Standard Deviation File Size (bytes)	1425522.11	665698.84
Section Count	3827	4099
Minimum Section Size (bytes)	0	0
Maximum Section Size (bytes)	8949760	7232512
Mean Section Size (bytes)	135856.66	51553.24
Standard Deviation Section Size (bytes)	454389.8	239764.41
Mean Sections Per File	4.94	5.3

3.4 Converting Files to Images

As described in Chapter 2, there have been many different approaches to representing PE files as images. Here the benefits and drawbacks of different approaches are evaluated.

3.4.1 RGB vs Greyscale

The majority of Image-based Malware Detection and Classification have used greyscale representations with some experimenting with RGB.

Singh et al. [13] experimented with greyscale and RGB images. Their approach to representing files as RGB images was an interesting and new approach. Instead of using the value of a byte to represent a greyscale pixel like previous approaches they instead used the upper and lower nibbles of the byte to be an index into a 2D colour map. They found that RGB images achieved a greater classification accuracy than greyscale images.

RGB

Using the ideas from [13] by using a colour map raises issues. A colour map with indeces of 0 to 15 has a total of 256 values and the total possible RGB values are 256^3 leaving 256^4 possible mappings from bytes to RGB values. The issue here is that this then becomes a search problem for finding the best mapping of bytes to RGB values.

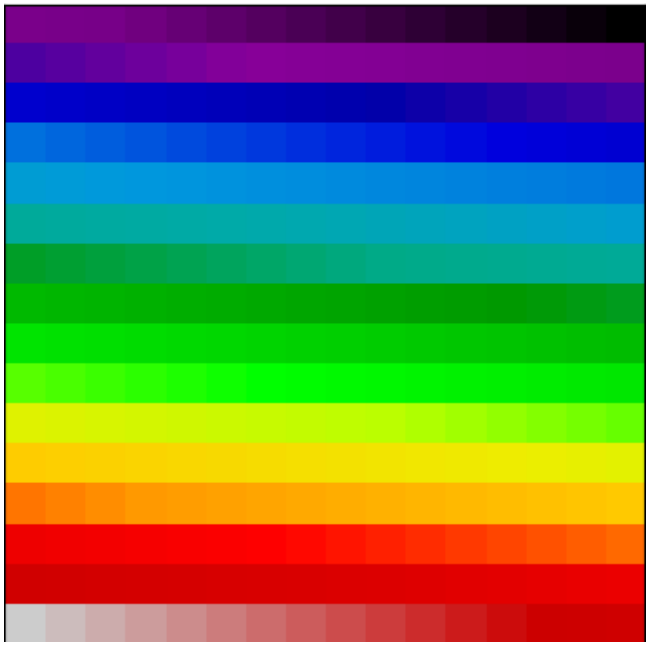


Figure 3.1: 2D Colour Map

This is the colour map used to experiment with how succesful a 2D colour Map was. This colour map was chosen as it tried to represent the complete range of possible RGB values.

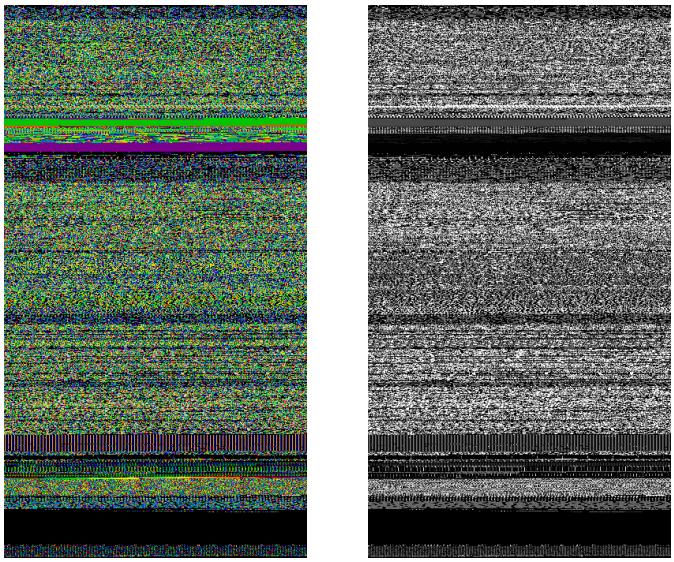
Greyscale

Experiments with both 1 channel and 3 channel greyscale were attempted. The greyscale images were created and read as 3 channel greyscale images, i.e. reading a value x and writing the pixel as (x, x, x).

RGB vs Greyscale Results

The results from these experiments were that the 1 channel and 3 channel greyscale images were achieveing very similar classification results, unsuprisingly the difference being that the 1

channel images were achieving the maximum results at a faster rate due to it being the same data duplicated three times.



((a)) Colour Malware ((b)) Greyscale Malware

Figure 3.2: RGB and Greyscale Malware Images

Contrary to the ideas presented in [13] the RGB non greyscale images were consistently lower than both the 1 and 3 channel greyscale images. This could be due to certain mappings being greater than others and therefore this is a search problem to find the optimal colour map. However it is possibly because an RGB greyscale image is the optimal mapping of a 2D colour map, and as this achieves the same results as the 1 channel greyscale experiments, the best option is a 1 channel greyscale image.

3.4.2 File Sizes

A Convolutional Neural Network was used as a Classifier for reasons explained in Chapter 4. Because of this, all of the images needed to be the same size. There have been multiple approaches to this in previous research, such as resizing images to 32x32 images [13], resizing images to 64x64 images [8] [11] and Spatial Pyramid Pooling to use different image sizes as inputs [14].

As this report is already experimenting with multiple different areas, using Spatial Pyramid Pooling with different file sizes seems more like a search and optimization problem similar to the colour map. This is because trying to find the best way to resize files due to their overall file size will achieve different results. Therefore choosing a specific image size for all files will

result in cleaner results and not a due to an optimization problem. 32x32 Images were chosen over 64x64 Images due to it being more computationally efficient and wanting to run many different tests as shown in the rest of the report.

Regardless of which approach is used, there will be a tradeoff between efficiency and accuracy.

3.4.3 Methodology

The following is the methodology used to convert a file to an image in this report.

Using a fixed width of 384 as shown in [13] and a variable height based on the size of the image, an RGB image is created with all pixels as $0x000000$. The dimensions of an image are then defined as follows.

$$(384, \lceil fileSize \div 384 \rceil)$$

For every byte in the file, use this value to represent a greyscale RGB value and write it into the next pixel in the image. After writing all bytes into the file, the files are then resized to 32x32 images.

3.5 Evaluation

There are many issues with the dataset chosen and the approach of converting files to images. The chosen dataset is made of high quality files but with very few examples and all for similar purposes. Collecting a high quality Goodware dataset of PE Files would require having access to a private high quality database of files or spending a large amount of time gathering a large amount of data. The majority of all datasets available consisting of complete PE files and those used in Malware Detection research are either too limited in scope or very dated and will potentially achieve bad results because of it.

Another option would be to generate variants of the Goodware to generate more data, however this runs into many problems identified later in the report. Generating Goodware runs into many of the same issues that generating Adversarial Variants of Malware has, such as corrupting the file or potentially making the file malicious.

For future work however this could be achieved through sandbox tools [21] [22] and by methods explained in Chapter 5 on Adversarial Variants.

Another issue with the dataset is that it is using UPX packed Malware that has been unsuccesfully unpacked, however there is not a further check to ensure the Malware is not still

packed with another packing or obfuscation technique.

As the dataset size is very small and to increase the size of the dataset, .dll files could also be included due to them having the same PE File Format as .exe files. Alternatively, ELF files could be used instead of PE files as this may make the gathering of data easier.

Chapter 4

Malware Detection

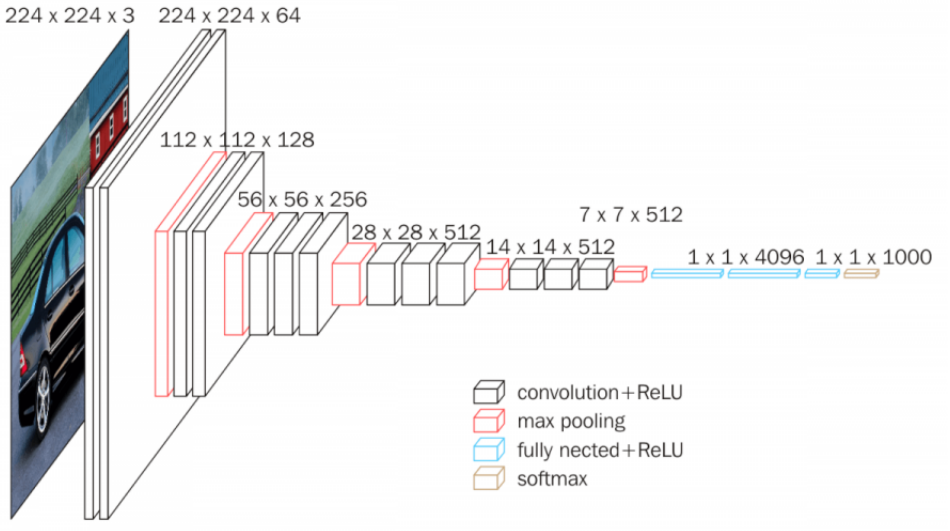
Two new methods of Malware Detection are presented in this Chapter. As previous research in Malware Detection uses different datasets, classification models and processing techniques it becomes very difficult to compare results. Therefore before presenting the two new Detection methods, a Detection method inspired by previous research is demonstrated using the dataset and processing techniques used in this report so that the results between the two new Detection methods can be compared more accurately.

4.1 Classifier Architecture

Previous approaches to Deep Learning-based Malware Detection have used K-Nearest Neighbors [4] [8], Support Vector Machines [6] [8] and Neural Networks [7] [10] [11] [12] [13] [14] [15]. As the research presented in this study is already experimenting with preparing data and new methods of Detection, a Convolutional Neural Network was chosen as the Classifier beacuse this requires less experimentation with regards to feature selection.

Previous research into Image Classification has seen great improvements with research into Convolutional Neural Networks such as ResNet-50 [13], LeNet-5 [38] and VGG [39]. These architectures were experimented with however they were not suitable as they were potentially too deep for the small dataset used in this report. Surpsingly, LeNet-5 didn't achieve the best results even though it is a well known 32x32 Image Classifier.

The Network Architecture used in this report is a small Network inspired by the structure of VGG-16 [39].



https://neurohive.io/en/popular-networks/vgg16/

Figure 4.1: VGG-16 Architecture

Because VGG-16 is designed for 224x224 images and the dataset in this report is for 32x32 images, the architecture was immitated with respect to pooling and filters. The layers had to be reduced due to the smaller dataset.

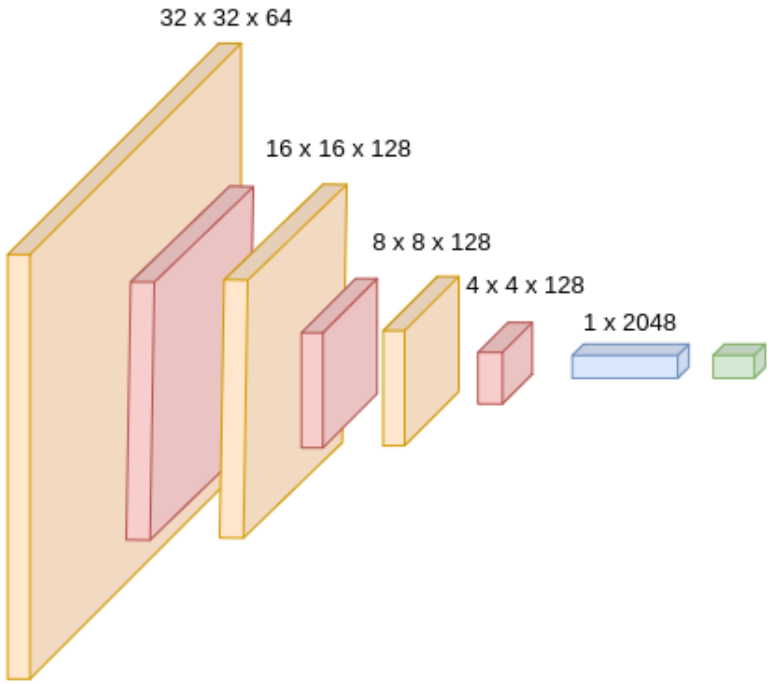


Figure 4.2: 32x32 Malware Detection Architecture Diagram

32x32 Malware Detection Architecture			
Layer	Filters	Filter Size	Stride
Convolution	64	3x3	1x1
Max Pooling	-	2x2	-
Convolution	128	3x3	1x1
Max Pooling	-	2x2	-
Convolution	128	3x3	1x1
Max Pooling	-	2x2	-
Fully Connected	2048	-	-
SoftMax	-	-	-

All of the experiments in this report are using the Malware Detection Architecture in Figure 4.2. The experiments in this report range from binary classification to multinomial classification, therefore the Architecture stays the same throughout with the exception that the final layer - SoftMax - will change with different experiments.

The training and testing dataset is randomized for every new classification, with the exception that one specific file was always included in the training dataset as will be explained in Chapter 5.

4.2 File Classification

The classifier that was used to compare results for the new Detection methods uses ideas from previous research [11] [13]. The approach is simple and consists of taking two labelled sets of data (Goodware and Malware), converting each set of files to images and using these two sets as inputs to train the Classifier.

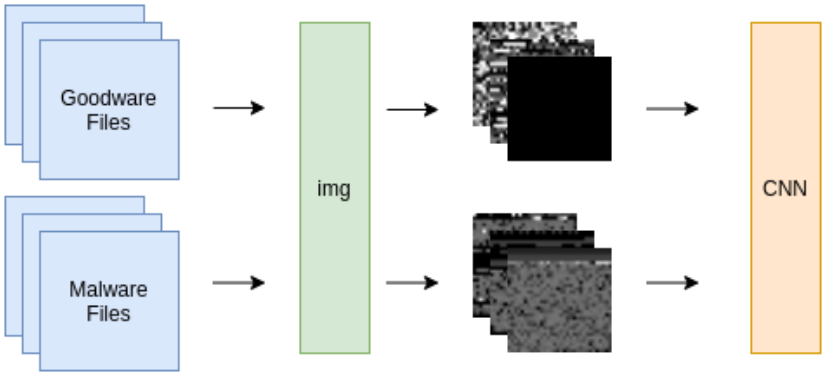


Figure 4.3: File Classifier Training

After training the Classifier, testing is very similar. Convert files to images and predict with the Classifier.

4.2.1 File Classification Results

After running the dataset on the File Classifier 100 times, each with randomized data and weights to ensure the results are consistent, the classifier was able to reach a maximum accuracy of 94.19% and minimum of 88.71% with a mean of 91.32%.

File Classifier Metrics	
Metric	%
Maximum Accuracy	94.19
Minimum Accuracy	88.71
Mean Accuracy	91.32
Accuracy	94.19
Precision	92.02
Recall	96.77
F1	94.33

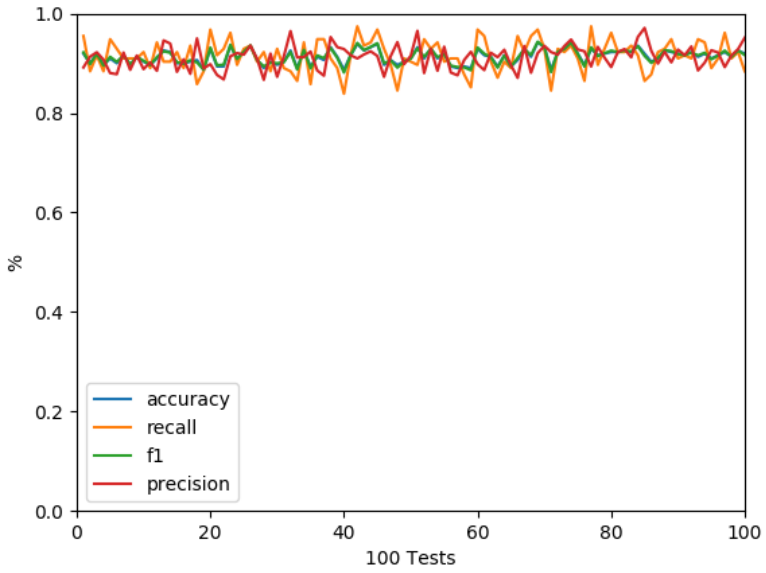


Figure 4.4: File Classification Metrics

4.3 Section Classification

Before presenting the two new Detection methods for Malware, it is important to recall one of the axioms explained in Chapter 2.

A section from Malware may or may not be malicious.

This implies that analysing a file by its sections may open up more possibilities for analysis, by trying to identify not just if something is malicious, but which part.

Instead of representing a file as one 32x32 image, breaking a file down into multiple 32x32 images makes the dataset larger, allowing each file to be examined by the Network in more detail without increasing the time complexity of the Network. Secondly, because different clusterings of data can be used, this will improve Classification Accuracy as will be demonstrated in Chapter 6.

4.3.1 Section Data Preparation

When preparing the files for Section Classification, the metadata in the Section Table is used to identify where the data for each section is located. This data is then read and written into a new file. For a file that contains n sections, n new section files will be created and the metadata and header data will be discarded. These files are then converted into images in the same way as described in Chapter 3. This can be visualized in the left side of Figure 4.6.

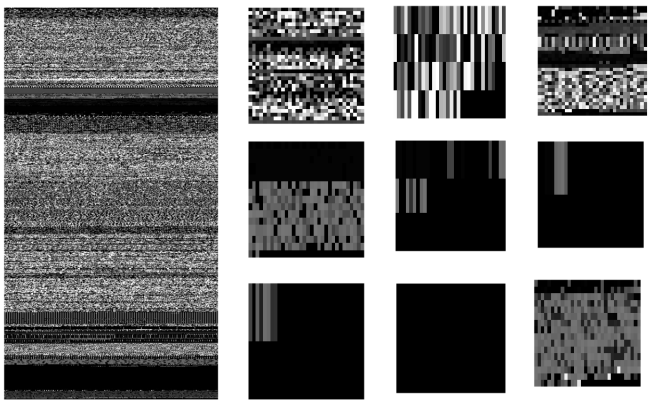


Figure 4.5: Section Images

4.3.2 Section Training

After preparing the data, the training of a Section Classifier is very similar to the File Classifier. The training data is two labelled datasets, Goodware Section Images and Malware Section Images. This can be visualized in the right side of Figure 4.6. These are used as inputs to train the Classifier with the labels being the original label of the file that the section came from.

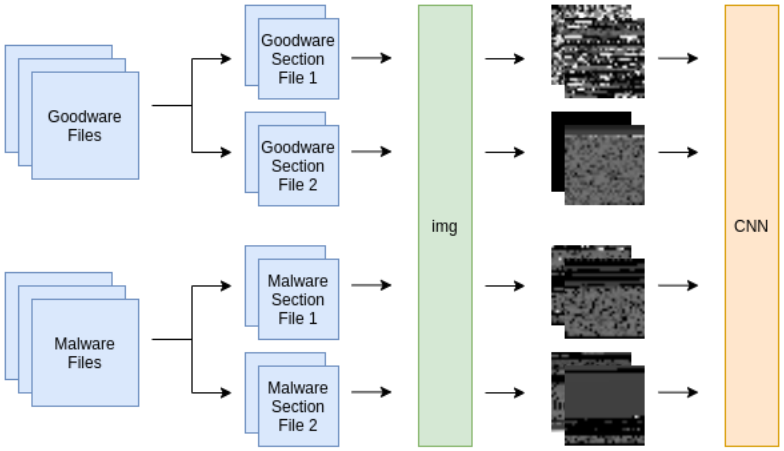


Figure 4.6: Section Classification Training

Section Testing Accuracy	
Metric	%
Maximum Accuracy	91.05
Minimum Accuracy	78.75
Mean Accuracy	89.04

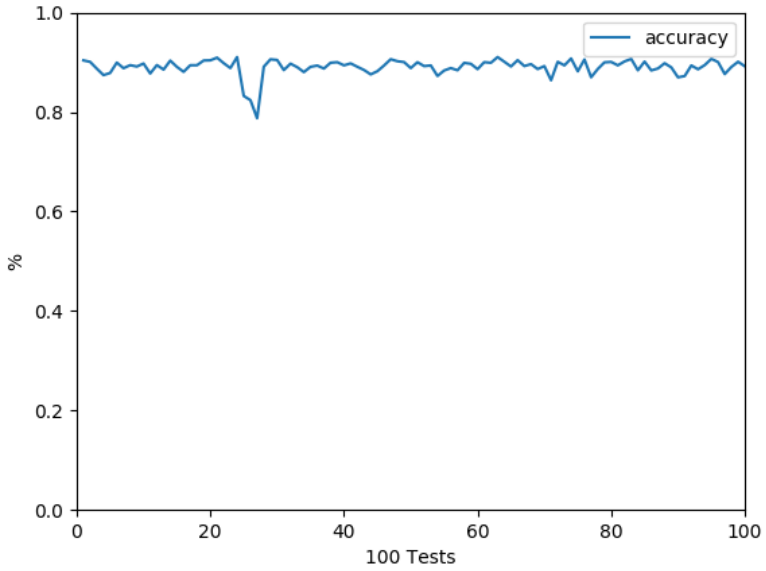


Figure 4.7: Section Classification Testing Accuracy

4.3.3 Section Testing

If Section Testing is conducted in the same way as training, i.e. split a file into sections, convert section files to images and predict the classification of images, the accuracy of this Testing achieves worse results than File Classification.

One possible reason for this is that the data is unclean to begin with because sections from

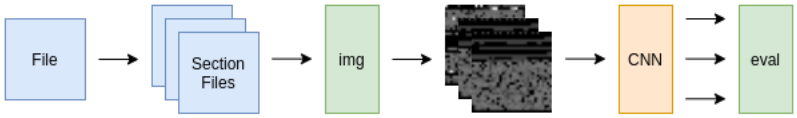


Figure 4.8: Section Classification Testing

Malware may not be malicious. The real power behind Section Classification is when an extra check is added to rectify this during testing.

During testing, each section of a file recieves a score from the Classifier of it's maliciousness, 0 for benign up to 1 for malign. Each section score from each file can then be evaulated together with a final evaluation function as seen in Figure 4.8. Two evaluation functions are presented here, the At Least One Classifier and Majority Vote Classifier.

At Least One (ALO) Classifier

The reasoning behind the ALO Classifier is based on the 1st and 3rd Axiom described in Chapter 2.

Software that is classified as benign must contain no sections that are malicious

Software that is classified as malicious must contain at least one section that is malicious.

Therefore, if a file has n sections the Neural Network will then make n classifications. If one of these is classified as malign then the file is classified as Malware, otherwise if all are classified as benign, the software is Goodware.

At Least One (ALO) Results

After running the dataset on the Section Classifier with ALO Evaluation 100 times, each with randomized data and weights to ensure the results are consistent, the classifier was able to reach a maximum accuracy of 82.9% and minimum of 54.19% with a mean of 64.77%.

Section Classifier (ALO) Metrics	
Metric	%
Maximum Accuracy	82.9
Minimum Accuracy	54.19
Mean Accuracy	64.77
Accuracy	82.9
Precision	74.52
Recall	100
F1	85.4

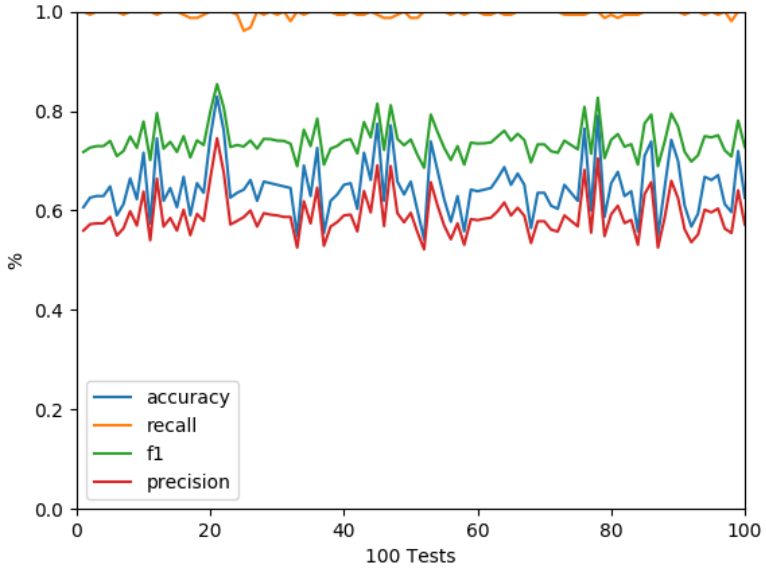


Figure 4.9: ALO Classification Testing

Majority Vote (MV) Classifier

The Majority Vote Classifier is a relaxed version of the At Least One Classifier. The ALO Classifier is a realistic approach and checks to see if at least one section is malicious to determine the classification, the Majority Vote Classifier uses a threshold. If the theshold is set at 0.6, then if 60% or more sections in a file are classified as malign, therefore it is Malware, otherwise it is Goodware. After conducting Section Training, an extra check on the testing set is checked to find the most optimal threshold that maximizes Accuracy.

Majority Vote (MV) Results

After running the dataset on the Section Classifier with MV Evaluation 100 times, each with randomized data and weights to ensure the results are consistent, the classifier was able to reach a maximum accuracy of 98.06% and minimum of 89.03% with an average of 95.05%.

4.4 Evaluation

In terms of Malware Detection, it is easy to see that the Majority Vote Classifier constantly outperforms the ideas from previous research, which itself outperforms the At Least One Clas-

Section Classifier (MV) Metrics	
Metric	%
Maximum Accuracy	98.06
Minimum Accuracy	89.03
Mean Accuracy	95.05
Accuracy	98.06
Precision	98.69
Recall	98.06
F1	99.34

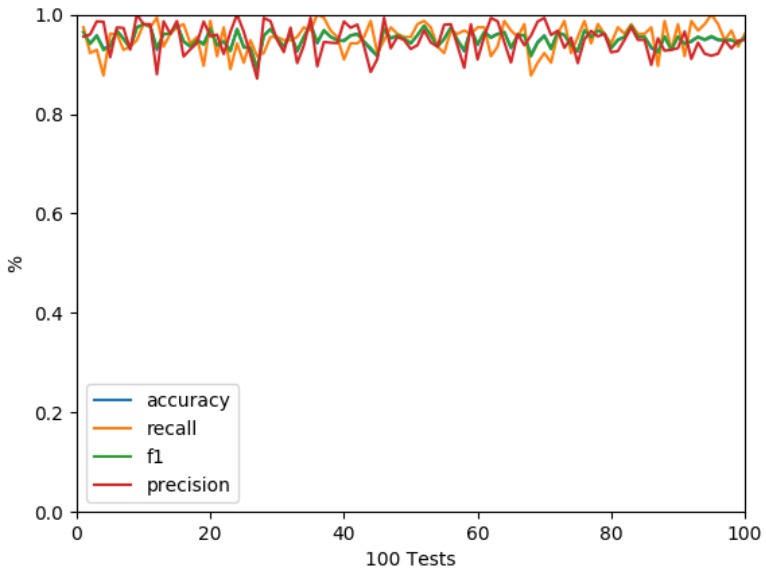


Figure 4.10: MV Classification Testing

sifier. It is a possibility that as training becomes more accurate, the Majority Vote threshold will approach the optimal percentage of malicious sections per file and At Least One accuracy will start increasing.

Malware Detection Metrics			
Metric	File Classificatier	Section (ALO)	Section (MV)
Maximum Accuracy	94.19	82.9	98.06
Minimum Accuracy	88.71	54.19	89.03
Mean Accuracy	91.32	64.77	95.05
Accuracy	94.19	82.9	98.06
Precision	92.02	74.52	98.69
Recall	96.77	100	98.06
F1	94.33	85.4	99.34

One possible reason for the ALO Classifier achieving a worse classification rate is because it is more likely that a file is classified as Malware as the average sections per file being between 4 and 5. With only one of those sections needing to be classified as malign for the file to

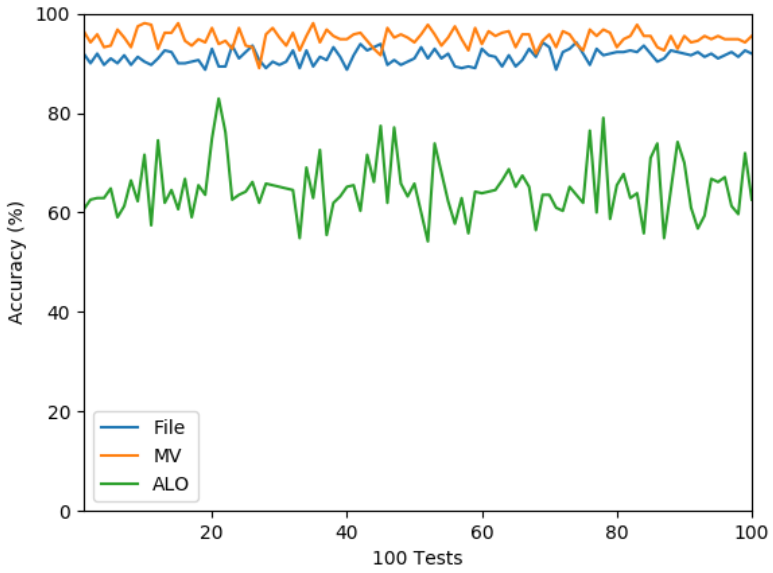


Figure 4.11: Detection Accuracy

be labelled Malware. Also not every section from from Malware is malicious, so there may be incorrectly labelled data causing the training to be flawed which will cause the testing to predict many False Positives and False Negatives.

Further areas for experimentation could involve selecting different features such as GIST, Gabor Wavelet features as previous reports of Malware Detection have examined. Further research into different evaluation functions other than MV and ALO could also be experimented with.

The Majority Vote Classifier shows that analysing by section can achieve greater Detection rates than analysing by file, this is due to examining the data in greater detail without increasing time complexity of the classifier as well as using Evaluation functions.

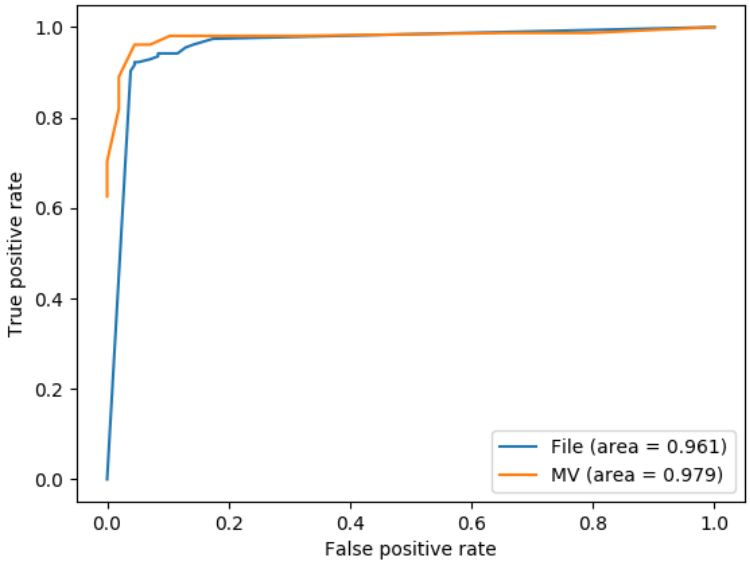


Figure 4.12: ROC Graph

Chapter 5

Adversarial Variants

The past decade has seen a vast amount of research on Adversarial Variants to trick Deep Learning models for Image Classification. It was shown that making small perturbations of images could cause Classification models to misclassify the image [16] and the FGSM method for creating Variants was created [17].

The FGSM method works for creating Adversarial Variants of Images, previous researchers have attempted to use these methods to create Adversarial Variants of Image representations of Malware. Applying these perturbation techniques to image representations of Malware is fundamentally flawed [24] [28]. The reason being is that the image is a representation of the PE File Format, making changes to any pixel of an image will represent a change to a byte in the file which could easily cause the file to become corrupted or change instructions in the file that will alter the behaviour. Another problem with this approach is that altering the image is unrealistic in itself. In a real world scenario, Malware would be obfuscated prior to being captured for analysis. Altering the image doesn't ensure that the changes being made to the file are possible.

In this section, a formalization for creating Adversarial Variants is discussed, followed by different approaches to creating Adversarial Variants of Malware. Finally, Adversarial Variants are created for the three classifiers presented in Chapter 4.

5.1 Formalization of Adversarial Variants

Papernot et al. [19] described the Adversarial Goal as producing a minimally altered version of any input $\overrightarrow{x}$, named adversarial sample, and denoted $\overrightarrow{x*}$, misclassified by oracle

$O : \bar{O}(\overrightarrow{x*}) \neq \bar{O}(\overrightarrow{x})$.

It's possible that because this is regarding Image Adversarial Variants, creating a minimally altered version is the goal as altering it too much may cause it to be become unrecognisable. However in the example of Malware, the goal is to create an altered version of Malware so that the functionality remains the same whilst also causing misclassification.

Creating an Adversarial Variant with the goal of satisfying the condition $O : \bar{O}(\overrightarrow{x*}) \neq \bar{O}(\overrightarrow{x})$ is also not specific enough for certain conditions. As there are multiple different goals when creating Adversarial Variants, e.g. Targeted Misclassification and Untargeted Misclassification [18] [15].

Hu and Tan describe a Generator that converts a feature vector into an Adversarial Version [23]. Dang, Huang and Chang describe a Morpher and Tester when creating Adversarial Variants of PDF Malware where the Morpher inserts, replaces or deletes objects in a PDF file and the Tester ensures the behaviour remains the same [22].

Following on from these ideas, it is important to define a strict method of creating Adversarial Variants. This involves a Generator G, Oracle O and Classifier C.

Generator

A Generator takes data and generates an Adversarial Variant. With regards to Image Variants using the FGSM method [17] this is defined as

$$x + \epsilon * \text{sign}(\nabla_{\text{x}} J(\theta, x, y))$$

In the example of PDF Malware [22] this is a Tester T that inserts, replaces or deletes an object in a file. A more general definition of this is $G(x)$. Generator G takes example x and generates x'

Oracle

An Oracle ensures that the Adversarial Variant created by the Generator maintains its original purpose. In the image examples [16] [17], this is simply checking that the generated variant is still identifiable to the human eye. This is the issue that other researchers of creating Adversarial Variants for Image Representations of Malware have encountered [24] [28]. This being that in creating an Adversarial Variant of an Image Representation of Malware, the only check to see if it is a succesful variant is that the images look the same to the human eye.

With this formalization of a Generator and an Oracle, the description of a minimally altered version as previously described is not neccesary [19]. This itself can be replaced by an Oracle

that ensures the variant is still recognisable to the human eye regardless of it being a minimally altered version. In other examples of Adversarial Variants of Malware, Cuckoo Sandbox was used as an Oracle to ensure the behaviour of the variant remained the same [21] [22].

An Oracle can then be defined as $O(x, x')$. A variant is evaluated as a succesful variant if the qualities of both x and x' remain the same.

Classifier

The Classifier $\mathcal{C}$ outputs a predicted classification for the variant, this is then used to evaluate whether the variant achieved the original goal, i.e. Untargeted Misclassification or Targeted Misclassification. For a variant x' and original classification y, Untargeted Misclassification can then be defined as $C(x') \neq y$ and Targeted Misclassification can be defined as $C(x') = t$ where t is the target class.

Therefore when creating Adversarial Variants for Untargeted Misclassification the following definition must hold true for them to be succesful.

$$(C(G(x)) \neq y) \wedge (O(x, G(x)))$$

For Targeted Misclassification the following must hold true for them to be succesful.

$$(C(G(x)) = t) \wedge (O(x, G(x)))$$

These definitions should ensure that future Adversarial Variants of Malware are succesful.

5.2 Adversarial Variants of Malware

As discussed by Anderson [20], there are many different approaches to generating Adversarial Variants of PE Malware Files.

1. Add Function to Import Address Table
2. Manipulate Section Names
3. Create new unused Sections
4. Append Extra Bytes to the end of Sections
5. Create a new entry point
6. Manipulate signature

7. Manipulate debug information
8. Pack/Unpack file
9. Modify Header Checksum
10. Append bytes to the overlay

This can be further summarized into the following areas

1. Modify Metadata
2. Modify Existing Data
3. Insert New Data

In this report, Adversarial Variants will be generated by inserting new data into the file with File Infection attacks as described in Chapter 2 to add new Sections to the file [31] [32]. The Generator can then be defined as follows

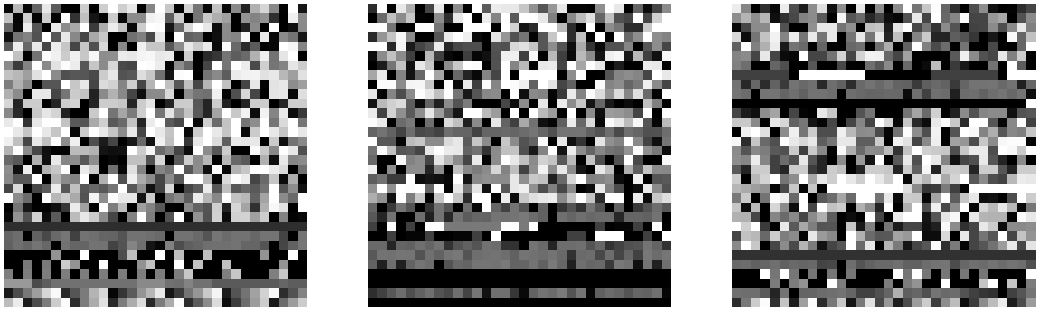
1. Get data to insert into/infect the victim file
2. Resize the victim file
3. Alter Header Information
4. Add New Sections
5. Add New Entry in Section Table

The Oracle in this example is ensured by modifying the File Format correctly. By ensuring there is enough space for a new section header in the file, the modification won't be overwriting any data in the first section and will appear to be genuine. Any variants that attempted to modify existing data would require a more advanced Oracle. The Classifier is the Convolutional Neural Network as shown in Figure 4.2.

5.2.1 File Classifier - Adversarial Variants

The vulnerability that allows attackers to create Adversarial Variants for the File Classifier is that the images are resized. This means that the final image can be manipulated by the amount of data used to infect the file. A 1MB file infected with a 9MB file will be dominated by the 9MB file.

To create Adversarial Variants for the File Classifier, every piece of Malware was infected with a new section containing all of the bytes from *MicrosoftEdge.exe*. When running experiments on File Classification and Adversarial Variants, *MicrosoftEdge.exe* was always included in the training dataset.



((a)) MicrosoftEdge.exe ((b)) Malware ((c)) Malware infected with MicrosoftEdge.exe

Figure 5.1: Goodware, Malware, Adversarial Variant

5.2.2 Section Classifier - Adversarial Variants

The Section Classifiers work by analysing each section individually. Because of this, the section Adversarial Variants are created by infecting the victim with multiple new sections. By doing this, when making judgements such as Majority Vote, the vote can be influenced by the infected sections raising the amount of Goodware Sections.

These Adversarial Variants were created by inserting the *.text* section from *ApplySettingsTemplateCatalog.exe* 10 times into every piece of Malware. When running experiments on Section Classification and Adversarial Variants, *ApplySettingsTemplateCatalog.exe* was always included in the training dataset.

5.3 Results

The following results show the successes and failures of the Adversarial Variants. The method for testing the Adversarial Variants is as follows. Train the Classifier on randomized initial weights and dataset (with the exception that the Goodware file used to infect the Malware is always included in the training dataset). After training, the filenames from the testing dataset are used to load two dataset, (1) Malware and (2) Adversarial Variants of the Malware. These two sets are then ran through the Classifier. As this report is using binary classification and targeted misclassification, the success of the variants can be seen through the change in False

Negatives.

5.3.1 File Classifier

Comparing the False Negatives of the File Classifier for each set (Malware and Adversarial Variants), it is easy to see that in over 100 randomized tests the Adversarial Variants always work in bypassing the Classifier.

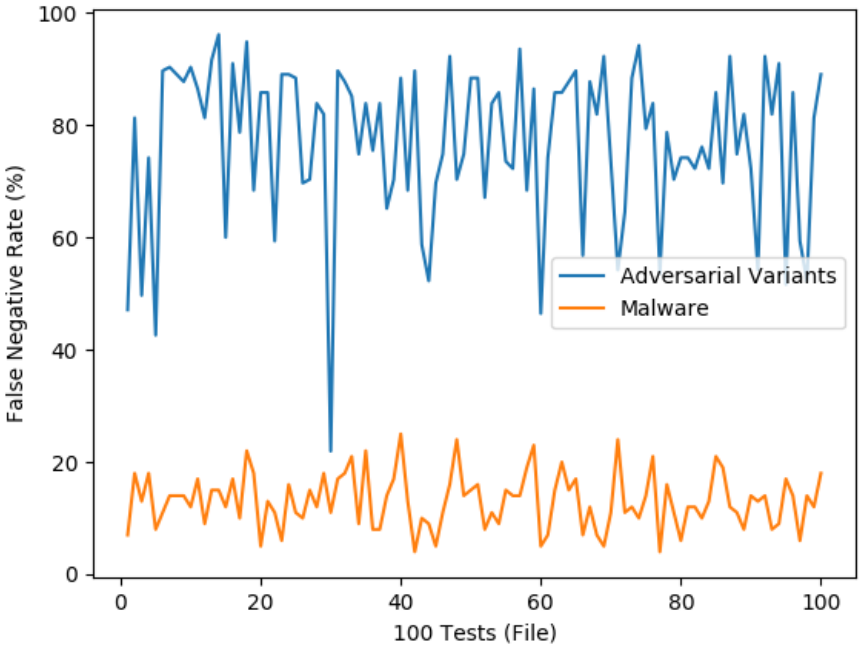


Figure 5.2: Adversarial Variants - File Classifier

File Classifier - Adversarial Variants		
False Negative Rate	Malware	Variants
Maximum	16.13	96.13
Minimum	2.58	21.94
Mean	8.44	76.93

5.3.2 Majority Vote Section Classifier

Comparing the False Negatives of the Majority Vote Section Classifier for each set (Malware and Adversarial Variants), it is easy to see that over 100 randomized tests the Adversarial Variants appear to alternate between 100% False Negatives and 0% False Negatives, with the rare exception of 90-99% False Negatives.

The reason for this is that when the section that was added 10 times in each file - regardless of being a part of the training Goodware dataset - is classified as Goodware, the file is seen to the Classifier and evaluation function as having at least 10 Goodware Sections and will only be classified as Malware depending on the threshold and count of other sections being classified as Malicious. This is the reason that the False Negatives are either 100% or close to 100%.

The Adversarial Variant False Negatives will be 0% when the file used for infection is classified as malicious as at least 10 sections in every file will be classified as malicious.

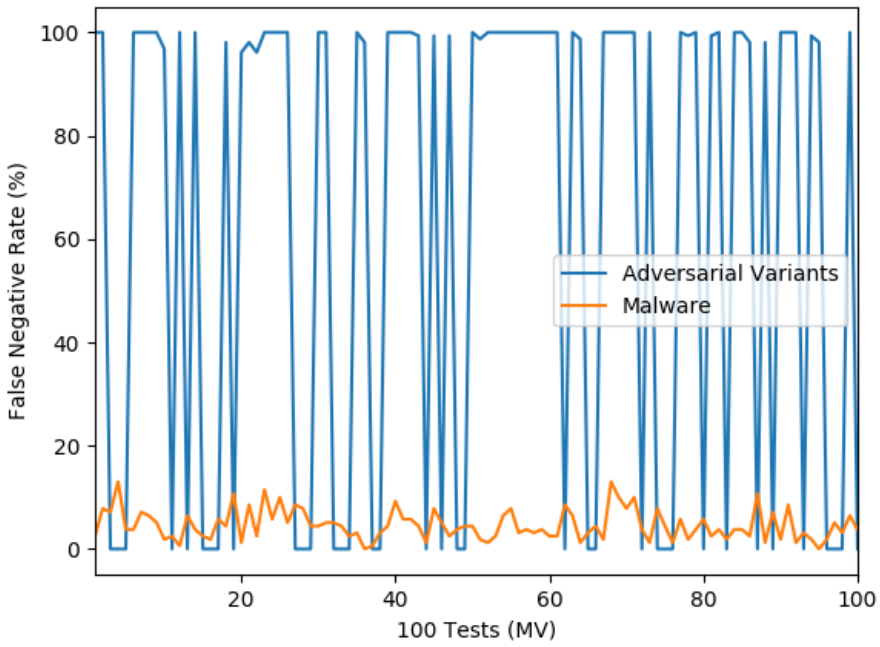


Figure 5.3: Adversarial Variants - Majority Vote

MV Classifier - Adversarial Variants		
False Negative Rate	Malware	Variants
Maximum	12.26	100
Minimum	0	0
Mean	4.65	62.71

5.3.3 At Least One Section Classifier

Comparing the False Negatives of the At Least One Section Classifier for each set (Malware and Adversarial Variants), the False Negatives of each group are always very low or 0%.

Because of the nature of the ALO Classifier only looking for one malicious section in a file, the Adversarial Variants here cannot make the classifier misclassify the Malware. The reason

for this is that if the unaltered file is classified as malign and it has n sections containing m sections already classified as malicious, therefore $1 \leq m \leq n$. As the Adversarial Variants presented here don't modify the existing sections, new sections cannot reduce the value of m.

Therefore by inserting sections that the Classifier considers to be benign, the classification will not change, but by inserting sections that are classified as malign then a file that contained only sections classified as benign will now be considered Malware. Therefore these attacks can only make files more malicious in nature not less malicious to this classifier.

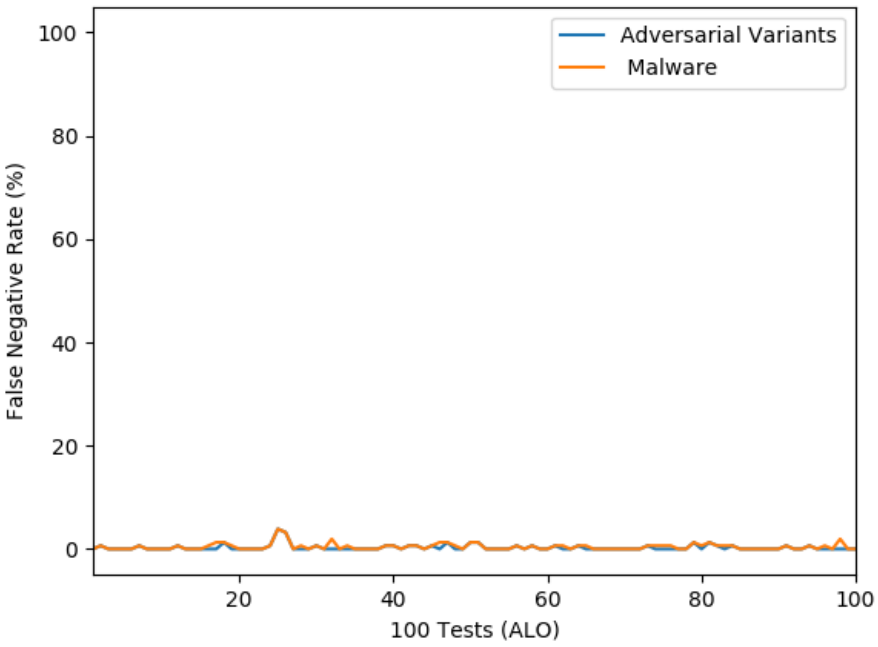


Figure 5.4: Adversarial Variants - At Least One

ALO Classifier - Adversarial Variants		
False Negative Rate	Malware	Variants
Maximum	12.26	3.87
Minimum	0	0
Mean	4.65	0.28

5.4 Evaluation

The File Classifier and Majority Vote Classifiers are both incredibly vulnerable to File Infection attacks. The At Least One classifier is invulnerable to File Infection attacks focussing on adding new data. Adversarial Variants for the ALO Classifier need to be approached from modifying

existing data to maintain the behaviour and executability of the file whilst also modifying the file.

Adversarial Variants			
False Negative Rate	File	MV	ALO
Maximum	96.13	100	3.87
Minimum	21.94	0	0
Mean	76.93	62.71	0.28

There has been research involving appending data to files to create Adversarial Variants [26] [27] [15] however nothing is mentioned about the modification of the PE File Format, Section Table and Headers. Although this would be succesful in creating Adversarial Variants for the File Classifier and other approaches in previous research, it would likely not be effective against the Section Classifiers described here. The reasoning is that the Section Classifiers use the metadata in the Section Table to locate the section data, if a variant is created by appending data to the file without altering the Section metadata this could result in the modification being ignored when reading the data.

The methods used here to infect the Malware were simple and more advanced methods could be used, such as modification of data. One potential issue with the variants generated for the Section Classifiers in this report are that when creating the adversarial variants, there was no check to ensure there was space to inject 10 sections in the section table.

Alternatively, using the ELF File Format would circumvent this issue as the Section Table is located at the end of the file. In a PE File, the Section Table could also be made larger and the other Sections relocated, however this would involve altering many instructions within those files such as memory locations and certain instructions in the binary. Future work into bypassing ALO needs to revolve around modifying existing data rather than inserting new data.

The Adversarial Variant formalization could also be used to create variants of files to create a larger dataset, as that approach was criticized in Chapter 3.

Chapter 6

Proposal

In Chapters 4 and 5, two new methods of Detection were examined. The Majority Vote Classifier was shown to be highly accurate but very exploitable whereas the At Least One Classifier didn't achieve a high accuracy but was invulnerable to File Infection attacks. This Chapter outlines and describes a third and final Malware Detection method that aims to achieve a high accuracy similar to the Majority Vote Classifier whilst also being invulnerable to File Infection attacks by injecting sections like the At Least One Classifier.

The premise behind this third Classifier is that Classification can be conducted not just by benign/malign groups but also the purpose of the section. Before continuing on to the explanation, it is first important to review the axioms described previously in Chapter 2 as these will be commonly used to explain the new Detection architecture.

I Software that is classified as Benign must contain no sections that are malicious.

II A section from Malware may or may not be malicious.

III Software that is classified as malicious must contain at least one section that is malicious.

IV A section from Goodware can be trusted to behave as described by its name.

V A section from Malware cannot be trusted to behave as described by its name.

6.1 Proof of Concept

It is possible to seperate the sections of the Goodware into their respective types by the section names. It is not possible to do this with Malware due to irregular section names, and obfuscation techniques. To show that the concept of classifying by section type works, the Goodware is

seperated into two different datasets. The 3 Class dataset consists of the following classes: *DATA*, *.text* and *OTHER*. The *DATA* group consists of *.data*, *.idata*, *.didat* and *.rdata* sections. The 8 Class dataset consists of the following classes: *.data*, *.text*, *.didat*, *.idata*, *.rsrc*, *.reloc*, *.rdata* and *OTHER*. The *OTHER* group consists of any section that doesn't fit in the other categories.

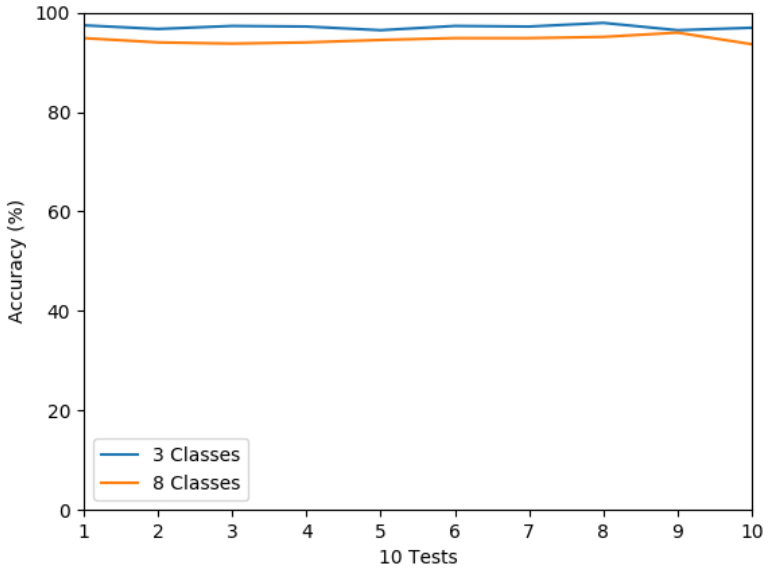


Figure 6.1: Proof of Concept

Goodware Section Classifier		
Metric	8 Groups	3 Groups
Maximum Accuracy	95.96	97.92
Minimum Accuracy	93.63	96.45
Mean Accuracy	94.55	97.08

Recall from Section 4.3.3 that the Section Testing Accuracy before applying ALO and MV was able to achieve a maximum accuracy of 91.05%, after applying the MV Evaluation this then achieved 98.06% Accuracy. If a correctly labelled Malware Section dataset existed then it could have the potential to improve the pre evaluation accuracy from 91.05% to 97.92% as shown in Figure 6.1 and therefore this would greatly improve the MV and ALO Accuracy.

Another benefit of this approach is that experiments can be done with different groupings, this will also allow a section to not only be identified as malign or benign, but also if it is a *.data* section or *.text* etc.

Some examples of groupings could be *Malware* or *Goodware*. More advanced groupings could also be applied such as *Malign .data*, *Benign .data*, *Malign .text*, *Benign .text*, *Malign*

OTHER and *Benign OTHER*.

This cannot be applied to Malware without first having a correctly labelled Malware Section dataset. One approach would be to manually analyse every section from Malware in the Malware dataset to categorise it into its type however this is infeasible due to time. It's also important to note that because every section from Malware is not neccesarily malign, if using a correctly labelled Malware Section Dataset some sections taken from Malware would be grouped into the Goodware Section Dataset.

6.2 Section Classification Proposal

Following on from the Proof of Concept, a proposal is explained that will allow this concept to be applied to a Malware Detection Dataset. This involves five steps that will be explained here and visualised in Figure 6.2.

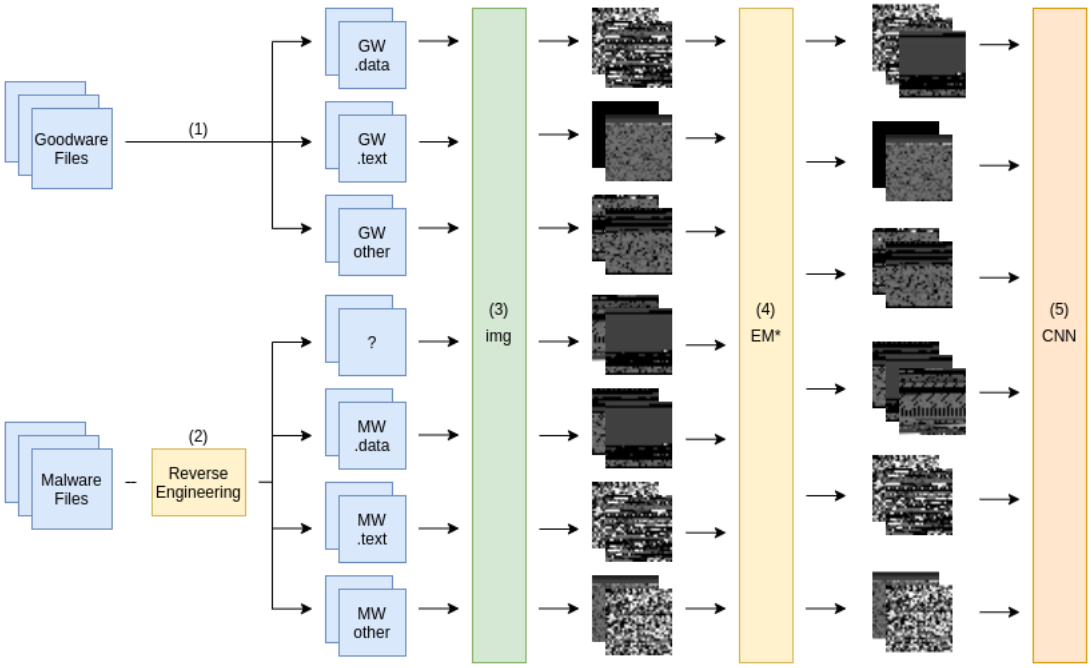


Figure 6.2: New Classifier Training Architecture

6.2.1 Preparing Goodware

The first section labelled (1) in Figure 6.2 is simple and involves taking the Goodware Dataset and breaking each file into section files. Because Goodware must contain only benign sections and the sections can be trusted by the section name, the section names can be used to determine

the section type and therefore determine which group the section belongs to. This results in a correctly grouped Goodware Section dataset.

6.2.2 Preparing Malware

The second section labelled (2) in Figure 6.2 is more complicated. Because Malware must contain at least one malign section to be considered Malware and a section from Malware cannot be trusted by its name, therefore any section from Malware may belong to Malware or Goodware.

Manual Malware Analysis and Reverse Engineering is then required to analyse the Malware to identify whether a section is malicious and the section type. It is important to note that just because there are no sections identified as malicious during manual analysis, this makes the file unidentified, instead of Goodware. Thus, if the target classes for both Malware and Goodware as shown in Figure 6.2 are *.data*, *.text* and *OTHER*, the resulting groupings of data that can discovered through manual analysis of the Malware can only be *Malign .data*, *Malign .text*, *Malign OTHER* and *Unknown*.

As much manual analysis as possible is beneficial as this will result in a more accurate dataset, however the fourth section to this architecture will handle any sections that are unknown or have not been analysed.

6.2.3 Converting to Images

The third section labelled (3) in Figure 6.2 is for converting files to images and can be approached in the same way as described in Chapter 3.

6.2.4 Clustering

The fourth section labelled (4) in Figure 6.2 is where things become slightly more complicated. In the example in Figure 6.2, there are 7 classes. Benign and malign *.data*, *.text* and *OTHER* as well as an unknown class. The purpose of Clustering is so the unknown class can be clustered into the other malign or benign groups. This is therefore a Semi Supervised Learning problem.

This could be achieved through a modified Semi Supervised version of Expectation Maximization [40] as it will find the maximum likelood estimates of incomplete data, with the incomplete data being the unknown classifications of the unknown dataset. Image Segmentation could be used with Expectation Maximization to identify features for clustering [41].

There are two requirements of the modified clustering algorithm to ensure it achieves optimal results. (1) Because Expectation Maximization uses probabilities to predict classifications of data, every time the algorithm estimates the values for the variables and optimizes the probabilities of the model, all of the data that didn't belong to the unknown class after section 2 are hard clustered into their original classification with probability of 1. This ensures that the decisions made before clustering remain the same and only the sections in the unknown class will change classification. (2) Because Malware must contain at least one section to be considered malicious, a minimal requirement for the clustering algorithm must ensure that at least one section from every Malware file must belong to at least one malign cluster, e.g. *Malign .data*, *Malign .text* or *Malign OTHER*.

6.2.5 Classification

The fifth section labelled (5) in Figure 6.2 is the same method as the other Classifiers explained in this report and explained in Chapter 4.

6.2.6 Testing

After training the classifier, the testing of files is the same as the ALO Section Classifier in Chapter 4. This ensures the Malware Detector is invulnerable to File Infection attacks demonstrated in Chapter 5.

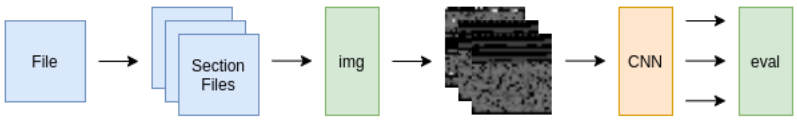


Figure 6.3: Section Classification Testing

6.3 Evaluation

This approach to Malware Detection will firstly allow an improvement in Accuracy and Speed without increasing the time complexity of the Network due to scanning multiple small images instead of one large image. It allows different clusterings of data such as section types, which will ultimately improve Accuracy as well as classify the type of section. Testing the Classifier with the ALO Evaluation means that this will be invulnerable to File Infection attacks that insert new data into files.

The limitations of this approach is that it requires much more manual intervention and work to prepare a dataset. Also with using Windows files, the files are consistently named which allows easy categorization of files by naming. Using other datasets may have inconsistently named files due to different Compilers.

Chapter 7

Legal, Social, Ethical and Professional Issues

With research such as this, involving software with the potential to cause damage and infect many other machines, great rules of care must be adhered to to prevent unnecessary damage. Additionally, it is of the upmost important to abide by the rules and regulations set forward in the British Computer Society (BCS) Code of Conduct & Code of Good Practice [42]. The Malware Dataset was given from VirusTotal [1]. As the Malware is designed to cause damage or have unintended consequences to the user, extra care was taken when running and analysing the Malware during research. This involved running the Malware in a Virtual Machine on an air gapped Computer so that the Malware was unable to spread and infect other Computers. As the Goodware Dataset is from the Windows Operating System, it would be against the Software Licence Terms to submit the dataset as part of the research submission. Additionally, it would be unethical to submit the Malware Dataset in the case it being downloaded and ran under the assumption it was not Malware. The field of Security can be seen as a cat and mouse arms race of technology to stay ahead of the other side. This report involves a large discussion and creation of software that can alter Malware to make it more effective in thwarting methods of defence. The premise behind this is that good offence is often the best defence and this research discusses and concludes by demonstrating a Detection method that can defend against these attacks and help improve the field of Security.

Chapter 8

Conclusion

8.1 Future Work

There are many areas examined in this report that can be researched for future work.

8.1.1 Dataset

Gathering Data

One of the biggest problems encountered in this report was finding a high quality large dataset of PE Goodware files. This would require gathering lots of complete, new and old PE Files from multiple sources, all that are examined with VirusTotal to ensure the file is benign.

File to Image Conversion

There were many decisions that were made in deciding on the approach for converting files to images. Many of these decisions were flawed in at least one way and others were search problems for finding the optimal solution.

Singh et al. [13] explained that their approach to using a 2D colour map achieved greater results than greyscale. The research in this report as explained in Chapter 3 found the opposite. This is a search problem for finding the optimal mapping of values from a byte to RGB values (256^4 possible mappings), that is then used for converting images that maximizes Classification Accuracy.

Other research resized files into different groups depending on the file size [4] [6]. This is another search problem for finding the optimal size for resizing files based on their original file size.

Unpacking techniques were only briefly examined in this research and could be further researched to ensure the files in the Malware Dataset were completely unpacked. There has been research on how to safetly unpack Malware [43] as well as automated unpacking services [44].

8.1.2 Malware Detection

The most important area for future work is undoubtably the main idea presented in the research, the proposal for a high Accuracy Section Classifier that is invulnerable to File Infection attacks that insert data.

Other methods for Detection could involve Section Classification with other feature methods such as GIST and Gabor Wavelet features used in other Detection and Classification research [6] [4] [8].

With the proposal for the new Section Classifier, future research could be looked into to examining data sections to determine the location of variables and to identify variables that appear to be for Ransomware or Shellcode.

8.1.3 Adversarial Variants

Anderson described multiple ways in which Adversarial Variants of Malware can be created [20]. The Adversarial Variants in this report were using File Infection methods to insert new Sections into the file. The ALO Classifier was demonstrated to be invulnerable to these methods, so future work to create Variants for this Classifier must examine modifying metadata or modifying existing data.

As described in Chapter 5, previous researchers have attemped creating Variants by altering existing data in the file but this is often flawed due to not considering the behaviour of the generated variant. In Chapter 5, an formalization of creating Adversarial Variants is presented involving a Generator, Oracle and Classifier. The Oracle is often what is ignored in the research that generates unusable Variants, therefore another area of research could be examining how to create an Oracle that ensures the behaviour of a generated variant remains the same as the original example. The problem for this however, could be viewed as the Halting Problem [45].

Although the approach from Hu and Tan [23] used to create Adversarial Variants was a theoretical representation and not a realistic approach by representing a file as a 160 length vector, each feature representing an API call. This method could be used to examine which API functions could be added to a file with File Infection attacks to create new Adversarial Variants.

8.2 Conclusion

Two new Detection methods have been demonstrated with a new Malware Detection Training and Testing Structure. Previous unsuccesful attempts at creating Adversarial Variants have been reviewed and a formalization for creating succesful Variants presented. Succesful Variants for the new Detection methods have been evaluated and finally a proposal for a classifier that is invulnerable to File Infection methods by inserting new data has been proposed.

Ultimately, the use of examining by section has been shown to improve Detection whilst also being invulnerable to Adversarial Variants. This has also shown that previous assertions of Deep Learning-based Malware Detection not being resistant to obfuscation is not yet certain [3].

References

[1] V. Total, "Virus Total." https://www.virustotal.com/gui/home. [Online; accessed 16-April-2020].

[2] A. Test, "Malware." https://www.av-test.org/en/statistics/malware/. [Online; accessed 16-April-2020].

[3] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[4] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: Visualization and automatic classification," 07 2011.

[5] U. of California Santa Barbara, "University of California Santa Barbara Vision Research Lab." https://vision.ece.ucsb.edu/. [Online; accessed 10-April-2020].

[6] K. Kancherla and S. Mukkamala, "Image visualization based malware detection," in *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pp. 40–44, 2013.

[7] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20, 2015.

[8] A. Makandar and A. Patrot, "Malware class recognition using image processing techniques," in *2017 International Conference on Data Management, Analytics and Innovation (ICDMAI)*, pp. 76–80, 2017.

[9] Microsoft, "Microsoft Malware Classification Challenge (BIG 2015)." https://www.kaggle.com/c/malware-classification/discussion/73433. [Online; accessed 16-April-2020].

[10] Dali Zhu, Hao Jin, Ying Yang, D. Wu, and Weiyi Chen, "Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, pp. 438–443, 2017.

[11] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of iot malware based on image recognition," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, pp. 664–669, 2018.

[12] Y. Ye, L. Chen, S. Hou, W. Hardy, and X. Li, "Deepam: a heterogeneous deep learning framework for intelligent malware detection," *Knowledge and Information Systems*, vol. 54, pp. 1–21, 04 2017.

[13] A. Singh, A. Handa, N. Kumar, and S. K. Shukla, "Malware classification using image representation," in *Cyber Security Cryptography and Machine Learning* (S. Dolev, D. Hendler, S. Lodha, and M. Yung, eds.), (Cham), pp. 75–92, Springer International Publishing, 2019.

[14] K. He and D. Kim, "Malware detection with malware images using deep learning techniques," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 95–102, 2019.

[15] A. Khormali, A. Abusnaina, S. Chen, D. Nyang, and A. Mohaisen, "Copycat: Practical adversarial attacks on visualization-based malware detection," 2019.

[16] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013.

[17] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014.

[18] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 372–387, 2016.

[19] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS âĂŹ17*, 2017.

[20] H. S. Anderson, "Evading machine learning malware detection," 2017.

[21] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers: A case study on pdf malware classifiers," in *NDSS*, 2016.

[22] H. Dang, Y. Huang, and E.-C. Chang, "Evading classifiers by morphing in the dark," pp. 119–133, 10 2017.

[23] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on gan," 2017.

[24] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. OâĂŹReilly, "Adversarial deep learning for robust detection of binary encoded malware," 01 2018.

[25] H. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to evade static pe machine learning malware models via reinforcement learning," 01 2018.

[26] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving end-to-end deep learning malware detectors using adversarial examples," 2018.

[27] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," *2018 26th European Signal Processing Conference (EUSIPCO)*, Sep 2018.

[28] X. Liu, J. Zhang, Y. Lin, and H. Li, "Atmpa: attacking machine learning-based malware visualization detection methods via adversarial examples," pp. 1–10, 06 2019.

[29] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," pp. 39–57, 05 2017.

[30] Microsoft, "PE File Format." `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`. [Online; accessed 16-April-2020].

[31] dtm, "PE File Infection." `https://0x00sec.org/t/pe-file-infection/401`. [Online; accessed 16-April-2020].

[32] Athenian, "Another detailed guide to PE infection." `http://www.rohitab.com/discuss/topic/41510-another-detailed-guide-to-pe-infection`. [Online; accessed 16-April-2020].

[33] MalShare, "MalShare." `https://malshare.com/`. [Online; accessed 16-April-2020].

[34] D. Arp, M. Spreitzenbarth, H. M., G. H., and R. K., "Drebin: Effective and explainable detection of Android malware in your pocket," 2014.

[35] H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," *ArXiv e-prints*, Apr. 2018.

[36] "Classification of Malware with PE headers." `https://github.com/urwithajit9/ClaMP`. [Online; accessed 16-April-2020].

[37] UPX, "the Ultimate Packer for eXecutables." `https://upx.github.io/`. [Online; accessed 16-April-2020].

[38] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[39] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[40] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.

[41] S. Belongie, C. Carson, H. Greenspan, and J. Malik, "Color- and texture-based image segmentation using em and its application to content-based image retrieval," pp. 675–682, 02 1998.

[42] B. C. Society, "Code of Conduct." `https://cdn.bcs.org/bcs-org-media/2211/bcs-code-of-conduct.pdf`. [Online; accessed 19-April-2020].

[43] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 431–441, 2007.

[44] OpenAnalysis, "UNPACME." `https://www.unpac.me/#/`. [Online; accessed 19-April-2020].

[45] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

Appendix A

User Guide

Contents

This section explains how to run the code. First, the requirements are given, then it is explained how to run the File and Section Classifiers using the pre-prepared image dataset provided that was used in the report. Afterwards, how to run each file individually and an explanation on how to run the code on a new dataset is given. The code provided is split into

two areas, Data Preparation and Classification. Data Preparation involves splitting files into sections, converting files to images and creating Adversarial Variants. The Classification code involves running the File and Section Classifiers. As explained in Chapter 7, the PE Dataset was unable to be uploaded due to legal and ethical reasons, therefore any experiments involving Data Preparation will need to be tested using ones own dataset, this is explained at the end of the User Guide in A.4.

A.1 Requirements

The provided code requires Python 2 and Python 3. It has been tested with Python 3.6.9 and Python 2.7.17. Python 2 is used for generating Adversarial Variants with the files *fileInjection.py* and *sectionInjection.py*. Python 3 is used for all other files.

A.1.1 Python 2

1. pefile==2019.4.18

A.1.2 Python 3

1. pefile==2019.4.18
2. Keras==2.3.1
3. scipy==1.4.1
4. numpy==1.18.3
5. Pillow==7.1.1
6. statistics==1.0.3.5
7. sklearn==0.0
8. tensorflow==1.14.0
9. matplotlib==3.2.1

A.2 Running With Provided Dataset

The *data* directory of the project contains the goodware, malware and adversarial images of the dataset used in the report so the Classification can be tested with the dataset provided as explained next.

There are two different paths when running the code, the File Classification and Section Classification. Running the code in A.2.1 and A.2.2 will show how the results in the report are achieved.

Note that the -i flag in the fileClassifier and sectionClassifier files don't need to be used with the provided dataset as the default value is using the required value.

A.2.1 File Classification

```
python3 src/fileClassifier.py -g data/goodwareFileImages/ -m data/
    malwareFileImages/ -a data/adversarialFileImages/ -e 20
```

A.2.2 Section Classification

```
python3 src/sectionClassifier.py -g data/goodwareSectionImages/ -m data/
    malwareSectionImages/ -a data/adversarialSectionImages/ -e 20
```

A.3 Running Files Individually

A.3.1 fileSplitter.py

The fileSplitter file will take a directory of PE files and generate a new directory of directories, each new directory containing the section files for a specific file.

```
python3 src/fileSplitter.py -s /directory/of/pe/files/ -d /directory/to/write/to
    /
```

A.3.2 fileImageWriter.py

The fileImageWriter will take a directory of PE Files and create a new directory containing the images of these files. The file is capable of creating images of different sizes and RGB images, however for creating 32x32 greyscale images, the following is used.

```
python3 src/fileImageWriter.py -f /path/to/pe/files/ -g /path/to/write/greyscale
    /images/to/ -r 32
```

A.3.3 sectionImageWriter.py

The sectionImageWriter will take a nested directory of PE Files created by fileSplitter and create a new nested directory containing the images of these files. The file is capable of creating images of different sizes and RGB images, however for creating 32x32 greyscale images, the following is used.

```
python3 src/sectionImageWriter.py -f /path/to/section/pe/file/directory/ -g /
    path/to/write/greyscale/images/to/ -r 32
```

A.3.4 fileInjection.py

The fileInjection file will inject the data from the file into a new section of every file in the directory provided. The *infect* function was taken from https://axcheron.github.io/code-injection-with-python/

```
# python 2
python src/adversarial-variants/fileInjection.py /path/to/injection/file /path/
    to/files/to/infect/
```

A.3.5 sectionInjection.py

The sectionInjection file will inject the data from the file into 10 new sections of every file in a directory provided. The *infect* function was taken from https://axcheron.github.io/code-injection-with-python/

```
# python 2
python src/adversarial-variants/fileInjection.py /path/to/injection/file /path/
    to/files/to/infect/
```

A.3.6 fileClassifier.py

The fileClassifier is used for Malware Detection and testing the dataset. It requires a directory of goodware images, a directory of malware images, a directory of adversarial variants, and the name of the file used to infect the malware to create Adversarial Variants. The name to the injection file is required to ensure that the file is included in the training set.

Note that the injectionFilename needs to correspond to the .png file in the goodware image dataset, if the file used to create the Adversarial Variants was called example1.exe, this is likely to be example1.png.

```
python3 src/fileClassifier.py -g /path/goodwareFileImages/ -m /path/
    malwareFileImages/ -a /path/adversarialFileImages/ -e 20 -i
    injectionFilename
```

A.3.7 sectionClassifier.py

The sectionClassifier is used for Malware Detection using ALO and MV and testing the dataset. It requires a directory of directories containing goodware section images, a directory of directories containing malware section images, a directory of directories containing adversarial malware section images and the filename that contains the section that was used to create the adversarial variants.

Note that the injectionFilename needs to correspond to the directory in the goodware section image dataset, if the file used to create the Adversarial Variants was called example1.exe, this is likely to be example1.exe.

```
python3 src/sectionClassifier.py -g /path/goodwareSectionImages/ -m /path/
    malwareSectionImages/ -a /path/adversarialSectionImages/ -e 20 -i
    injectionFilename
```

A.4 Creating a New Dataset

Note that preparing a dataset requires creating new directories and will alter the filesystem, it is important to understand what changes are being made and why. It is advised to run each step yourself to understand what is happening, however two scripts to create a new file and section dataset have been provided. When running the scripts a few changes need to be made to the file as will be explained next. Running the Classifiers with the dataset provided requires a single command shown in A.2.1 and A.2.2 of the User Guide. An example of how to run the code on a new dataset is a long and involves process as explained next.

A.4.1 File Classification

1. Create Directory of Goodware PE Files
2. Create Directory of Malware PE Files
3. Copy Malware Files to Adversarial Directory

4. Create File Adversarial Variants of Malware using *fileInjection.py*

5. Create Images for the three directories (Goodware, Malware, Adversarial Variants) using *fileImageWriter.py*

6. Run File Classifier *fileClassifier.py*

First, two directories are required to hold the Malware and Goodware dataset. Copy the Malware files to a new Adversarial File directory.

```
# create directories for the files
mkdir /path/goodwareFiles
mkdir /path/malwareFiles
# copy the dataset you are using into goodwareFiles and malwareFiles before
    carrying on

# copy malware into adversarial variants, note that the adversarial variants
    will be altered.
cp -R /path/malwareFiles /path/adversarialFiles
```

After this, three lines in the *createfiledata* script need to be altered. The datadir needs to contain the path to the directories just created. The projectpath variable needs to contain the path to the project root. The infectionFilename needs to be the name of the file that will be used to infect the Malware to create Adversarial Variants.

```
datadir="/path/"
projectpath="/home/user/malwaredetection/"
infectionFilename="example.exe"
```

A.4.2 Section Classification

1. Create Directory of Goodware PE Files

2. Create Directory of Malware PE Files

3. Copy Malware Files to Adversarial Directory

4. Create Section Adversarial Variants of Malware using *sectionInjection.py*

5. Create Section Directories for Goodware, Malware and Adversarial Variants

6. Split each of the directories into section files using *fileSplitter.py*

7. Create Image Directories for Goodware, Malware and Adversarial Variants

8. Create Images for the three directories (Goodware, Malware, Adversarial Variants) using *sectionImageWriter.py*

9. Run Section Classifier *sectionClassifier.py*

First, two directories are required to hold the Malware and Goodware dataset. Copy the Malware files to a new Adversarial File directory.

```
# create directories for the files
mkdir /path/goodwareFiles
mkdir /path/malwareFiles
# copy the dataset you are using into goodwareFiles and malwareFiles before
    carrying on

# copy malware into adversarial variants, note that the adversarial variants
    will be altered.
cp -R /path/malwareFiles /path/adversarialFiles
```

After this, three lines in the *createsectiondata* script need to be altered. The datadir needs to contain the path to the directories just created. The projectpath variable needs to contain the path to the project root. The infectionFilename needs to be the name of the file that will be used to infect the Malware to create Adversarial Variants.

```
datadir="/path/"
projectpath="/home/user/malwaredetection/"
infectionFilename="example.exe"
```

Appendix B

Source Code

Contents

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

Jordan Joe Watson

21/04/2020

B.1 Data Preparation

B.1.1 fileSplitter.py

This file will convert a directory of PE Files to a nested directory of section files.

```
import pefile
from os import listdir, system
from os.path import isfile, join, getsize
from numpy import array
from PIL import Image
import os
from statistics import mean, stdev
import numpy as np
import argparse

"""
File that takes a PE file and splits it into individual section files
-s Src directory of files to split
-d Dst directory to store files

reads from a directory structure of
/dir/file1
/dir/file2
...

writes to a directory structure of
/dir/file1/file1section1
/dir/file1/file1section2
...

Malware and Goodware should be handled seperately to keep the directories
    serparate
"""

parser = argparse.ArgumentParser()

parser.add_argument("-s", "--src", default=None, required=True,
                    help="Src directory to convert")
parser.add_argument("-d", "--dst", default=None, required=True,
                    help="Dst directory to convert")

args = parser.parse_args()
```

```
37
38 # read in all files in src directory
39 files = [f for f in listdir(args.src) if isfile(join(args.src, f))]
40
41 """
42 For every file in src directory, create new directory. Read in the file
43 for everysection in the file, read the data, create a new file and
44 write in dst directory
45 """
46 for f in files:
47     createDirCmd = 'mkdir {}{}'.format(args.dst, f)
48     system(createDirCmd)
49     pe = pefile.PE(args.src + f)
50     i = 0
51     for s in pe.sections:
52         sdata = s.get_data()
53
54         newfilename = "{}_{}".format(str(i), f)
55
56         _f = open('{}{}/{}'.format(args.dst, f, newfilename), "wb")
57         _f.write(sdata)
58         _f.close()
59         i += 1
```

B.1.2 fileImageWriter.py

The fileImageWriter file will convert a directory of PE files to a directory of images.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib import colors
4 import matplotlib
5 import os
6 import array
7 import numpy as np
8 from scipy import misc
9 from PIL import Image
10 from math import ceil
11 import argparse
12 from os import listdir
13 from os.path import isfile, join
14
```

```
15 """
16 This file is for converting a directory of files to images. The
       sectionImageWriter
17 file is for converting a nested directory structure of exes into a nested
18 directory structure of imgs, like created in the fileSplitter file.
19
20 provide grey filepath if you want to write grey files, and color if you want
21 color files. Size is the maximum filesize, so if your maximum filesize is 10MB
22 but you want it to be 11MB with empty space the size argument will add this
23 extra space. Otherwise each file will be only as large as it needs to be to fit
24 in the data. Resize is for deciding how large you want the files, e.g. i'm
25 working with a 32 bit CNN so all of mine willl be -r 32.
26 """
27
28 WIDTH = 384
29
30 parser = argparse.ArgumentParser()
31 parser.add_argument("-g", "--grey", default=None,
32                     help="path to save grey files")
33 parser.add_argument("-c", "--color", default=None,
34                     help="path to save color files")
35 parser.add_argument("-s", "--size", type=int, default=None,
36                     help="maximum bytes for image file")
37 parser.add_argument("-f", "--files", default=None, required=True,
38                     help="directory containing files to convert")
39 parser.add_argument("-r", "--resize", default=None, type=int,
40                     help="values to resize to, e.g. -r 32")
41 args = parser.parse_args()
42
43
44 def generate_black_image(n,m):
45     """
46     Creates a black png image to write file to
47
48     Args:
49         n (int): width of image to create
50         m (int): height of image to write
51     """
52     img = Image.new('RGB', (n,m), (0, 0, 0))
53     img.save("null.png", "PNG")
54
55 def generate_colormap():
```

```
56      """
57      Creates a colormap to use when creating RGB color images of files
58
59      Returns:
60          im: colormap
61          x: values
62      """
63      x = np.array([ np.arange(x,x+16)/256 for x in range(0,256,16) ])
64      im = plt.imshow(x, cmap=plt.cm.nipy_spectral)
65
66      # plt.show()
67      return im, x
68
69
70  def write_bytes_to_file(filepath, filename, cmap, vals, size, colorPath,
        greyPath, resize, colDct):
71      """
72      Write bytes to an image file
73
74      Args:
75          filepath (string): filepath to directory to read file from
76          dirname (string): name of file to use as new directory name containing
        section images
77          filename (string): filename of section file
78          size (int): maximum size if specified, otherwise size of file is used
79          colorPath (string): path to directory to store color images
80          greyPath (string): path to directory to store grey images
81          resize (int): size to resize images to, e.g. resizexresize
82          colDct (color map dictionary): dictionary storing color map for RGB
83      """
84      f = open(filepath + filename, 'rb')
85      bytes = f.read()
86
87      if not size:
88          size = len(bytes)
89          if size == 0:
90              return
91          generate_black_image(WIDTH, int(ceil(size/WIDTH)))
92
93      elif size > len(bytes):
94          size = len(bytes)
95
```

```
    nullImageColor = None
    nullImageGrey = None
    colorPixels = None
    greyPixels = None

    if greyPath:
        nullImageGrey = Image.open("null.png")
        greyPixels = nullImageGrey.load()
    if colorPath:
        nullImageColor = Image.open("null.png")
        colorPixels = nullImageColor.load()

    byteCounter = 0

    for i in range(0, int(ceil(size/WIDTH))):
        for j in range(0,WIDTH):
            if byteCounter == size-1: break
            byte = hex(bytes[byteCounter])

            if greyPath:
                greyRGBvalue = int(byte,16)
                greyPixels[j, i] = (greyRGBvalue, greyRGBvalue, greyRGBvalue)
            if colorPath:
                x,y = 0,0
                if len(byte) == 3:
                    x,y = 0, int(byte[2], 16)
                else:
                    x, y = int(byte[2],16), int(byte[3], 16)
                # 0,0 is top left. 15,0 is top right. x is correct y is flipped.
                colorPixels[j, i] = colDct[x][y]
            byteCounter += 1

    filename = filename[:-4] + ".png" if filename[-4:] == '.exe' else filename +
     ".png"
    if greyPath:
        if resize:
            nullImageGrey = nullImageGrey.resize((resize,resize))
        nullImageGrey.save(greyPath + filename)
    if colorPath:
        if resize:
            nullImageColor = nullImageColor.resize((resize, resize))
        nullImageColor.save(colorPath + filename)
```

```
137
138
139 if args.size:
140     generate_black_image(WIDTH, int(ceil(args.size/WIDTH)))
141
142 cmap, vals = generate_colormap()
143 # read in filepaths
144 colDct = {0 : {},1 : {},2 :
        {},3:{},4:{},5:{},6:{},7:{},8:{},9:{},10:{},11:{},12:{},13:{},14:{},15:{}}
145 for i in range(0,16):
146     for j in range(0,16):
147         r,g,b,_ = cmap.to_rgba(vals[j, i])
148         if not colDct[i].get(j):
149             colDct[i][j] = (int(r*255),int(g*255),int(b*255))
150
151 files = [f for f in listdir(args.files) if isfile(join(args.files, f))]
152
153 for f in files:
154     write_bytes_to_file(args.files, f, cmap, vals, args.size, args.color, args.
        grey, args.resize, colDct)
```

B.1.3 sectionImageWriter.py

The sectionImageWriter file will convert a nested directory of PE files (section files) to a nested directory of section images.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib import colors
4 import matplotlib
5 import os
6 import array
7 import numpy as np
8 from scipy import misc
9 from PIL import Image
10 from math import ceil
11 import argparse
12 from os import listdir, system
13 from os.path import isfile, join, isdir
14
15 """
16 This file is for converting a nested directory structure of exes into a nested
```

```
directory structure of imgs, like created in the fileSplitter file. The other
fileImageWriter file is for converting a directory of files to a directory of
    images

provide grey filepath if you want to write grey files, and color if you want
color files. Size is the maximum filesize, so if your maximum filesize is 10MB
but you want it to be 11MB with empty space the size argument will add this
extra space. Otherwise each file will be only as large as it needs to be to fit
in the data. Resize is for deciding how large you want the files, e.g. i'm
working with a 32 bit CNN so all of mine willl be -r 32.
"""

WIDTH = 384

parser = argparse.ArgumentParser()
parser.add_argument("-g", "--grey", default=None,
                    help="path to save grey files")
parser.add_argument("-c", "--color", default=None,
                    help="path to save color files")
parser.add_argument("-s", "--size", type=int, default=None,
                    help="maximum bytes for image file")
parser.add_argument("-f", "--files", default=None, required=True,
                    help="directory containing files to convert")
parser.add_argument("-r", "--resize", default=None, type=int,
                    help="values to resize to, e.g. -r 32")
args = parser.parse_args()

def generate_black_image(n,m):
    """
    Creates a black png image to write file to

    Args:
        n (int): width of image to create
        m (int): height of image to write
    """
    img = Image.new('RGB', (n,m), (0, 0, 0))
    img.save("null.png", "PNG")

def generate_colormap():
    """
    Creates a colormap to use when creating RGB color images of files

```

```
58      Returns:
59          im: colormap
60          x: values
61      """
62      x = np.array([ np.arange(x,x+16)/256 for x in range(0,256,16) ])
63      im = plt.imshow(x, cmap=plt.cm.nipy_spectral)
64      #plt.show()
65      return im, x
66
67  def write_bytes_to_file(filepath, dirname, filename, size, colorPath, greyPath,
        resize, colDct):
68      """
69      Write bytes to an image file
70
71      Args:
72          filepath (string): filepath to directory to read file from
73          dirname (string): name of file to use as new directory name containing
        section images
74          filename (string): filename of section file
75          size (int): maximum size if specified, otherwise size of file is used
76          colorPath (string): path to directory to store color images
77          greyPath (string): path to directory to store grey images
78          resize (int): size to resize images to, e.g. resizexresize
79          colDct (color map dictionary): dictionary storing color map for RGB
80      """
81
82      path = '{}/{}/{}'.format(filepath,dirname,filename)
83      f = open(path, 'rb')
84      bytes = f.read()
85
86      if not size:
87          size = len(bytes)
88          if size == 0:
89              return
90          generate_black_image(WIDTH, int(ceil(size/WIDTH)))
91
92      elif size > len(bytes):
93          size = len(bytes)
94
95      # create new directory for files
96      cmd = 'mkdir {}/{}'.format(greyPath, dirname)
97      system(cmd)
```

```
98
99      nullImageColor = None
100     nullImageGrey = None
101     colorPixels = None
102     greyPixels = None
103
104     if greyPath:
105         nullImageGrey = Image.open("null.png")
106         greyPixels = nullImageGrey.load()
107     if colorPath:
108         nullImageColor = Image.open("null.png")
109         colorPixels = nullImageColor.load()
110
111     byteCounter = 0
112
113     for i in range(0, int(ceil(size/WIDTH))):
114         for j in range(0,WIDTH):
115             if byteCounter == size-1: break
116             byte = hex(bytes[byteCounter])
117
118             if greyPath:
119                 greyRGBvalue = int(byte,16)
120                 greyPixels[j, i] = (greyRGBvalue, greyRGBvalue, greyRGBvalue)
121             if colorPath:
122                 x,y = 0,0
123                 if len(byte) == 3:
124                     x,y = 0, int(byte[2], 16)
125                 else:
126                     x, y = int(byte[2],16), int(byte[3], 16)
127                 # 0,0 is top left. 15,0 is top right. x is correct y is flipped.
128                 colorPixels[j, i] = colDct[x][y]
129             byteCounter += 1
130
131     filename = filename[:-4] + ".png" if filename[-4:] == '.exe' else filename +
         ".png"
132     if greyPath:
133         saveTo = '{}/{}/{}'.format(greyPath, dirname, filename)
134         if resize:
135             nullImageGrey = nullImageGrey.resize((resize,resize))
136         nullImageGrey.save(saveTo)
137     if colorPath:
138         if resize:
```

```
            nullImageColor = nullImageColor.resize((resize, resize))
        nullImageColor.save(colorPath + filename)

if args.size:
    generate_black_image(WIDTH, int(ceil(args.size/WIDTH)))

colDct = {}
if args.color:
    cmap, vals = generate_colormap()
    # generate colordict once so it doesn't get recomputed
    colDct = {0 : {},1 : {},2 : {},3:{},4:{},5:{},6:{},7:{},8:{},9:{},10:{},11:{},12:{},13:{},14:{},15:{}}
    for i in range(0,16):
        for j in range(0,16):
            r,g,b,_ = cmap.to_rgba(vals[j, i])
            if not colDct[i].get(j):
                colDct[i][j] = (int(r*255),int(g*255),int(b*255))

# get directories for files
dir = [f for f in listdir(args.files) if isdir(join(args.files, f))]

# for each filedirectory in dir, e.g. each file1.exe dir representing sections
for d in dir:
    # get all section files
    files = [ f for f in listdir(args.files + d) if isfile(join(args.files + d, f))]
    # for each section, write bytes to image file
    for fl in files:
        write_bytes_to_file(args.files, d, fl, args.size, args.color, args.grey, args.resize, colDct)
```

B.1.4 fileInjection.py

This file requires Python2. This file is for infecting a file with data. Note that the *infect* function was taken from https://axcheron.github.io/code-injection-with-python/

```
import pefile
import mmap
import os
from os import listdir
from os.path import join, isdir, isfile
from sys import argv
```

```
7
8  """
9  File to inject a binary into a new section 10 times of all files in a directory
10
11 Code for the infect function taken from: https://axcheron.github.io/code-
       injection-with-python/
12
13 First argument should be a file containing data used to infect other files. The
14 second argument should be a filepath for a directory containing PE files that
15 will be infected with the first file 10 times. Make sure you have a backup of
16 these files before infecting
17 """
18
19 if len(argv) != 3:
20     print("Error: Incorrect arguments")
21     print("python fileInjection.py <fileToInject> <directoryOfFiles>")
22     exit(1)
23
24 f = open(argv[1], "rb")
25 shellcode = f.read()
26
27 def align(val_to_align, alignment):
28     return ((val_to_align + alignment - 1) / alignment) * alignment
29
30 def infect(exe_path, shellcode):
31     """
32     Function to infect a file with other data into a new section
33
34     Args:
35         exe_path (string): file path used for a file to infect
36         shellcode (byte string): byte string containing the data to use to
       infect a file
37     """
38
39     original_size = os.path.getsize(exe_path)
40
41     fd = open(exe_path, 'a+b')
42     map = mmap.mmap(fd.fileno(), 0, access=mmap.ACCESS_WRITE)
43     map.resize(original_size + 0x1000)
44     map.close()
45     fd.close()
46
```

```
47      pe = pefile.PE(exe_path)
48
49      number_of_section = pe.FILE_HEADER.NumberOfSections
50      last_section = number_of_section - 1
51      file_alignment = pe.OPTIONAL_HEADER.FileAlignment
52      section_alignment = pe.OPTIONAL_HEADER.SectionAlignment
53      new_section_offset = (pe.sections[number_of_section - 1].get_file_offset() +
         40)
54
55      # Look for valid values for the new section header
56      raw_size = align(0x1000, file_alignment)
57      virtual_size = align(0x1000, section_alignment)
58      raw_offset = align((pe.sections[last_section].PointerToRawData +
59                          pe.sections[last_section].SizeOfRawData),
60                         file_alignment)
61
62      virtual_offset = align((pe.sections[last_section].VirtualAddress +
63                              pe.sections[last_section].Misc_VirtualSize),
64                             section_alignment)
65
66      # CODE | EXECUTE | READ | WRITE
67      characteristics = 0xE0000020
68      # Section name must be equal to 8 bytes
69      name = ".data" + (3 * '\x00')
70
71      # Create the section
72      # Set the name
73      pe.set_bytes_at_offset(new_section_offset, name)
74      # Set the virtual size
75      pe.set_dword_at_offset(new_section_offset + 8, virtual_size)
76      # Set the virtual offset
77      pe.set_dword_at_offset(new_section_offset + 12, virtual_offset)
78      # Set the raw size
79      pe.set_dword_at_offset(new_section_offset + 16, raw_size)
80      # Set the raw offset
81      pe.set_dword_at_offset(new_section_offset + 20, raw_offset)
82      # Set the following felds to zero
83      pe.set_bytes_at_offset(new_section_offset + 24, (12 * '\x00'))
84      # Set the characteristics
85      pe.set_dword_at_offset(new_section_offset + 36, characteristics)
86
87      # Modify the Main Headers
```

```
88      pe.FILE_HEADER.NumberOfSections += 1
89      pe.OPTIONAL_HEADER.SizeOfImage = virtual_size + virtual_offset
90      pe.write(exe_path)
91      pe = pefile.PE(exe_path)
92      number_of_section = pe.FILE_HEADER.NumberOfSections
93      last_section = number_of_section - 1
94      new_ep = pe.sections[last_section].VirtualAddress
95      raw_offset = pe.sections[last_section].PointerToRawData
96
97      pe.set_bytes_at_offset(raw_offset, shellcode)
98
99      pe.write(exe_path)
100
101 path = argv[2]
102 files = ([path + d for d in listdir(path) if isfile(join(path, d))])
103 # for every file to infect, run infection function
104 for f in files:
105     infect(f, shellcode)
```

B.1.5 sectionInjection.py

This file requires Python2. This file is for infecting a file with 10 new sections. Note that the *infect* function was taken from https://axcheron.github.io/code-injection-with-python/

```
1  import pefile
2  import mmap
3  import os
4  from os import listdir
5  from os.path import join, isdir, isfile
6  from sys import argv
7
8  """
9  File to inject a binary into a new section 10 times of all files in a directory
10
11 Code for the infect function taken from: https://axcheron.github.io/code-
       injection-with-python/
12
13 First argument should be a file containing data used to infect other files. The
14 second argument should be a filepath for a directory containing PE files that
15 will be infected with the first file 10 times. Make sure you have a backup of
16 these files before infecting
17 """
```

```
18
19
20 if len(argv) != 3:
21     print("Error: Incorrect arguments")
22     print("python sectionInjection.py <fileToInject> <directoryOfFiles>")
23     exit(1)
24
25 f = open(argv[1], "rb")
26 shellcode = f.read()
27
28 def align(val_to_align, alignment):
29     return ((val_to_align + alignment - 1) / alignment) * alignment
30
31 def infect(exe_path, shellcode):
32     """
33     Function to infect a file with other data into a new section
34
35     Args:
36         exe_path (string): file path used for a file to infect
37         shellcode (byte string): byte string containing the data to use to
       infect a file
38     """
39     original_size = os.path.getsize(exe_path)
40
41     fd = open(exe_path, 'a+b')
42     map = mmap.mmap(fd.fileno(), 0, access=mmap.ACCESS_WRITE)
43     map.resize(original_size + 0x1000)
44     map.close()
45     fd.close()
46
47     pe = pefile.PE(exe_path)
48     number_of_section = pe.FILE_HEADER.NumberOfSections
49     last_section = number_of_section - 1
50     file_alignment = pe.OPTIONAL_HEADER.FileAlignment
51     section_alignment = pe.OPTIONAL_HEADER.SectionAlignment
52     new_section_offset = (pe.sections[number_of_section - 1].get_file_offset() +
        40)
53
54     # Look for valid values for the new section header
55     raw_size = align(0x1000, file_alignment)
56     virtual_size = align(0x1000, section_alignment)
57     raw_offset = align((pe.sections[last_section].PointerToRawData +
```

```
58                          pe.sections[last_section].SizeOfRawData),
59                         file_alignment)
60
61     virtual_offset = align((pe.sections[last_section].VirtualAddress +
62                             pe.sections[last_section].Misc_VirtualSize),
63                            section_alignment)
64
65     # CODE | EXECUTE | READ | WRITE
66     characteristics = 0xE0000020
67     # Section name must be equal to 8 bytes
68     name = ".data" + (3 * '\x00')
69
70     pe.set_bytes_at_offset(new_section_offset, name)
71     pe.set_dword_at_offset(new_section_offset + 8, virtual_size)
72     pe.set_dword_at_offset(new_section_offset + 12, virtual_offset)
73     pe.set_dword_at_offset(new_section_offset + 16, raw_size)
74     pe.set_dword_at_offset(new_section_offset + 20, raw_offset)
75     pe.set_bytes_at_offset(new_section_offset + 24, (12 * '\x00'))
76     pe.set_dword_at_offset(new_section_offset + 36, characteristics)
77
78     pe.FILE_HEADER.NumberOfSections += 1
79     pe.OPTIONAL_HEADER.SizeOfImage = virtual_size + virtual_offset
80     pe.write(exe_path)
81
82     pe = pefile.PE(exe_path)
83     number_of_section = pe.FILE_HEADER.NumberOfSections
84     last_section = number_of_section - 1
85     new_ep = pe.sections[last_section].VirtualAddress
86
87     raw_offset = pe.sections[last_section].PointerToRawData
88     pe.set_bytes_at_offset(raw_offset, shellcode)
89     # print "\t[+] Shellcode wrote in the new section"
90
91     pe.write(exe_path)
92
93 path = argv[2]
94 files = ([path + d for d in listdir(path) if isfile(join(path, d))])
95
96 # for every file to infect, infect it 10 times
97 for f in files:
98     for i in range(0,10):
99         infect(f, shellcode)
```

B.1.6 etc.py

This file contains miscellaneous functions.

```
from os import listdir
from os.path import join
from numpy import array
from PIL import Image

"""
Miscellaneous functions
"""

def read_files(dirs, path):
    """
    read all image files in a directory into an array

    Args:
        dirs (array): array of directories to read files from
        path (string): filepath to use to read dirs from

    Returns:
        array of image vectors
    """
    imgs = []
    for d in dirs:
        sectionFiles = listdir(join(path + d))
        imgs += [ array(Image.open("{}{}/{}".format(path,d,f))) for f in
    sectionFiles ]
    return imgs

def read_file(file, path):
    """
    read image file in a directory into an array

    Args:
        file (string): filename of file to read in
        path (string): path to read the file from

    Returns:
        image vector
    """
    imgs = []
```

```
    sectionFiles = listdir(join(path + file))
    imgs += [ array(Image.open("{}{}/{}".format(path,file,f))) for f in
    sectionFiles ]
    return imgs

# pass in confusion matrix in the form [ tp, fp, fn, tn ]
def accuracy(ls):
    """
    Confusion matrix of list

    Args:
        ls (list): list of [TP, FP, FN, TN]

    Returns:
        accuracy of confusion matrix
    """
    return (ls[0]+ls[-1])/sum(ls)
```

B.2 Classifiers

B.2.1 CNN.py

This file holds a function to load the Keras Model.

```
from keras.utils import to_categorical
from keras.models import Sequential, Model, load_model
from keras.layers import Conv2D, MaxPool2D, LeakyReLU, Flatten,  Dense
from keras import layers
from keras.callbacks import EarlyStopping, ModelCheckpoint, Callback

"""
Creates the model structure for the Classification and Adversarial Variant
tasks. The model is similar in sizing to VGG-16 but reduced to 32x32
Images.
"""

def create_model(e, savemodel, X_train, y_train, X_test, y_test, outputs=2):
    """
    Creates the CNN Model used in the research

    Args:
        e (int): epochs to use for training
```

```
19          savemodel (string): file to save the best model found through training
20          X_train (array): training data array
21          y_train (array): training results array
22          X_test (array): testing data array
23          y_test (array): testing results array
24          outputs (int): amount of outputs for the results, e.g. 2 for binary
    classification
25
26      Returns:
27          Keras CNN Model
28      """
29      model = Sequential()
30      model.add(Conv2D(64, kernel_size=(3,3), strides=(1,1),activation='linear',
    input_shape=(32,32,3),padding='same'))
31      model.add(MaxPool2D((2,2),padding='same'))
32      model.add(LeakyReLU(alpha=0.1))
33      model.add(Conv2D(128, (3,3), strides=(1,1), activation='linear',padding='
    same'))
34      model.add(MaxPool2D(pool_size=(2, 2),padding='same'))
35      model.add(LeakyReLU(alpha=0.1))
36      model.add(Conv2D(128, (3, 3), strides=(1,1), activation='linear',padding='
    same'))
37      model.add(MaxPool2D(pool_size=(2, 2),padding='same'))
38      model.add(LeakyReLU(alpha=0.1))
39      model.add(Flatten())
40      model.add(Dense(2048, activation='linear'))
41      model.add(LeakyReLU(alpha=0.1))
42      model.add(Dense(outputs, activation='softmax'))
43
44      callbacks = [EarlyStopping(monitor='val_loss', mode='min', verbose=1,
    patience=40)]
45      if savemodel:
46          callbacks.append(ModelCheckpoint('{}'.format(savemodel), save_best_only=
    True, monitor='val_accuracy', mode='max'))
47
48      model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['
    accuracy'])
49      model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=e,
    callbacks=callbacks)
50      model = load_model(savemodel)
51      return model
```

B.2.2 evaluation.py

```
from etc.etc import read_file
from numpy import array

"""
Evaluation functions and code for testing new detection methods
ALO (at_least_one_binary_classification)
MV (majority_vote_binary_classification)
"""

MALWARE = 1
GOODWARE = 0

def at_least_one_binary_classification(y_pred):
    """
    If at least one is malicious, the entire file is classified as malicious
    Only for binary classification

    Args:
        y_pred (array): array of section classifications for a file
    Pass in a list of section classifications belonging to same file

    Returns:
        (int): Classification of file
    """
    for cls in y_pred:
        if cls[MALWARE] > (cls[GOODWARE]):
            return MALWARE # Malware
    return GOODWARE # Goodware



def majority_vote_binary_classification(y_pred):
    """
    The majority of votes determines the classification
    Pass in a list of section classifications belonging to same file

    Args:
        y_pred (array): array of section classifications for a file

    Returns:
```

```
        malwareLikelihood (float): probability between 0. and 1. of
                                   predictions being Malware
    """

    binaryPredictions = (y_pred == y_pred.max(axis=1)[:, None]).astype(int).sum(
    axis=0)
    malwareLikelihood = (binaryPredictions[1]) / (sum(binaryPredictions))
    return malwareLikelihood


def predict(fs, dir, classifier):
    """
    Tests MV and ALO Evaluation with files in directory with classifier

    Args:
        fs (string): array of filenames to test
        dir (string): path to directory containing fs
        classifier (Keras Model): model to classify data from dir and fs

    Returns:
        tuple of ALO and MV predictions for the files
    """
    alo_predictions = []
    mv_predictions = []

    for file in fs:
        imgs = read_file(file, dir)
        cls = classifier.predict(array(imgs))
        alo = at_least_one_binary_classification(cls)
        mv = majority_vote_binary_classification(cls)

        alo_predictions.append(alo)
        mv_predictions.append(mv)

    return (alo_predictions, mv_predictions)
```

B.2.3 fileClassifier.py

This file is for running the File Classifier.

```
from os import listdir, stat
from os.path import isfile, join
from numpy import array
```

```
4 from PIL import Image
5 from sklearn.model_selection import train_test_split
6 from sklearn.utils import shuffle
7 from sklearn.metrics import confusion_matrix, roc_curve, auc
8 import numpy as np
9 import argparse
10 from classifiers.CNN import create_model
11 from keras.utils import to_categorical
12 from etc.etc import accuracy
13
14 """
15 File for Classification and Adversarial Variants based on previous research
16 ideas, i.e. not using sections for classification
17
18 File Classifier
19 """
20
21 parser = argparse.ArgumentParser()
22
23 parser.add_argument("-g", "--goodware", default=None, required=True,
24                     help="directory containing goodware file directories
       containing images")
25 parser.add_argument("-m", "--malware", default=None, required=True,
26                     help="directory containing goodware file directories
       containing images")
27 parser.add_argument("-a", "--advars", default=None, required=True,
28                     help="directory containing adversarial variants")
29 parser.add_argument("-i", "--infection", default="MicrosoftEdge.png",
30                     help="name of file used to infect the Malware, ensures this
       image is included in dataset")
31 parser.add_argument("-s", "--savemodel", default='.tmpfile',
32                     help="file to save model")
33 parser.add_argument("-e", "--epochs", type=int, default=4,
34                     help="epochs for CNN")
35 parser.add_argument("-r", "--results", default=None,
36                     help="file to save results")
37
38 args = parser.parse_args()
39
40 # read files
41 gwfiles = [f for f in listdir(args.goodware) if isfile(join(args.goodware, f))
       if f != args.infection]
```

```
mwfiles = [f for f in listdir(args.malware) if isfile(join(args.malware, f))]

# get 20 percent of each for testing
gwX_train, gwX_test, gwy_train, gwy_test = train_test_split(gwfiles, [0] * len(
    gwfiles), test_size=0.2)
mwX_train, mwX_test, mwy_train, mwy_test = train_test_split(mwfiles, [1] * len(
    mwfiles), test_size=0.2)

gwX_train.append(args.infection)
mwAdv = [ array(Image.open('{}{}'.format(args.advars, mwfile))) for mwfile in
    mwX_test ]

gwX_train = [ array(Image.open(args.goodware + gwfile)) for gwfile in gwX_train
    ]
gwy_train = [0] * len(gwX_train)
gwX_test = [ array(Image.open(args.goodware + gwfile)) for gwfile in gwX_test ]
gwy_test = [0] * len(gwX_test)
mwX_train = [ array(Image.open(args.malware + mwfile)) for mwfile in mwX_train ]
mwy_train = [1] * len(mwX_train)
mwX_test = [ array(Image.open(args.malware + mwfile)) for mwfile in mwX_test ]
mwy_test = [1] * len(mwX_test)

X_train = np.array(gwX_train + mwX_train) /255
X_test = np.array(gwX_test + mwX_test) / 255
y_train = gwy_train + mwy_train
y_test = gwy_test + mwy_test
X_train, y_train = shuffle(X_train, y_train)
# Adversarial Variants
mwAdv = np.array(mwAdv)/255
y_train = to_categorical(y_train)
y_test_cat = to_categorical(y_test)

model = create_model(args.epochs, args.savemodel, X_train, y_train, X_test,
    y_test_cat)

x = model.predict(X_test)
x = [ 0 if c[0] > c[1] else 1 for c in x ]
advResults = model.predict(mwAdv)
mwAdvResults = [ 0 if c[0] > c[1] else 1 for c in advResults ]
advFN = mwAdvResults.count(0)
ttl = len(mwAdv)

```

```
(tn, fp, fn, tp) = (confusion_matrix(y_test, x).ravel())
print("File Classifier Accuracy: " + str(accuracy([ tp, fp, fn, tn ])))

#WRITE TO FILE
if args.results:
    results = [
        accuracy([tp, fp, fn, tn]),
        tp, fp, fn, tn,
        advFN
    ]
    f = open(args.results, "a")
    f.write("\n")
    f.write(', '.join([ str(res) for res in results ]))
    f.close()

print("False Negatives: " + str(fn/ttl))
print("Adversarial Variant False Negatives: " + str(advFN/ttl))
```

B.2.4 sectionClassifier.py

This file is for running the Section Classifiers, Section (No Evaluation), ALO and MV.

```
from classifiers.CNN import create_model
from classifiers.evaluation import predict
from etc.etc import read_file, read_files, accuracy
import argparse
from os import listdir
from os.path import join, isdir
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix, auc, roc_curve
from keras.utils import to_categorical
from numpy import array, arange, count_nonzero, argmax
from matplotlib import pyplot as plt
from PIL import Image

"""
File for Section Classification and Adversarial Variants of Section Classifiers

Section Classification (No Evaluation)
At Least One Classifier
Majority Vote Classifier
```

```
"""

MALWARE = 1
GOODWARE = 0

parser = argparse.ArgumentParser()

parser.add_argument("-g", "--goodware", default=None, required=True,
                    help="directory containing goodware file directories
    containing images")
parser.add_argument("-m", "--malware", default=None, required=True,
                    help="directory containing goodware file directories
    containing images")
parser.add_argument("-a", "--advars", default=None, required=True,
                    help="directory containing adversarial variant file
    directories containing images")
parser.add_argument("-i", "--infection", default="ApplySettingsTemplateCatalog.
    exe",
                    help="name of file the section was taken from that was used
    to infect the Malware, this name must match the directory name holding the
    section files")
parser.add_argument("-s", "--savemodel", default='.tmpfile',
                    help="file to save model")
parser.add_argument("-e", "--epochs", type=int, default=4,
                    help="epochs for CNN")
parser.add_argument("-r", "--results", default=None,
                    help="file to save model")

args = parser.parse_args()

# Read in directories containing files
goodware_directories = ([d for d in listdir(args.goodware) if isdir(join(args.
    goodware, d)) if d != args.infection])
malware_directories = ([d for d in listdir(args.malware) if isdir(join(args.
    malware, d))])
gwX_train, gwX_test, gwy_train, gwy_test = train_test_split(goodware_directories
    , (len(goodware_directories) * [GOODWARE]), test_size=0.2)
mwX_train, mwX_test, mwy_train, mwy_test = train_test_split(malware_directories,
     (len(malware_directories) * [MALWARE]), test_size=0.2)

# read in data from testing files to train the section classifier
gwX_train.append(args.infection)
```

```
53 gwX_train_imgs = read_files(gwX_train, args.goodware)
54 mwX_train_imgs = read_files(mwX_train, args.malware)
55 # normalize into 0 -> 1 values instead of RGB
56 imgs = array(gwX_train_imgs + mwX_train_imgs) / 255
57 # classification for the imgs
58 classifications = (len(gwX_train_imgs) * [GOODWARE]) + (len(mwX_train_imgs) * [
       MALWARE])
59 X_train, y_train = shuffle(imgs, classifications)
60 y_train = to_categorical(y_train)
61
62 # same as above, but for test data for the section classifier
63 gwX_test_imgs = read_files(gwX_test, args.goodware)
64 mwX_test_imgs = read_files(mwX_test, args.malware)
65 imgs = array(gwX_test_imgs + mwX_test_imgs) / 255
66 classifications = (len(gwX_test_imgs) * [GOODWARE]) + (len(mwX_test_imgs) * [
       MALWARE])
67 X_test, y_test = shuffle(imgs, classifications)
68 y_test = to_categorical(y_test)
69
70 # CREATE CNN MODEL
71 model = create_model(args.epochs, args.savemodel, X_train, y_train, X_test,
       y_test)
72
73 y_test = [ 1 if res[1] == 1 else 0 for res in y_test ]
74 y_pred = model.predict(X_test)
75 y_pred = [ argmax(y) for y in y_pred ]
76
77 # Sectino Confusion Matrix
78 tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
79 section_confusion_matrix = [ tp, fp, fn, tn ]
80
81 # get predictions on test data for mv and alo
82 alo_gw_predictions, mv_gw_predictions = predict(gwX_test, args.goodware, model)#
       , bias)
83 alo_mw_predictions, mv_mw_predictions = predict(mwX_test, args.malware, model)#
       , bias)
84
85 # ALO confusion Matrix
86 # tp, fp, fn, tn
87 alo_confusion_matrix = [
88     alo_mw_predictions.count(MALWARE),
89     alo_gw_predictions.count(MALWARE),
```

```
    alo_mw_predictions.count(GOODWARE),
    alo_gw_predictions.count(GOODWARE)
]

# MV Confusion Matrix
ttl_mw_count = len(mv_mw_predictions)
ttl_gw_count = len(mv_gw_predictions)

max_accuracy = 0
bias = 0
threshold = 0.0
mv_confusion_matrix = []

# find optimal threshold
for t in arange(0.0, 1.05, 0.05):
    tn = count_nonzero(mv_gw_predictions < t)
    tp = count_nonzero(mv_mw_predictions >= t)
    _accuracy = (tn+tp) / (ttl_gw_count + ttl_mw_count)

    if _accuracy > max_accuracy:
        max_accuracy = _accuracy
        threshold = t

tn = count_nonzero(mv_gw_predictions < threshold)
tp = count_nonzero(mv_mw_predictions >= threshold)
mv_confusion_matrix = [ tp, ttl_gw_count - tn, ttl_mw_count - tp, tn ]

# RESULTS
print("Majority Vote Accuracy: " + str(accuracy(mv_confusion_matrix)))
print("At Least One Accuracy: " + str(accuracy(alo_confusion_matrix)))
print("Section Accuracy (No Evaluation): " + str(accuracy(
    section_confusion_matrix)))

alo, mv = predict(mwX_test, args.advars, model)
mv_fn = count_nonzero(mv < threshold)
alo_fn = alo.count(GOODWARE)
print("Majority Vote False Negatives: " + str(float(mv_confusion_matrix[2])/
    float(len(mwX_test))))
print("Majority Vote Adversarial False Negatives: " + str(float(mv_fn)/float(len
    (mwX_test))))
print("At Least One False Negatives: " + str(float(alo_confusion_matrix[2])/
    float(len(mwX_test))))
```

```
print("At Least One Adversarial False Negatives: " + str(float(alo_fn)/float(len
    (mwX_test))))

#WRITE TO FILE
if args.results:
    results = [
        args.epochs,
        threshold,
        accuracy(section_confusion_matrix),
        section_confusion_matrix[0],
        section_confusion_matrix[1],
        section_confusion_matrix[2],
        section_confusion_matrix[3],
        accuracy(mv_confusion_matrix),
        mv_confusion_matrix[0],
        mv_confusion_matrix[1],
        mv_confusion_matrix[2],
        mv_confusion_matrix[3],
        accuracy(alo_confusion_matrix),
        alo_confusion_matrix[0],
        alo_confusion_matrix[1],
        alo_confusion_matrix[2],
        alo_confusion_matrix[3],
        mv_fn,
        alo_fn,
        len(mwX_test)
    ]
    f = open(args.results, "a")
    f.write("\n")
    f.write(', '.join([ str(res) for res in results ]))
    f.close()
```

B.3 Build Scripts

B.3.1 createfiledata

```
#!/bin/bash
# note that this file will make changes to the file system.
# also if using a large dataset this could take a long time to run
# it is highly advised to do this manually as it will be
# easier to see what each file is doing
```

```
6
7  # set datadir to the directory that will hold the data, e.g. /home/user/dir/
8  # projectpath should be the path to the project root
9  # infection file name is the file used to infect the malware for adversarial
       variants
10 # this file must exist in the goodware dataset
11 datadir="/home/user/example/"
12 projectpath="/home/user/malwaredetection/"
13 infectionFilename="example.exe"
14
15 # Before running the script, the following commands must be done manually
16 # create three directories as followed
17 # mkdir ${DATAPATH}/goodwareFiles
18 # mkdir ${DATAPATH}/malwareFiles
19 # now copy the goodware and malware files for your own dataset into these
       directories
20 # copy the malware directory into a new directory
21 # cp -R ${DATAPATH}/malwareFiles ${DATAPATH}/adversarialFiles
22
23 echo "[*] Creating Adversarial Variants by injecting ${infectionFilename}"
24 python ${projectpath}src/adversarial-variants/fileInjection.py ${datadir}
       goodwareFiles/${infectionFilename} ${datadir}adversarialFiles/
25
26 echo "[*] Creating image directories"
27 mkdir ${datadir}goodwareFileImages
28 mkdir ${datadir}malwareFileImages
29 mkdir ${datadir}adversarialFileImages
30
31 echo "[*] Converting goodware files to images"
32 python3 ${projectpath}src/data-preparation/fileImageWriter.py -f ${datadir}
       goodwareFiles/ -g ${datadir}goodwareFileImages/ -r 32
33
34 echo "[*] Converting malware files to images"
35 python3 ${projectpath}src/data-preparation/fileImageWriter.py -f ${datadir}
       malwareFiles/ -g ${datadir}malwareFileImages/ -r 32
36
37 echo "[*] Converting adversarial variants to images"
38 python3 ${projectpath}src/data-preparation/fileImageWriter.py -f ${datadir}
       adversarialFiles/ -g ${datadir}adversarialFileImages/ -r 32
39
40 echo "[*] Finished"
```

B.3.2 createsectiondata

```
#!/bin/bash
# note that this file will make changes to the file system.
# also if using a large dataset this could take a long time to run
# it is highly advised to do this manually as it will be
# easier to see what each file is doing

# set datadir to the directory that will hold the data, e.g. /home/user/dir/
# projectpath should be the path to the project root
# infection file name is the file used to infect the malware for adversarial
    variants
# this file must exist in the goodware dataset
datadir="/home/user/example/"
projectpath="/home/user/malwaredetection/"
infectionFilename="user.exe"

# Before running the script, the following commands must be done manually
# create three directories as followed
# mkdir ${datadir}/goodwareFiles
# mkdir ${datadir}/malwareFiles
# now copy the goodware and malware files for your own dataset into these
    directories
# copy the malware directory into a new directory
# cp -R ${datadir}/malwareFiles ${datadir}/adversarialFiles

echo "[*] Creating Adversarial Variants by injecting ${infectionFilename}"
python ${projectpath}src/adversarial-variants/sectionInjection.py ${datadir}
    goodwareFiles/${infectionFilename} ${datadir}adversarialFiles/

echo "[*] Creating section directories"
mkdir ${datadir}goodwareSectionFiles
mkdir ${datadir}malwareSectionFiles
mkdir ${datadir}adversarialSectionFiles

echo "[*] Splitting goodware into section files"
python3 ${projectpath}src/data-preparation/fileSplitter.py -s ${datadir}
    goodwareFiles/ -d ${datadir}goodwareSectionFiles/
echo "[*] Splitting malware into section files"
python3 ${projectpath}src/data-preparation/fileSplitter.py -s ${datadir}
    malwareFiles/ -d ${datadir}malwareSectionFiles/
echo "[*] Splitting adversarial files into section files"
```

```
36 python3 ${projectpath}src/data-preparation/fileSplitter.py -s ${datadir}
       adversarialFiles/ -d ${datadir}adversarialSectionFiles/
37
38 echo "[*] Creating image directories"
39 mkdir ${datadir}goodwareSectionImages
40 mkdir ${datadir}malwareSectionImages
41 mkdir ${datadir}adversarialSectionImages
42
43 echo "[*] Converting goodware files to images"
44 python3 ${projectpath}src/data-preparation/sectionImageWriter.py -f ${datadir}
       goodwareSectionFiles/ -g ${datadir}goodwareSectionImages/ -r 32 2>/dev/null
45
46 echo "[*] Converting malware files to images"
47 python3 ${projectpath}src/data-preparation/sectionImageWriter.py -f ${datadir}
       malwareSectionFiles/ -g ${datadir}malwareSectionImages/ -r 32 2>/dev/null
48
49 echo "[*] Converting adversarial variants to images"
50 python3 ${projectpath}src/data-preparation/sectionImageWriter.py -f ${datadir}
       adversarialSectionFiles/ -g ${datadir}adversarialSectionImages/ -r 32 2>/dev
       /null
51
52 echo "[*] Finished"
```

Appendix C

Results

Contents

The following is a listing of the results obtained through experimentation in this report. As the Adversarial Variant results were gathered at the same time, the False Negatives of the Adversarial Variants are given at the same time.

C.1 File Classifier

File Classification Results					
Experiment	TP	FP	FN	TN	Adversarial FN
1	148	18	7	137	73
2	137	13	18	142	126
3	142	12	13	143	77
4	137	14	18	141	115
5	147	20	8	135	66
6	144	20	11	135	139
7	141	12	14	143	140
8	141	18	14	137	138
9	141	13	14	142	136

File Classification Results					
Experiment	TP	FP	FN	TN	Adversarial FN
10	143	18	12	137	140
11	138	15	17	140	134
12	146	19	9	136	126
13	140	8	15	147	142
14	140	9	15	146	149
15	143	19	12	136	93
16	138	14	17	141	141
17	145	20	10	135	122
18	133	7	22	148	147
19	137	17	18	138	106
20	150	17	5	138	133
21	142	20	13	135	133
22	144	22	11	133	92
23	149	14	6	141	138
24	139	12	16	143	138
25	144	13	11	142	137
26	145	10	10	145	108
27	140	14	15	141	109
28	143	22	12	133	130
29	137	12	18	143	127
30	144	21	11	134	34
31	138	13	17	142	139
32	137	5	18	150	136
33	134	13	21	142	132
34	146	14	9	141	116
35	133	11	22	144	130
36	147	19	8	136	117
37	147	21	8	134	130
38	141	7	14	148	101
39	138	10	17	145	109

File Classification Results					
Experiment	TP	FP	FN	TN	Adversarial FN
40	130	10	25	145	137
41	142	13	13	142	106
42	151	15	4	140	139
43	145	13	10	142	91
44	146	12	9	143	81
45	150	14	5	141	108
46	144	21	11	134	116
47	139	13	16	142	143
48	131	8	24	147	109
49	141	16	14	139	116
50	140	13	15	142	137
51	139	5	16	150	137
52	147	20	8	135	104
53	144	11	11	144	130
54	146	19	9	136	133
55	140	10	15	145	114
56	141	19	14	136	112
57	141	20	14	135	145
58	136	14	19	141	106
59	132	11	23	144	134
60	150	17	5	138	72
61	148	19	7	136	115
62	140	12	15	143	133
63	135	13	20	142	133
64	140	11	15	144	136
65	138	16	17	139	139
66	148	22	7	133	88
67	143	10	12	145	136
68	148	20	7	135	127
69	150	13	5	142	143

File Classification Results					
Experiment	TP	FP	FN	TN	Adversarial FN
70	144	10	11	145	114
71	131	11	24	144	84
72	144	13	11	142	100
73	143	10	12	145	137
74	145	8	10	147	146
75	141	11	14	144	123
76	134	11	21	144	130
77	151	18	4	137	83
78	139	10	16	145	122
79	144	14	11	141	109
80	149	18	6	137	115
81	143	12	12	143	115
82	143	11	12	144	112
83	145	14	10	141	118
84	142	7	13	148	112
85	134	4	21	151	133
86	136	11	19	144	108
87	143	16	12	139	143
88	144	12	11	143	116
89	147	16	8	139	127
90	141	11	14	144	112
91	142	13	13	142	84
92	141	10	14	145	143
93	147	19	8	136	127
94	146	16	9	139	141
95	138	11	17	144	80
96	141	12	14	143	133
97	149	18	6	137	92
98	141	13	14	142	81
99	143	11	12	144	126

File Classification Results					
Experiment	TP	FP	FN	TN	Adversarial FN
100	137	7	18	148	138

C.2 Section Classifier - ALO

Section Classification (ALO) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
1	155	122	0	33	0
2	154	115	1	40	1
3	155	115	0	40	0
4	155	115	0	40	0
5	155	109	0	46	0
6	155	127	0	28	0
7	154	119	1	36	1
8	155	104	0	51	0
9	155	117	0	38	0
10	155	88	0	67	0
11	155	132	0	23	0
12	154	78	1	77	1
13	155	118	0	37	0
14	155	110	0	45	0
15	155	122	0	33	0
16	154	102	1	53	0
17	153	125	2	30	0
18	153	105	2	50	2
19	154	112	1	43	0
20	155	78	0	77	0
21	155	53	0	102	0
22	155	74	0	81	0
23	155	116	0	39	0
24	154	112	1	43	1

Section Classification (ALO) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
25	149	105	6	50	6
26	150	100	5	55	5
27	155	118	0	37	0
28	154	105	1	50	0
29	155	107	0	48	0
30	154	107	1	48	1
31	155	109	0	46	0
32	152	107	3	48	0
33	155	140	0	15	0
34	154	95	1	60	0
35	155	115	0	40	0
36	155	85	0	70	0
37	155	138	0	17	0
38	155	118	0	37	0
39	154	113	1	42	1
40	154	107	1	48	1
41	155	107	0	48	0
42	154	122	1	33	1
43	154	87	1	68	1
44	155	105	0	50	0
45	154	69	1	86	1
46	153	116	2	39	0
47	153	69	2	86	2
48	154	105	1	50	0
49	155	114	0	41	0
50	153	104	2	51	2
51	153	122	2	33	2
52	155	142	0	13	0
53	155	81	0	74	0
54	155	99	0	56	0

Section Classification (ALO) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
55	155	117	0	38	0
56	154	130	1	25	1
57	155	115	0	40	0
58	154	136	1	19	1
59	155	111	0	44	0
60	155	112	0	43	0
61	154	110	1	45	1
62	154	109	1	46	0
63	155	104	0	51	0
64	154	96	1	59	1
65	154	107	1	48	0
66	155	101	0	54	0
67	155	108	0	47	0
68	155	135	0	20	0
69	155	113	0	42	0
70	155	113	0	42	0
71	155	121	0	34	0
72	155	123	0	32	0
73	154	107	1	48	1
74	154	112	1	43	0
75	154	117	1	38	0
76	154	72	1	83	0
77	155	124	0	31	0
78	155	65	0	90	0
79	153	126	2	29	2
80	154	106	1	49	0
81	153	98	2	57	2
82	154	114	1	41	1
83	154	111	1	44	0
84	154	136	1	19	1

Section Classification (ALO) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
85	155	90	0	65	0
86	155	81	0	74	0
87	155	140	0	15	0
88	155	110	0	45	0
89	155	80	0	75	0
90	155	93	0	62	0
91	154	120	1	35	1
92	155	134	0	21	0
93	155	126	0	29	0
94	154	102	1	53	1
95	155	105	0	50	0
96	154	101	1	54	0
97	155	120	0	35	0
98	152	122	3	33	0
99	155	87	0	68	0
100	155	116	0	39	0

C.3 Section Classifier - MV

Section Classification (MV) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
1	151	7	4	148	155
2	143	6	12	149	155
3	144	2	11	153	0
4	136	2	19	153	0
5	149	14	6	141	0
6	149	4	6	151	155
7	144	4	11	151	155
8	145	11	10	144	155
9	147	0	8	155	155

Section Classification (MV) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
10	152	3	3	152	150
11	151	3	4	152	0
12	154	21	1	134	155
13	145	2	10	153	0
14	149	6	6	149	155
15	151	2	4	153	0
16	152	14	3	141	0
17	146	11	9	144	0
18	148	9	7	146	152
19	139	2	16	153	0
20	153	7	2	148	149
21	142	6	13	149	152
22	151	13	4	142	149
23	138	5	17	150	155
24	146	0	9	155	155
25	140	5	15	150	155
26	147	13	8	142	155
27	142	21	13	134	0
28	143	1	12	154	0
29	148	2	7	153	0
30	148	8	7	147	155
31	147	12	8	143	155
32	147	4	8	151	0
33	148	16	7	139	0
34	151	10	4	145	0
35	150	1	5	154	155
36	155	18	0	137	152
37	154	9	1	146	0
38	150	9	5	146	0
39	148	9	7	146	155

Section Classification (MV) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
40	141	2	14	153	155
41	146	4	9	151	155
42	146	3	9	152	155
43	148	10	7	145	154
44	153	20	2	135	0
45	143	14	12	141	154
46	147	1	8	154	0
47	151	11	4	144	154
48	149	7	6	148	0
49	148	8	7	147	0
50	148	11	7	144	155
51	152	10	3	145	153
52	153	5	2	150	155
53	151	9	4	146	155
54	145	10	10	145	155
55	143	3	12	152	155
56	150	3	5	152	155
57	149	10	6	145	155
58	150	18	5	137	155
59	149	3	6	152	155
60	151	15	4	140	155
61	151	7	4	148	155
62	142	1	13	154	0
63	145	2	10	153	155
64	153	9	2	146	153
65	150	16	5	139	0
66	148	6	7	149	0
67	152	10	3	145	155
68	136	6	19	149	155
69	140	2	15	153	155

Section Classification (MV) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
70	143	1	12	154	155
71	140	6	15	149	155
72	149	5	6	150	0
73	153	11	2	144	155
74	143	7	12	148	0
75	148	16	7	139	0
76	153	8	2	147	0
77	146	5	9	150	155
78	152	7	3	148	154
79	149	6	6	149	155
80	146	12	9	143	0
81	151	12	4	143	154
82	149	8	6	147	155
83	152	4	3	151	0
84	149	8	6	147	155
85	149	8	6	147	155
86	151	17	4	138	152
87	139	7	16	148	0
88	153	12	2	143	152
89	144	11	11	144	0
90	152	11	3	144	155
91	142	5	13	150	155
92	153	15	2	140	155
93	150	9	5	146	0
94	152	13	3	142	154
95	155	14	0	141	152
96	152	13	3	142	0
97	147	8	8	147	0
98	150	11	5	144	0
99	145	8	10	147	155

Section Classification (MV) Results					
Experiment	TP	FP	FN	TN	Adversarial FN
100	149	8	6	147	0