Malware Analysis with Machine Learning Methods

4th Year Honours Project

Author: Cedric Ecran (H00259115)
Supervisor: Dr. Hani Ragab
B.Sc. (Hons) Computer Systems
School of Mathematical and Computer Science
Heriot Watt University
April 2020

Declaration

I, Cedric Ecran, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of references employed is included.

Signed: Cedric Ecran

Date: 23/04/2020

Abstract

With the growing amount of data in cyberspace, so too does the number of malicious files, seeking to harm unsuspecting users. Malware can be very tricky to avoid as the expanding collection and complexity are overwhelming the current signature-based solutions. Machine Learning could be the future with promises of efficiency and a higher detection rate of zero-day malware then-current anti-viruses. Several existing research proposals explore the challenge of malware detection using machine learning. They, however, utilized a variety of datasets to both train and assess their models. This results in the infeasibility of comparing between those proposals.

In this project, we seek to compare the various proposals of malware detection using machine learning. We built a reference dataset of 22,300 benign and 22,300 malicious files. We also implement a universal feature extractor, that can extract all header features used by the investigated papers, for all windows executable formats. We proceeded to train and test all investigated papers using the features extracted from the reference dataset. For each paper, we follow, as closely as possible, the steps and settings used by its authors. Lastly, we compare and critically review the investigated papers and attempt recombination of features based on the acquired results.

In addition to our originally planned objectives and requirements, we proposed a new machine learning approach for malware detection that managed to get a higher accuracy than the four papers we implemented.

Table of Contents

Chapter 1

Introduction

The rate of malware is ever-expanding at an alarming rate, Kaspersky reported that in the first quarter of 2019 alone there were more than 150 million malware attacks in just the META region, averaging out to approximately 1.6 million per day [1]. These figures indicate a 108% increase over the number of malware attacks in the first quarter of 2018 [1]. The unprecedented scale of malware attacks highlights the need for machine learning-based detection mechanisms that operate effectively and efficiently to combat the ever-growing influx.

1. 1. Aim

The dissertation aims to **critically review and implement a selection of machine learning-based malware detection research proposals to evaluate the results achieved**. We will focus on portable executable static feature extracted research proposals, with the primary factors being the achieved accuracy of each reviewed proposal.

1. 2. Objectives

The objectives of the dissertation are the following:

- Construct an extensive dataset of both malicious and benign files, with all files passing validation tests from current anti-virus solutions
- Perform feature extraction and reduction on the dataset
- Apply various machine learning techniques, from a selection of research proposals, on the extracted features from the dataset
- Evaluate the achieved results by the different machine learning techniques

1.3. Extra Achievements

In addition to the above objectives, we proposed a new machine learning approach for malware detection that managed to get a higher accuracy than the four papers we implemented.

1.4. Report Structure

The report structure is as follows; the first section introduces the concept of malware and machine learning. These topics include a variety of reference sources based on their respective topic. The next section showcases the related works that are relevant to the topic of this project. The following section introduces the design and construction of the dataset that is utilized for all testing. The proceeding section coves the design methodology and implementation of the critical areas of the project. The section after showcases the achieved results and the analysis of those results; an evaluation is also present at the end of the section. Lastly, a conclusion is present, which includes future proposed work on the project.

1. 5. Introduction to Malware and Machine Learning

1. 5. 1. Malware

The term malware is a combination of the words malicious and software hinting heavily to its definition. The definition of malware or malicious software is any software that brings harm to either the user, computer, or network that it is executed on [2]. Malware can adopt a wide variety of forms, including viruses, worms, rootkits, and trojans, although the basic principle stands for all variants. In this dissertation project, all known types of malware, in the Windows executable format, will be utilized for training and testing purposes.

1. 5. 2. Machine Learning

Machine learning (ML) grew out of the broader field of Artificial Intelligence, whereby machines attempt to mimic intelligent human abilities [3]. In Machine Learning algorithms, the "learning" element involves training from examples to perform their task rather than traditional ways of having pre-defined rule sets [4]. Machine learning aims to predict, and correctly classify/label sensed data based on experience/training [3]. There are two types of learning unsupervised and supervised, unsupervised revolves around detecting either anomalies within the input data or hidden regularities [3]. In supervised learning, the algorithm attempt to assign a label to each example to classify based on the previously seen examples [4]. In unsupervised, the data has no labels or meanings to the algorithm, whereas supervised does, this labeling allows for the useful classification of malware by supervised learning machine learning algorithms. We will focus on supervised learning techniques for training and testing for detecting malicious files.

Chapter 2

Literature Review

2. 1. Background Theoretical Research

2. 1. 1. Malware Analysis

The goal of malware analysis is to extract information from a given malware sample with the intent to identify and possibly classify. The reason for doing so usually involves accessing the damage dealt by it, examining the security breach on the target system, and identifying defense vulnerabilities that were exploited [5]. The process involves two main techniques: Static analysis where malware is examined in a "dead" state and Dynamic analysis, where the malware is executed to be observed [5].

Static analysis usually is the faster and more straightforward approach to malware analysis due to none of the overhead that comes with executing malware. Static analysis involves two main techniques the first, and simplest, involves examining the Portable Executable (PE) a file format containing information used by the OS loader to execute the code that it is wrapped around [6]. The most useful information, from a malware analysis perspective, located in the PE are the import functions that the executable calls. These imports allow a program to call functions stored in a different program. This becomes particularly important considering code libraries are linked in this fashion, thereby allowing the malware creator to call functions without the code to perform that function being present within the malware [6]. The import functions can be implemented in several ways, one of which is statically whereby the linked library is loaded into the executable before any code is run. The more commonly utilized way is by dynamic linking whereby the libraries are not loaded into the executable; instead, the functions are run within the libraries and are only run once called upon [6]. This is useful in two ways for the malware; it keeps the malware's size limited as it does not load the libraries, thus keeping

efficiency high. Secondly, by not loading in any libraries, the malware does not change, thereby possibly helping to avoid detection from file fingerprinting. File fingerprinting is the process of hashing a file, performing some operation on/with it, and then once complete, rehashing it and comparing it to the original hash to ascertain if changes have occurred to the file [5]. The use of dynamic link libraries (DLLs) in malware design is widespread not only because of the efficiency and possible obfuscation of the true nature of the malware but additionally due to DLL's being ubiquitous and universal. Within the Windows OS, it allows malware engineers to reuse them, thereby allowing for code reusability and efficiency when creating additional malware [6]. The other, more advanced, static analysis technique requires disassembling the given malware to reveal the operations it will execute. Once disassembled, the operation codes (opcodes), instructions to the CPU detailing operations that need to be performed, are commonly examined as they are the lowest level building blocks of any executable, being the literal commands passed to the CPU [6].

Dynamic analysis allows the malware to execute to observe and possibly alter the malware as it executes [5]. Dynamic analysis has a distinct advantage over static analysis due to being able to observe certain runtime features and operations that may not have been detected with static analysis. However, dynamic analysis requires a more significant amount of setup and caution due to the presence of active running malware. This will usually involve a safe environment (virtual machine), a process monitor [5], and a debugger to view the runtime state of the malware [6]. Although extremely exhaustive static analysis could, in theory, reveal all possible functionalities of a given malware, this would be extremely difficult and time-consuming [5]. In modern malware analysis, elements from both static and dynamic analysis are used to gain a complete view of the analyzed malware [6].

2. 1. 2. Machine Learning

Within the field of machine learning, problems can be labeled as one of two types; classification or regression [3]. In a classification problem, the labeling of the done by the algorithm of the input data is discrete, an example being black or white, whereas, with a regression problem, the labels are more complex or real-valued [3]. The scope of use for this dissertation revolves around the classification problem of files, with their label being either malicious or benign. There are a wide variety of classification algorithms within the machine learning field; however, the focus for this dissertation will be on the most commonly used, based on researched papers on malware analysis using machine learning.

The k-Nearest Neighbor classification involves the training data being stored within a dimensional space in the form of points/positions, one for each piece of data. When the testing data is classified as its data, points are also classified into this space. Based on the distance to k amount of training data points, the test data [3]. This method is likely the simplest classification method to understand and has, on average, a shallow error rate, but the memory storage requirements and high computational workload make it not optimal for large datasets [3].

Another classification method is decision trees whereby, from training data, a tree is constructed based on partitioning the input space [3]. The tree is constructed from the training data by continually selecting the best/pure discerning features for each of the nodes of the tree [3]. The tree keeps expanding if input from the training data cannot be successfully classified by traversing down the given tree. Classification of the test data is also done by traversing down the tree, with the leaf nodes containing the classification labels [3]. Decision trees work best with smaller datasets with low dimensionality, as they do not scale well when increasing these factors and being to require a large amount of memory storage and increase the risk of overfitting [3] [7]. To reduce these effects, we can either limit the size that the tree can grow to, or perform pruning on the completed tree, whereby nodes are replaced with either sub nodes or leaves based on a bound or estimate [7]. Another implementation

of decision trees by J. Quinlan is named C4.5 and is commonly used within malware analysis using machine learning research [3].

Further development of decision trees is the random forests whereby a large amount, unusually in the hundredths of decision trees, are created using some part of the training data [7]. For each tree, only a random subset of the feature list is considered from the training data [4]. Once unseen data, test data, is passed to the random forest, each iteration of the decision tree will classify the data [4]. Based on the results of classification, each tree votes for classification of the input data with the majority classification being adopted [4].

The above methods within this section ensured that classification was clear and definitive for its constructed model, but if new unseen data, which is somewhat different from the training data used, are used, they may either fail or become computationally infeasible [7]. The possible solution is to use algorithms that allow for generalization, the abstractification of data, one of which is support vector machines (SVM) [3]. In SVM, the training data is mapped into a feature space where the data points are separated by a large margin hyperplane [3]. This hyperplane needs to have a large margin, the smallest distance between the hyperplane and a data point, to classify new data [7]. New data will be mapped into the feature space. Due to the extensive margin, the new data will either be on one of two sides of the hyperplane, thereby classifying it [3]. The exact distance from the hyperplane is not considered as it allows for generalization of the data [7].

Another algorithm that allows for generalization is boosting whereby linear predictors, the output from other classification algorithms, are weighed and then combined to create a final prediction [3] [7]. This generally results in significant performance improvements over the standalone algorithms [3]. The first boosting algorithm that will be focused on is Adaptive Boosting (AdaBoost), the very first practical implementation of boosting [7].

In machine learning, when using a single dataset, before running a classification or regression algorithm, typically, one must decide how to split that dataset. This is required as the algorithm must have samples to train on and separate samples to test the constructed model [7]. This can be done using the train test split approach whereby a percentage of the total dataset is reserved for testing purposes only; this is usually 30%. This approach is not always suitable if working with a smaller dataset, where withholding 30% of the data will drastically change the model's performance [7]. In this situation, the k-cross-fold validation can be applied, whereby the original dataset is split into k – subsections. The algorithm is trained on the union of the other subsects with testing occurring with the chosen subsect [7].

The primary output to measure algorithm success will be the achieved classification accuracy. There are other measures of success that will also be analyzed. The False Positive count is the number of benign files that were identified as being malware, the rate is calculated by dividing the total False Positives by the sum of False Positives and True Negatives [7]. This is an essential measure due to them being equivalent to real-world false alarms. The other measurement will be the False Negative count which is the total amount of malware that is identified as benign, the rate is calculated by dividing the total False Negatives by the sum of False Negatives and True Positives [7]. This measurement is crucial as it is equivalent to an undiscovered breach. These are the measurements that will be focused on in this project. Additional measurements will be reported both from our testing and the research papers that have been researched. Figure 1 showcases the rest of the measurements below.

Metric	Formulae	Description
True Positive (TP)	count()	Total malware which predicted as malware
False Positive (FP)	count()	Total benign which predicted as malware
True Negative (TN)	count()	Total benign which predicted as benign
False Negative (FN)	count()	Total malware which predicted as benign
Accuracy	$\frac{TP+TN}{TP+FP+TN+FN}$	Rate of correct predictions
Precision	$\frac{FP}{TN+FP}$	Proportion of predicted malware which are actual malware
Recall/TPR	$\frac{TP}{TP+FP}$	Proportion of actual malware which are predicted malware

Figure 1 [8] Performance Measure metrics

2. 2. Related Works

Liu Liu et al. [2] proposed using static analysis with the use of multiple feature extraction process, greyscale imaging, operation code (Opcode) features, and dynamic link library calls to more accurately locate the frequently used functions within the malware. Then extract the Opcode features with an n-gram model, performing text feature extraction, combined with a control flow graph (CFG). The CFG allows for relationships between the functions to be considered as it shows a high-level overview of the malware, whereas the n-gram model searches localized.

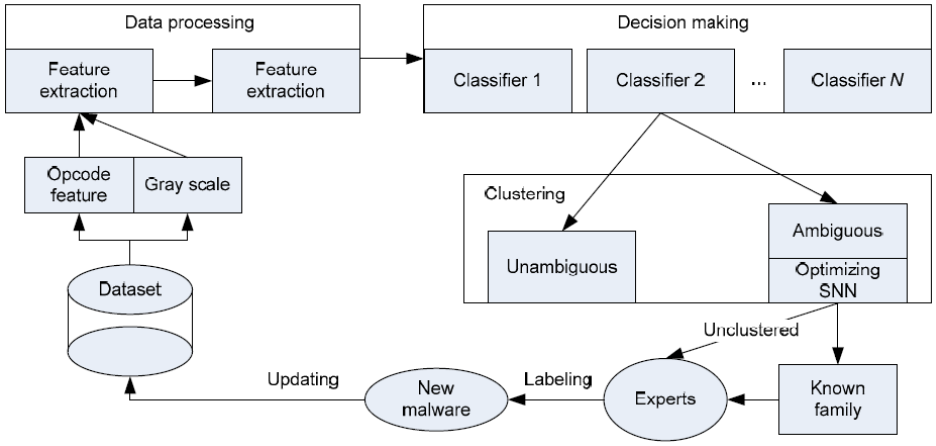

Figure 2 showcases a high-level overview of source [2] implementation

The malware collection used totaled 21,740 instances consisting of nine different malware types. After feature extraction, a variety of machine learning classifiers consisting of random forests (RF), K-nearest neighbor (K-NN), Gradient-boosting (GB), Naïve Bayes (NB), logistic regression (LR), support vector machine-poly (SP) and decision trees (DT) were applied on known malware but only with varying levels of n-gram feature extraction. The results indicated that 2- and 3-gram feature extraction were most accurate with the 3-gram RF approach, achieving a 94.7%. The next stage was to combine the greyscale imaging and the 3-gram feature extractions, which, when tested overall classifiers, averaged a 95.7% accuracy compared to the 3-gram average of 84.3% and the greyscale imaging average of 90.7%. The highest combined feature extraction accuracy was the RF classifier, which managed an accuracy of 98.9%.

After classification of the known malware, clustering was performed on the data, and an estimated eight clusters were formed. Then another testing sample was tested, but this time the sample consisted of 900 samples, of which 90 were new unknown samples but of the same family. Each classifier would vote on the likelihood of the file being suspicious, therefore malware. The experiment showed that the new malware was classified accurately 86.7% of the time with a new cluster being formed.

Critical Analysis

The malware dataset could be more extensive, and we are also not told the size of the benign dataset, as this could inflate the accuracy if the benign dataset is minimal. There is no mention of the origin of the dataset, nor if any validation with existing anti-virus tools was performed. The missed opportunity of including windows API calls into the feature extraction as this may have resulted in higher accuracy. They could also have included dynamic analysis to possibly integrate the results like [9] to possibly improve accuracy but to combat anti-analysis techniques. When performing their classification, they should have performed the k-fold cross-validation technique to increase the validity of their results. The lack of false negative and false positive data is unfortunate as it would give greater insight into the performance of the implemented machine learning method. The exciting element of this paper is the use of greyscale imaging combined with the more traditional static analysis approaches of Opcode and DLL calls. The use of greyscale could be better served by comparing before and after the execution of the program. This would allow us to see how much has changed and if that has a strong correlation to the program being malicious.

Jingling Zhao et al. [10] combined both static and dynamic information extraction before applying feature extraction on the combined information. In static extraction, they used Interactive Disassembler Professional (IDA) and IDA Python to disassemble their malware samples. The focus on static features was on the types of string's present and the DLL import functions with their frequencies being noted. The dynamic information extraction was done in the Windows operating system, with the malware being monitored with Intel PIN. Intel PIN provides varying levels of program execution

information, such as a function level and instruction level. The monitoring of the malware samples was focused on the assembly instructions the malware used and the API calls initiated during execution.

The feature selection consisted of two algorithms Information Gain and N-gram algorithm. Information gain looks at the amount of information that a particular feature adds to the classification system. The N-gram algorithm was used on the malware Opcodes performing a sliding window with a size of N. Two ML classification algorithms were tested with the now extracted feature set, one being Naïve Bays and the other Support Vector Machines (SVM).

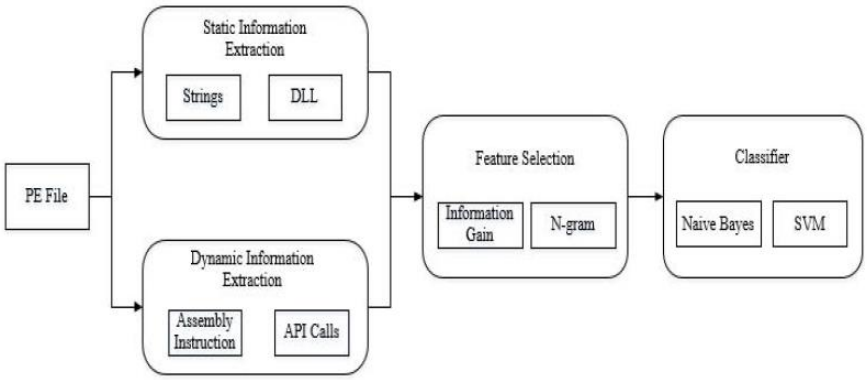

Figure 3 is the flowchart of the source [10] malware detection system

The dataset used consisted of 3,548 malware and 1,628 benign programs; the malware programs were collected from a website named Virrussign. The benign files were interestingly not a Windows OS executable collection, as they believed this to have a too high degree of similarity between the files. Instead, a collection was acquired from malwr.com. The results showed that the feature extraction/reduction obtained the top 200 API functions and strings, along with the top 100 DLL imports. The achieved accuracy using the Naïve Bays classification algorithm was 97%, and the SVM achieved 98%.

Critical Analysis

The dataset is small, but the ratio of benign to malware is good. The use of only two classification algorithms is puzzling, certainly when it is suggested from most research papers on the matter that

random forests usually perform very well with mid-sized datasets like the one used here. When performing their classification, they should have performed the k-fold cross-validation technique to increase the validity of their results. There is no mention of the false-negative/false-positive rates, and these should be included to give a better indication of the performance of implemented machine learning methods. The integration of static and dynamic features is similar to [9] implementation, but they each have separate reasons. An interesting note from this paper is the use of non-Windows OS executable collections as the OS due to similarity concerns. This could be a valid argument as the files usually do exhibit similar features and therefore be boosting the accuracy of detecting them.

Muhammad Ijaz et al. [11] compared static and dynamic malware analysis, and combinations of multiple dynamic features, to determine which achieves the highest attainable malware detection accuracy. The extraction of static features was done using a python library called Portable Executable PEFILE. PEFILE provided PE file headers from which many features were recorded, such as section size, size of data, and time and date stamps. Dynamic feature extraction was done using Cuckoo sandbox whereby registry keys, file state, API calls, and IPS and DNS queries were monitored and recorded. They then decided on combining multiple variants of the dynamic features, all of which can be seen in figure 3 below.

S. No	***Combinations***
1	*APIs+DLLs*
2	*APIs+Summary information*
3	*DLLs+Registry*
4	*DLLs+Summary information*
5	*Registry*
6	*DLLs*
7	*Registry + summary information +DLLs+APIs*
8	*Registry+summary information*
9	*APIs Calls*

Figure 4 shows the dynamic feature combinations tested.

The ML classification algorithms used were Logistic Regression, Decision Tree, Random Forest, Bagging Classifier, AdaBoost Classifier, Tree Classifier, and Gradient Classifier. The non-combined dynamic results showed that Gradient Classifier achieved the highest accuracy of 94.64%

with a false positive of 5.85% and a false negative of 13.96%. This proved to be the lowest false positive percentage with the AdaBoost Classifier achieving the lowest false-negative result of 10.91%. The next set of results are for the combined dynamic features using the same ML classification algorithms. The results showcase only the highest achieved accuracy with a given ML classification algorithm and with no false positives or negatives. The top accuracy was achieved by the combination of API's and summary information (S. No 2 in figure 3) with an accuracy of 95.86%. The final set of results is the static analysis features, once again utilizing the same ML classification algorithms. The highest achieved accuracy with static features was 99.36% using the Gradient Classifier. This resulted in a false positive percentage of 3.25% and a false negative of 2.73%. Random Forest achieved the lowest false-positive percentage, with 3.09% and the lowest false-negative by the Bagging Classifier reaching only 0.60%.

Critical Analysis

Within the paper, they state that the malware dataset was "small," not a very clear answer, and therefore, I can only conclude that it would be a tiny number. This may indicate why they achieved 99.36% using gradient classifiers with static analysis. The k-fold cross-validation technique would undoubtedly have shown a more accurate accuracy as it would make the most out of the "small" dataset. The accuracy improvement between the single dynamic feature and the combinations was rather small of approximately 1.2%. The results they achieved for decision trees using static analysis features have a possible indication of overfitting as the false negative it achieved was a minuscule 0.665%. Their method for extracting the static features, via PEFILE, is a tool we are considering for this dissertation. In the future work section, they theorize that by allowing the program to execute in a dynamic environment, mitigating most of the anti-analysis issues that come with static analysis as described in source [9]. Once the program has executed, perform the static analysis to obtain very accurate and efficient results.

Usukhbayar Baldangombo et al. [12] propose combining data mining methods with machine learning to achieve high classification accuracy. The dataset includes 247,348 files all in the Windows operating system executable format, of which 236,756 are malicious and 10,592 benign. The malicious files were constructed from several public domain sources, one of which is the VX Heavens Virus Collection. The benign section of the dataset was obtained from Download.com and are Windows system files. The authors developed their portable executable parser to perform static analysis named PE Miner, which extracted all the PE header information, API function names, and DLL names located in the PE. Feature extraction revealed the optimal features as the top 88 PE header features, top 130 frequent DLL names, and seven groups of frequently used API's by the malicious files that translates to 2,453 API functions. The classifiers used were support vector machines, decision tree-J48 (java specific version of C4.5), and Naïve-Bays. Cross-validation was performed ten times to assist in validating the robustness of the technique.

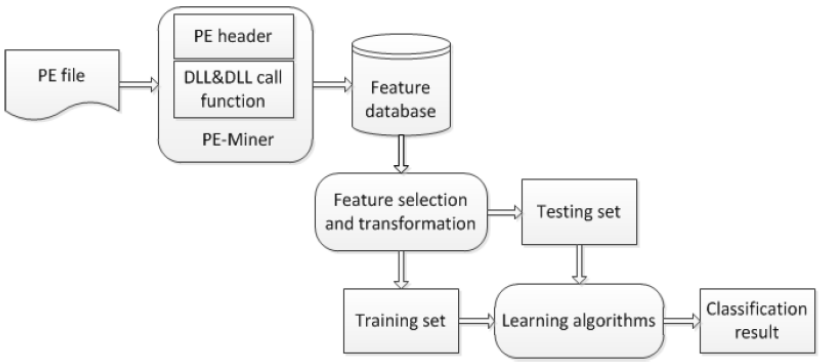

Figure 5 from source [12] showcases the malware detection system architecture

The first set of results was to see what impact that Principal Component Analysis (PCA) had on the direct detection rate of the J48 classifier. PCA is a data mining function, when used on the features, drastically reduced them by as much as 87%. By performing PCA on the selected features, the detection rate of the J48 classifier increased for all three feature categories, with a maximum improvement of 1.3%. The final set of results paired all three classifiers with each of the feature types, and two hybrids the first is PE header and DLLs, the second PE header and API functions. The results showed that the highest direct detection rate came from using the PE header as feature type and the J48 classifier obtaining a 99.5%. This combination also obtained the lowest false positive rate of only 2.7%

and the second-highest overall accuracy at 99%. The highest overall accuracy was achieved by combining API functions as the feature type with the J48 classifier; this achieved a 99.1% accuracy.

Critical Analysis

A very well sized dataset but could use more benign files as there is currently some oversampling going on in favor of malware. In-depth static analysis was performed, but the lack of information on exactly how their feature selection was made is questionable. The addition of the data mining technique, PCA, brought surprising results by increasing the already very accurate J48 classifier with PE headers by another 0.4%. Their results show that the hybrid implementations did not achieve the same high levels of accuracy as the single feature ones. This could be due to the increase in complexity as algorithms with more generalization will perform better with higher complexity. This is also validated when looking at the SVM results as they are the only ones that increase with the hybrid implementation of the feature selection.

Akash Kumar Singh et al. [9] have attempted to design a machine learning method for the detection of malware. These malicious files have been designed with anti-analysis features by integrating both static and dynamic analysis. The dataset is in Windows OS executable format and has various types of malware present. The size of the dataset is 109, including 25 benign and 84 malicious files. The virtual machine used for dynamic analysis is NORIBEN. The static analysis is combined with the output of the various anti-analysis features that the file may include; these are determined during the execution of the program with dynamic analysis. The first module is the anti-VM module whereby we check via the Interrupt Descriptor Table Register (IDTR) whether the file detects that it is running in a virtual machine. The second is the anti-debugging module whereby looking at specific windows API calls, like IsDebuggerPresent, if the malware has detected a debugger. The third is the packer detection module, here we check if the file has been packaged and in what way, this is done by using PEiD. The final module is the URL analysis module, where URLLIB is used to extract the URL connection, strings, and file information.

The data from both the static and dynamic analysis are integrated to ensure that even if the malware avoids static analysis detection, the dynamic analysis results will still distinguish it as a malicious file. A free machine learning tool was used to perform a classification called WEKA. Three classifier methods were implored, Naïve Bays, support vector machines, and random forests. In an attempt to bolster the robustness of the achieved results, 10-fold cross-validation was utilized. The classification was conducted on static analysis features, dynamic analysis features, and the integrated analysis features. The static analysis features results showed that SVM obtained the highest accuracy at 71%. The dynamic analysis features results indicated that random forests have the highest accuracy of 63.3%. The results using integrated analysis features showed random forests to have the highest accuracy of 73.4%.

Critical Analysis

The tiny dataset of only 109 files total makes the results questionable at best. The use of k-fold cross validation does attempt to help, but the dataset is so small there may not be much of an improvement to the result validity. The research paper looks at the anti-analysis design for certain malware types and offers some exciting solutions to detect them. The feature selection and integration of both the static and dynamic analyses attempted to show the improvement over single feature use. This is similar to the implementation in source [10], but the idea behind it is different. Overall accuracy did improve one the integrated features were used with the classifiers showcasing that their ML method could produce more accurate results, but with such a small dataset, more training and testing needs to be done.

Ajit Kumar et al. [8] propose a machine learning model for malware detection of portable executables. The feature selection proposed consists of 24 raw components from the header spread across the DOS, File, and Optional sections. The dataset utilized for the training and testing of the machine learning algorithms had 2,499 benign and 2,722 malicious files. The machine learning algorithms utilized were Logistic Regression, Linear Discriminant Analysis, Random Forests, Decision Trees, Gaussian Naïve Bayes, and k-Nearest Neighbor. The dataset was split in a 70/30 training to testing ratio. Tenfold cross-validation was also utilized to help prevent overfitting. The lowest achieved

accuracy result was utilizing the Gaussian Naïve Bayes algorithm, which only managed an accuracy of 56.04%. The highest achieved accuracy was 97.43% utilizing the Random Forest algorithm.

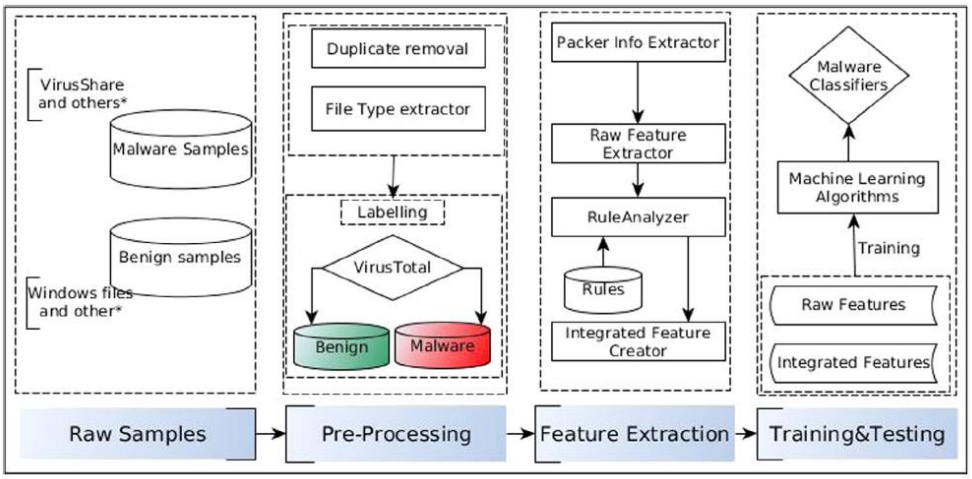

Figure 6 [8] The Process Diagram Utilized

Critical Analysis

As with previously discussed research papers, the size of the dataset is rather small. In this scenario, the problems associated with a biased dataset are not present due to the near equality of both benign and malicious files. The lowest achieve accuracy of 56.04% is in stark contrast to Jingling Zhao et al. [10], who achieved using the same algorithm a 95% accuracy score. This is probably due to Jingling Zhao et al. utilizing multiple data points, of which a fraction only came from the file's header.

Zane Markel and Michael Bilzor [13] performed malware classification utilizing three machine learning algorithms utilizing only features found in the portable executable header of the file. The feature selection included just six raw features from the optional header component. Two additional derived features supplemented these. The first being all the file header flags of a file. The second was a high entropy variable. This variable would only be set to true in the dataset if the file contained a single section that had an entropy of greater than seven. The dataset built for training and testing included 122,799 malicious and 42,003 benign files. The dataset was split 90 training and 10% testing. The machine learning algorithms utilized were Naïve Bayes, Decision Trees in the CART form, and Logistic

Regression. The lowest achieved accuracy was 51.27%, this was using the Naïve Bayes classifier. The highest achieved accuracy was 97.92% using Decision Trees in the CART form.

Critical Analysis

The dataset size is more than adequate; however, the bias towards malicious files could lead to incorrect accuracy scores as the testing set may include a more substantial proportion of malicious files. Typically this bias can be reduced with either having a large test percentile over the training percentile or utilizing cross folding. This is not the case as the testing percentile is very low at 10%, and no cross folding is utilized.

Hrushikesh Shukla et al. [14] investigated the achieved accuracy of five machine learning algorithms utilizing 50 portable executable raw features extracted from the file header. The algorithms used were Decision Trees, Logistic Regression, Random Forests, Artificial Neural Networks, and Naïve Bayes. The utilized dataset consisted of 5000 benign and 5000 malicious files. A split of 67% for training and 33% for testing was used on the dataset for each classification algorithm. The lowest achieved accuracy was 52.3% utilizing Logistic Regression algorithms. The highest achieving accuracy algorithm was Random Forests achieving an accuracy of 96.2%. 5-fold cross-validation was utilized on the dataset to complement the test and training split.

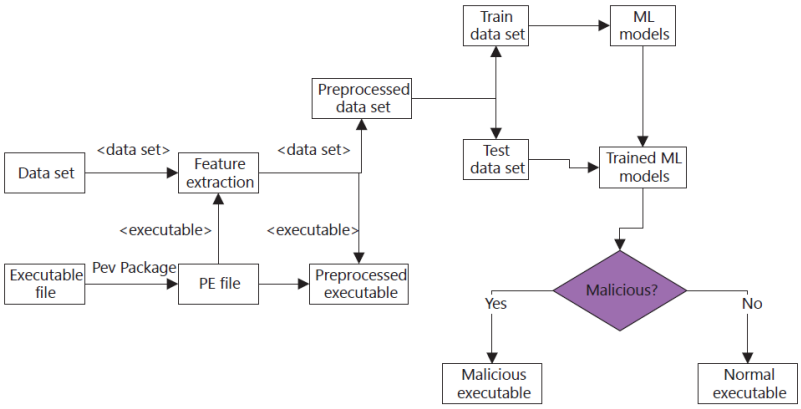

Figure 7 [14] The Utilized System Architecture

Critical Analysis

The conference proceedings do not indicate exactly which header information they utilized. This prevents us from utilizing this paper with our dataset as their feature list is unavailable. The dataset used, although balanced, is too small to accurately portray a wide variety of both malicious and benign files. The main reason this paper is included is that the feature extraction methodology closely portrays our own. They utilize the same python library named pefile for header extraction.

Mohamed Belaoud et al. [15] performed malware analysis utilizing seven machine learning algorithms. The feature selection was conducted on the portable executable header, specifically the file header and the optional header. The dataset utilized during training and testing consisted of 214 benign and 338 malicious files; this was then split by using 6-fold cross-validation. The classification algorithms used were Logistic Regression, Linear Discriminant Analysis, k-Nearest Neighbor, Decision Tree in the CART form, Naïve Bayes, Support Vector Machines, and Voting. The lowest achieved accuracy was 97.3% using Linear Regression, and the highest was 100% using k Nearest Neighbor or CART.

Critical Analysis

The dataset is incredibly small, and therefore results like the 100% achieved accuracies are highly disputable. The primary use of this paper is to showcase that research done in the field of specifically the file header and optional header of the pe file header.

2. 3. Critical Reflection

Source	Static/Dynamic Analysis	Dataset Size	Feature Extraction	Classification Algorithms	Highest Obtained Accuracy	False Negative Rate

Liu Liu et al. [2]	Static	21,740	Greyscale, Opcodes, DLL's	RF,K-NN,GBC,NB,LR,SVM,DT	RF 98.9%	NA
Jingling Zhao et al. [10]	Static & Dynamic	5,176	Strings, DLL's, Assembly Instructions, API Calls	NB, SVM	NB 95%	NA
Muhammad Ijaz et al. [11]	Static & Dynamic	"small"	API Calls, DLL's, Registers, Summary Information	LR, DT, RF, BC, AD, TR, GB	Static GB 99.36%	2.73%
Usukhbayar Baldangombo et al. [12]	Static	247,348	PE Header, DLL's, API Calls	NB, SVM, C4	API Calls C4 99.1%	NA
Akash Kumar Singh et al. [9]	Static & Dynamic	109	(Anti-VM, Debugging, Packer, URL Outputs), Functions, API Calls	NB, SVM, RF	Integrated RF 73.47%	NA
Ajit Kumar et al. [8]	Static	5,221	PE Header (DOS, File, and Optional Sections)	LR, LDA, RF, DT, NB, kNN	RF 97.43%	NA
Zane Markel and Michael Bilzor [13]	Static	164,802	Entire PE Header	NB, DT, LR	DT 97.2%	NA

Hrushikesh Shulka et al. [14]	Static	10,000	50 Features from the PE Header	DT, LR, FT, ANN, NB	RF 96.2	NA
Mohamed Belaoued et al. [15]	Static	552	Entire PE File and Optional Headers	LR, LDA, kNN, DT, NB, SVM, Voting	kNN/DT 100%	NA

Index: Random Forests (RF), K-nearest neighbor (K-NN), Gradient-boosting (GB), Naïve Bayes (NB), Logistic Regression (LR), Linear Discriminant Analysis (LDA), Support Vector Machine (SVM), Decision Trees (DT), Bagging Classifier (BC), AdaBoost Classifier (AD), Tree Classifier (TR), C4.5/J48 (C4), Artificial Neural Networks (ANN)

Table 1 research papers on malware analysis with machine learning methods

In conclusion, we have discussed a multitude of research papers regarding malware analysis using machine learning methods. Some papers also included topics such as data mining techniques for feature reduction [12] and clustering to create clusters or groups of varying types of malware [2].

Table 1 above displays the prominent attributes across multiple papers. A common feature is the presence of DLL's in the feature extraction of all the papers. Another common factor, apart from sources [12] and [13], are the small or tiny dataset sizes that question the validity of the above results. A multitude of the papers discussed integrated feature selection/reduction from both static and dynamic analysis. These results have yielded promising improvements over just using one set of features, but more in-depth research needs to be conducted in that field before concrete conclusions can be made. Overall of the results from the various papers, random forests performed very well when compared to the other more traditional classification algorithms. In terms of the large margin algorithms, SVM was used in nearly all papers, but generally, the boosting classifiers, Gradient and AdaBoost, did outperform it when they were employed. [11]

This literature review has showcased in-depth knowledge with regards to malware analyses and machine learning in the context of this dissertation. We have also explored multiple research papers based on the topic. We can now clearly see the need for more research into the field and have indications/trends in terms of what features and classifiers produce superior accuracy.

2.4 Selected Papers

The papers that we selected to run utilizing our dataset are Jingling Zhao et al. [10], Usukhbayar Baldangombo et al. [12], Ajit Kumar et al. [8], and Zane Markel and Michael Bilzor [13]. This decision was made due to all these papers having achieved high levels of accuracy while focusing on static features that are primarily located in the portable executable header. The decision to only pursue four papers was in light of the timeframe and computational power limits associated with the project.

Chapter 3

Dataset Construction

3.1 File Collection and Portable Executable Sorting

The dataset required both malware and benign files in order to train and test the machine learning algorithms. The acquisition of the malware files was focused on initially as access to them was restricted. The samples were collected from VirusShare [16], a security experts' community for the sharing of malicious files grouped in extensive collections. Access was provided after a lengthy two months process requiring multiple correspondences in order to establish our legitimacy. The latest eleven collections, as of November 2019, were acquired for use as part of our dataset. Table 2 showcases the selected collections and their corresponding total amount of files they contained.

Collection	Number of Files
VirusShare_00356	65,536
VirusShare_00355	65,536
VirusShare_00354	65,536
VirusShare_00353	65,536
VirusShare_00352	65,536
VirusShare_00351	65,536
VirusShare_00350	65,536
VirusShare_00349	65,536
VirusShare_00348	65,536
VirusShare_00347	65,536

VirusShare_00346	65,536

Table 2 malware collections

The total amount of suspected malicious files numbered just over 700,000, consisting of all VirsuShares malicious files posted between January 2017 and November 2019. The next step was to sort through the total collection to locate the files that were in the portable executable format. This was necessary as the collections did not have any sort of classification to them apart from the date created. This realization that the collection needed sorting came once we attempted to access the portable executable header of a subset of the collection. The collections from VirusShare come in a compressed form to reduce the download size, once unpacked, however, all files are set to be Unix executables. This is presumably a safety feature to not allow accidental execution of suspected malicious files on more common operating systems then Unix. Without the proper filename extensions, it is impossible to assume if the files were, in fact, portable executables. Utilizing python pefile to read the subset, we consistently received the same error of "e_magic not found," indicating a formatting issue. This is where the first script, named PEcheck.py, was constructed, utilizing python's pefile library to take as input a file and attempt to read the portable executable header from the file. The script would progress through an entire directory attempting to read the header from each file it found. A try and except was used to allow to script to handle the formatting error. If a file was found to be in a readable, portable executable form, it was transferred to another directory, therefore sorting the dataset for portable executables. The main functionality of the sorting script can be found in figure 8.

```
for entry in os.scandir(directory):
    try:
        pe = pefile.PE(entry, fast_load=False)
        if not os.path.exists(newpath):
            os.makedirs(newpath)
        shutil.copy(entry, newpath)
    except pefile.PEFormatError:
        continue
```

Figure 8 PEcheck.py

A crucial discovery was made after the first iteration of this script, in order to save time initially, we had set fast_load to True. This parameter allows for the pefile not to load the entire portable executable header into memory. Instead, it will only look at and load the formatting of the specific section names that are necessary for the file to be a portable executable. The entire header would only be loaded once another operation, such as dumping of the data, is called on said file. With this option set to True, the script ran at a faster rate, but due to this focus only on the proper formatting many files that had enough header formatting would pass. Once these files PE headers were fully loaded, the output would only consist of the section names but with no actual data being obtained.

3.2 VirusTotal Scanning and Reporting

With the entire collection now sorted for only readable, portable executables, a working collection of 27,307 files remained. The following step was to confirm that the files were malicious and ascertain their exact type. This process would be carried out by uploading each file to the website VirusTotal [17]. VirusTotal is a site where over 80 anti-virus programs scan files that are passed to it, the results are then centralized and can be recalled in a report format. This service that VirusTotal provides is comprehensive and widely used as a verification and classification tool for suspected malicious files. There are two methods by which submission of files for scanning can occur, a web portal or via their API. The sheer number of files that required scanning meant that it was impossible to perform the scanning operation manually, therefor a script was constructed to upload each file via the API. VirusTotal does provide a publicly accessible API key that limits the number of requests to 1,000 a day up to a maximum of 10,000 a month. A request is either an upload of a file or retrieval of a report. With just over 27,000 files, each requiring an upload and retrieval of a report, that equates to roughly 54,000 requests. To resolve this efficiency bottleneck, we contacted VirusTotal to requests, on academic grounds, a private API with a higher request limit. This was promptly handled after some back and forth via email correspondence, and our new API key allowed for up to 20,000 requests per

day and 600,000 requests per month, this being more than sufficient for the scale of our desired dataset. Figure 9 showcases a snippet from VTScan.py, the script used to upload files to VirusTotal.

```
url = 'https://www.virustotal.com/vtapi/v2/file/scan'

params = {'apikey': '819e0686e3a8ac0ca16eca7e523111fb326fd2d318749fd787c2ee5b00dd2ccc'}

rootdir = '//Volumes/MyBook/WARNING_VIRUSES_KEEP_OUT/PEMalware'

for filename in os.listdir(rootdir):
    if filename.startswith('Virus'):
        filehead = os.path.join(rootdir, filename)
        files = {'file': (filehead, open(filehead, 'rb'))}
        response = requests.post(url, files=files, params=params)
        dictresponse = response.json()
        newname = os.path.join(rootdir, dictresponse.get('md5'))
```

Figure 9 VTScan.py

The python library named “requests” was used to post the file, in bytes object form, to a specific URL with the required parameters, in this case, our private API key. The response from VirusTotal came in a JSON format shown in figure 10.


```
{'scan_id': '9649331da0834b4bf089485d9b4eb11b34d317983c9361cad7886d39885c9ba4-1586759217', 'sha1':
'6b9113fb1b0cb2b6a537d219e16d9b77fba1c599', 'resource':
'9649331da0834b4bf089485d9b4eb11b34d317983c9361cad7886d39885c9ba4', 'response_code': 1, 'sha256':
'9649331da0834b4bf089485d9b4eb11b34d317983c9361cad7886d39885c9ba4', 'permalink': 'https://www.virustotal.com/
file/9649331da0834b4bf089485d9b4eb11b34d317983c9361cad7886d39885c9ba4/analysis/1586759217/', 'md5':
'ab5f8ca2ddefc67160b631e7cb7eeed9', 'verbose_msg': 'Scan request successfully queued, come back later for the
report'}
```


Figure 10 VirusTotal JSON response

The main elements that the response was made up of were the unique scan-id, a variety of hash function codes from different hashes of the file, and a URL to the web portal showcasing that specific files report. The script from figure 9 utilized the response from the server in order to rename the file into it’s md5 hash code, one of the hash modes that were returned. This was done in order to facilitate the retrieval of the files specific report from VirusTotal. Since the only means of indicating a specific file for report retrieval was to either provide the unique scan-id, or one of the hashes of the file. The md5 hash was selected as it was the shortest both compared to the other hashing methods as to the scan-id value.

Once the scripts had completed, multiple variants were running in parallel for efficiency, the downloading and compilation of the reports per file was required. The script VTReport.py was constructed to accomplish this task, utilizing the now md5 hash named files as the resource indicator. Figure 11 is the primary function of that script.

```
for filename in os.listdir(rootdir):
    if not filename.startswith('VT'):
        params = {'apikey': '819e0686e3a8ac0ca16eca7e523111fb326fd2d318749fd787c2ee5b00dd2ccc', 'resource': filename}
        response = requests.get(url, params=params)
        writefile = open(filename + ".txt", "w")
        writefile.write(json.dumps(response.json()))
        writefile.close()
```

Figure 11 VTReport.py

With a similar approach as VTScan.py, the python library “response” is used once again, this time using get function instead of post. Once the report has been retrieved, a new text file is created with the same md5 hash being used as the name. The script then stores the JSON style report into the text file in text format. Figure 12 is taken from one of the reports of a suspected malicious file that was uploaded, and its report retrieved from VirusTotal using the methods mentioned above.


```
{"scans": {"Bkav": {"detected": true, "version": "1.3.0.9899", "result": "W32.AIDetectVM.malware2",
"update": "20200413"}, "MicroWorld-eScan": {"detected": true, "version": "14.0.409.0", "result":
"Trojan.Agent.DLYS", "update": "20200413"}, "FireEye": {"detected": true, "version": "32.31.0.0",
"result": "Generic.mg.0a4a7654567034e5", "update": "20200316"}, "CAT-QuickHeal": {"detected": false,
"version": "14.00", "result": null, "update": "20200413"}, "McAfee": {"detected": true, "version":
"6.0.6.653", "result": "Ursnif-FQIR!0A4A76545670", "update": "20200413"}, "Malwarebytes": {"detected":
true, "version": "3.6.4.335", "result": "Trojan.Injector", "update": "20200413"}, "Zillya": {"detected":
true, "version": "2.0.0.4066", "result": "Trojan.Ursnif.Win32.5054", "update": "20200411"}, "AegisLab":
{"detected": true, "version": "4.2", "result": "Trojan.Win32.Generic.4!c", "update": "20200413"},
"Sangfor": {"detected": true, "version": "1.0", "result": "Malware", "update": "20200412"}, "K7AntiVirus":
{"detected": true, "version": "11.102.33708", "result": "Trojan ( 005442341 )", "update": "20200407"},
"BitDefender": {"detected": true, "version": "7.2", "result": "Trojan.Agent.DLYS", "update": "20200413"},
"K7GW": {"detected": true, "version": "11.102.33770", "result": "Trojan ( 005442341 )", "update":
"20200413"}, "Cybereason": {"detected": true, "version": "1.2.449", "result": "malicious.456703",
"update": "20190616"}, "TrendMicro": {"detected": false, "version": "11.0.0.1006", "result": null,
"update": "20200413"}, "Baidu": {"detected": false, "version": "1.0.0.2", "result": null, "update":
"20190318"}, "F-Prot": {"detected": true, "version": "4.7.1.166", "result": "W32/S-8af6581a!Eldorado",
"update": "20200413"}, "Symantec": {"detected": true, "version": "1.11.0.0", "result":
"ML.Attribute.HighConfidence", "update": "20200412"}, "TotalDefense": {"detected": false, "version":
"37.1.62.1", "result": null, "update": "20200413"}, "APEX": {"detected": true, "version": "6.11",
"result": "Malicious", "update": "20200413"}, "Avast": {"detected": true, "version": "18.4.3895.0",
"result": "Win32:Adware-gen [Adw]", "update": "20200412"}, "ClamAV": {"detected": true, "version":
"0.102.2.0", "result": "Win.Malware.Ursnif-7001958-0", "update": "20200412"}, "GData": {"detected": true,
```


Figure 12 VirusTotal Report

Figure 12 showcases only a fraction of the report as over 80 various anti-virus programs are present in each report with each listing if the file was detected, it's version in their database, the result, and the updated version. The critical listing that is crucial for the classification stage are the result values, as these usually point at the probable type of malware. Usually, they include keywords and or names that can directly link the file to a type of malware classification.

3.3 Malware Classification

At this stage, all the reports for the just over 27,000 suspected malicious files had been downloaded and stored. The next task was to parse through the report files and, based on their report category for each anti-virus, deduce the type of said malware and categorically store it. To execute this task, we constructed a python script named Malwareclassifier.py, as can be seen in figures 13, 14, and 15. The first task was to identify a list of keywords that would be used to parse through the report files that, when located, would help indicate if that file matched that malware type. This task had previously been completed by a former Heriot-Watt MACS student named Naiyarah Adeeba Hussain [18]. Naiyarah's dissertation also revolved around malware analysis, but she did not specifically investigate the portable executable format and its file header as a primary source for static analysis. That project also required the classification of malicious files utilizing reports from VirusTotal. This word list to allow classification was used in our script; the list of keywords was taken from her projects classifymalware.py script. Figure 13 shows the exact list, and Figure 14 shows its use in our Malwareclassifier.py script.

```
trojan = " troj_gen trojan trj " #check for downloader dropper?
exploit= " exploit "
worm= " worm "
#worm bagle allaple sytro mydoom mamianune vobfus nimda chir fesber gamarue mortos palevo
backdoor=" backdoor "
#backdoor hupigon pcclient poison bifrose turkojan predator ircbot delf farfli
virus = " zherkov rehenes badda funlove vampiro kunkka trafaret troxa bleah slugin tolone hala bolzano relnek hidrag
adware = " adware " #softpulse #also contains bundlers, downloaders, droppers, installer, downware spyware
generic = " gen generic " #run last #variant? heuristic? behaveslike?
potentiallyUnwanted = " pup pua unwanted solimba installcore " #solimba unwanted? heuristic? heur?
riskware = " riskware " #hacktool
rootkit=" rootkit "
```

Figure 13 [18] Wordlist classifyMalware.py

Note for the above figure that not the entire list is shown as the virus category is cut in order to keep the figure legible.

```
with open(filehead) as f:
    trojanreg = re.findall('troj_gen|trojan|trj', f.read(), flags=re.IGNORECASE)
    exploitreg = re.findall('exploit', f.read(), flags=re.IGNORECASE)
    wormreg = re.findall('worm', f.read(), flags=re.IGNORECASE)
    backdoorreg = re.findall('backdoor', f.read(), flags=re.IGNORECASE)
    virusreg = re.findall('zherkov|rehenes|badda|funlove|vampiro|kunkka|trafaret|troxa|bleah|slugin|tolone|hala
    adwarereg = re.findall('adware', f.read(), flags=re.IGNORECASE)
    genericreg = re.findall('gen|generic', f.read(), flags=re.IGNORECASE)
    potentiallyUnwantedreg = re.findall('pup|pua|unwanted|solimba|installcore', f.read(), flags=re.IGNORECASE)
    riskwarereg = re.findall('riskware', f.read(), flags=re.IGNORECASE)
    rootkitreg = re.findall('rootkit', f.read(), flags=re.IGNORECASE)
```

Figure 14 Use of word list in Malwareclassifier.py

The script utilizes python's regular expression library named "re" to parse through and locate all the instances of the word in their respective categories. The find all function is utilized as it parses through the entire file to return any found matches in a list form. The flag IGNORECASE is invoked to allow for the matching of the regular expression in both upper and lowercase.

```
trojancount = len(trojanreg)
exploitcount = len(exploitreg)
wormcount = len(wormreg)
backdoorcount = len(backdoorreg)
viruscount = len(virusreg)
adwarecount = len(adwarereg)
genericcount = len(genericreg)
potentiallyUnwantedcount = len(potentiallyUnwantedreg)
riskwarecount = len(riskwarereg)
rootkitcount = len(rootkitreg)
```

Figure 15 Counting the number of appearances of regular expression matches per category from Malwareclassifier.py

Once the lists of the regular expression matches had been created for each category, the length of the list was proportional to the number of matches. This length was then used in figure 16's, using python's max method, to assign the files category based on the largest number of regular expression matches from parsing through the report file.

```
countdict = {'unknown' : 0, 'trojan': trojancount, 'adware': adwarecount, 'backdoor': backdoorcount,

mosttype = max(countdict, key=countdict.get)

if mosttype == 'trojan':
    shutil.move(filehead, Trojandir)
elif mosttype == 'adware':
    shutil.move(filehead, Adwaredir)
elif mosttype == 'backdoor':
    shutil.move(filehead, Backdoordir)
elif mosttype =='exploit':
    shutil.move(filehead, Exploitdir)
elif mosttype == 'generic':
    shutil.move(filehead, Genericdir)
```

Figure 16 Locating the maximum number of matches in a category and assigning the file to the respective category in Malwareclassifier.py.

The results of the classification of the 27,307 files using the method, as mentioned above, indicated that a significant majority were of the Trojan based malware variety. Twenty-two thousand seven hundred thirty-seven were classified as Trojans, 4,567 were classified as unclassified, and three were classified as worms. The unclassified category indicates that the files, while still being malicious, did not cause any of the regular expression to match. This could be indicative of only a partial word list or limited data in the response file from VirusShare. A large number of Trojans is not very surprising as the most common malware type are Trojans, and it is possible that due to filtering for only portable executable files that much of that subset were indeed Trojans. The final composition of the Malware dataset comprised of 20,107 Trojans, 3 Worms, and 2,690 unidentified malicious files, this totaled a dataset of 22,800 malware.

The construction of the benign dataset was much more straightforward, as no classification was necessary as the original file name extensions were available. This allowed for easy identification of files in the portable executable format. All benign files were collected from a fresh install of Windows 10 32-bit OS version 10.0.18363, Windows 8 32-bit OS, and Windows 7 32-bit OS Home Basic SVP1. This was done to ensure that the files collected were in-fact, benign. All major portable executable types were collected, as shown in table 3, this was done to ensure variety.

Filename Extension	Number of Files
.acm	18
.ax	54
.cpl	60
.dll	14,265
.drv	45
.efi	30
.exe	2,632
.mui	7,789
.ocx	30
.scr	41
.sys	1,712
.tsp	27

Table 3 Benign files collected

In total, 26,703 files were collected from the Windows OS; usually this would have been our benign file collection; however, during reading of some of the benign files with python's pefile, we encountered formatting errors. After running the PEcheck.py on the benign dataset, several files were found not to be readable using our method for header extraction. This resulted in the benign dataset having to shrink down to a total of 22,800 files. This resulted in the final dataset file count for both malicious and benign files being 45,600. This was done on purpose to have a non-bias, equal, dataset.

Chapter 4

Methodology and Implementation

4.1 Methodology

The primary objective of this project was to investigate various research papers proposals on the use of static portable executable header attributes to train and test with machine learning algorithms to perform classification problems. In order to accomplish this task, we understood that the machine learning algorithm would be provided with a CSV file with all the necessary attributes. With python's pefile module, the entire portable executable header could be loaded and displayed in either a python dictionary form or a text-based form. Initially, it was theorized, and small scale experimented to see if, with a research paper specific parser, we could immediately go from raw data to the necessary CSV file. This proved to be very inefficient as the entire dataset would need to be parsed in its entirely to create just one CSV file. Additionally, the number of similar scripts performing nearly the same function would be proportional to the number of papers tested.

The solution was to instead of going direct from raw data to the CSV file that there would be a data formatting step. The raw data would be parsed only once and sorted into an iterable data format from which the CSV files could be generated more efficiently. The data format chosen was JSON due to its extensive use and open format usage policies [19]. The initial design to transform the raw data into JSON was to extract the files into the python dictionary form. The python JSON library has a method to allow direct input of a dictionary and will transform that into a JSON compliant file. This failed as the data dictionary that was created from python's pefile included elements in binary form. In the JSON format, all iterable elements must be human-readable text form and not binary; therefore, the direct transformation was not possible. The solution was a custom parser that would extract most of the portable executable into an organized set of nested dictionaries. These dictionaries were directly

transformed into a JSON file using the JSON library. Two separate JSON files were created one for the malware and the other for the benign files.

The next step was to create the CSV files from the necessary attributes within each JSON file. This is where a custom script had to be developed for each research paper as they each had their attribute requirements. The design of the CSV files was standardized to such that each line of the file would be a separate file, either benign or malicious, with columns indicating all the various attributes necessary. The very first column would denote whether that line was either benign or malicious, with each proceeding column indicating the result for that attribute. The problem of missing information was irrelevant as all files had been parsed for the same information. Therefore any information not found was not missing but just not available. In case of information being not available, a zero would indicate it on the CSV file.

The final step included the creation of scripts that would execute the required machine learning algorithm based on the selected research paper. The primary python library used was the scikit-learn library, a popular python machine learning library with support for all necessary algorithms. The scripts would also produce the confusion matrix and accuracy reports as necessary by the functional requirements of this dissertation.

4.2 Implementation

4.2.1 Text Header Files

With the dataset now entirely constructed, we had to extract the portable executable header for each file into textual form. Once again, we would utilize python's pefile library to load the file's header and now store that data into a text file of the same name. The script created to do this is PEextracttext.py, as seen in figure 17.

```
pe = pefile.PE(filehead, fast_load=False)
info = pe.dump_info()
try:
    with open(filename + ".txt", 'w') as textfile:
        textfile.write(info)
```

Figure 17 PEextracttext.py

The output of this script is seen below in figure 18. This is the direct textual output of a portable executable file header.

```
----------Parsing Warnings----------

Byte 0x00 makes up 70.2601% of the file's contents. This may indicate truncation / malformation.

----------DOS_HEADER----------

[IMAGE_DOS_HEADER]
0x0        0x0   e_magic:                       0x5A4D
0x2        0x2   e_cblp:                        0x90
0x4        0x4   e_cp:                          0x3
0x6        0x6   e_crlc:                        0x0
0x8        0x8   e_cparhdr:                     0x4
0xA        0xA   e_minalloc:                    0x0
0xC        0xC   e_maxalloc:                    0xFFFF
0xE        0xE   e_ss:                          0x0
0x10       0x10  e_sp:                          0xB8
0x12       0x12  e_csum:                        0x0
0x14       0x14  e_ip:                          0x0
0x16       0x16  e_cs:                          0x0
0x18       0x18  e_lfarlc:                      0x40
0x1A       0x1A  e_ovno:                        0x0
0x1C       0x1C  e_res:
0x24       0x24  e_oemid:                       0x0
0x26       0x26  e_oeminfo:                     0x0
0x28       0x28  e_res2:
0x3C       0x3C  e_lfanew:                      0xF0

----------NT_HEADERS----------

[IMAGE_NT_HEADERS]
0xF0       0x0   Signature:                     0x4550

----------FILE_HEADER----------

[IMAGE_FILE_HEADER]
0xF4       0x0   Machine:                       0x14C
0xF6       0x2   NumberOfSections:              0x6
0xF8       0x4   TimeDateStamp:                 0x57BF2593 [Thu Aug 25 17:06:27 2016 UTC]
0xFC       0x8   PointerToSymbolTable:          0x0
0x100      0xC   NumberOfSymbols:               0x0
0x104      0x10  SizeOfOptionalHeader:          0xE0
0x106      0x12  Characteristics:               0x10F
Flags: IMAGE_FILE_32BIT_MACHINE, IMAGE_FILE_EXECUTABLE_IMAGE, IMAGE_FILE_LINE_NUMS_STRIPPED, IMA

----------OPTIONAL HEADER----------
```

Figure 18 PEextracttext.py output

4.2.2 Parsing to JSON

With all the files having their header extracted into the text format, the next stage was to parse the entire header for each file to build the JSON files. This task was accomplished by the parser.py script as shown in figures 19,20,21,22,23, and 24 below.

Figure 19 parser.py Regular Expression Section

A class was set up to store all the regular expressions used to parse through the text file. All regular expressions were created and tested multiple times to ensure the accurate capture of the intended information. The regular expressions are sectioned off based on their sections in the header text file. Each section also has a regular expression to detect the start of that section, this is required as sections can be repeated, and the parser must be aware when a new section has started. The regular expressions were designed whereby the required information was stored in a group separate from the search parameter. This is seen as the "()" brackets denote a group within that regular expression. Therefore, if the line "e_magic: hello" were parsed with the reg_e_magic regular expression in figure 19, group 0 would include the entire match: "e_magic: hello" but group 1 would only include "hello." This was crucial to extract the exact information from the document.

```
def __init__(self, line):
    # check whether line has a positive match with all of the regular expressions
    self.image_dos_header = self._reg_image_dos_header.search(line)
    self.e_magic = self._reg_e_magic.search(line)
    self.e_cblp = self._reg_e_cblp.search(line)
    self.e_cp = self._reg_e_cp.search(line)
    self.e_crlc = self._reg_e_crlc.search(line)
    self.e_cparhdr = self._reg_e_cparhdr.search(line)
    self.e_minalloc = self._reg_e_minalloc.search(line)
    self.e_maxalloc = self._reg_e_maxalloc.search(line)
    self.e_ss = self._reg_e_ss.search(line)
    self.e_sp = self._reg_e_sp.search(line)
    self.e_csum = self._reg_e_csum.search(line)
    self.e_ip = self._reg_e_ip.search(line)
    self.e_cs = self._reg_e_cs.search(line)
    self.e_lfarlc = self._reg_e_lfarlc.search(line)
    self.e_ovno = self._reg_e_ovno.search(line)
    self.e_oemid = self._reg_e_oemid.search(line)
    self.e_oeminfo = self._reg_e_oeminfo.search(line)
    self.e_lfanew = self._reg_e_lfanew.search(line)
    self.image_nt_headers = self._reg_image_nt_headers.search(line)
```

Figure 20 parser.py Regular Expression Calls

In figure 20, the regular expressions set up in figure 19 are evoked by providing their search parameter and the line to search. The search parameters used from python's “re” library consisted of two options, search which would only return the first valid match. Alternatively, find all whereby all matches in the line provided were returned.

```
for entry in os.listdir(directory):
    if entry.endswith('.txt'):
        filehead = os.path.join(directory, entry)
        with open(filehead) as file:
            for line in file:
                line = line.strip()
                reg_match = _RegExLib(line)

                if reg_match.image_dos_header:
                    i = 1

                if reg_match.e_magic and i == 1:
                    e_magic = reg_match.e_magic.group(1)
                    dictdos['e_magic'] = e_magic

                if reg_match.e_cblp and i == 1:
                    e_cblp = reg_match.e_cblp.group(1)
                    dictdos['e_cblp'] = e_cblp

                if reg_match.e_cp and i == 1:
                    e_cp = reg_match.e_cp.group(1)
                    dictdos['e_cp'] = e_cp
```

Figure 21 parser.py Start of For Loop and Storage of Keys and Values into Section Headers

As with previous scripts parser.py iterates through all files in the given directory using python’s os library. Figure 21 shows that additionally to iterating through all files, parser.py also iterates line by

line through the file. This is due to the regular expressions needing to be matched by line and not throughout the file. This is key as the position of a match to a regular expression is just as important as the value the match returns. Knowing in which section a match occurs is crucial for the JSON file as it will follow the following structure format: file, section, header, and individual attributes. The "i" value seen in figure 21 under the first if statement is used as section control throughout parser.py. Many of the regular expressions will match at various points throughout the header file, as various sections can have the same attributes. An example of one of these is the name attribute that appears both in the PE-sections section as well as the Imported Symbols section. The "i" value only allows for the regular expressions to match if the match is made within their designated section.

```
if reg_match.Size and i == 20:
    Size = reg_match.Size.group(1)
    dictres['Size'] = Size
    dictheader['IMAGE_DIRECTORY_ENTRY_RESERVED'] = dictres
    tempdir = copy.deepcopy(dictheader)
    dictsect['Directories'] = tempdir
    dictheader.clear()
# --------------------------------
if reg_match.image_import_descriptor:
    i = 22
    if dll == 1:
        dictimp[DLLType] = Listdll
        Listimp.insert(e, copy.deepcopy(dictimp))
        dictheader[Name] = Listimp[e]
        Listdll.clear()
        dictimp.clear()
        e = e + 1
        dll = 0
    else:
        e = 0

if reg_match.OriginalFirstThunk and i == 22:
    OriginalFirstThunk = reg_match.OriginalFirstThunk.group(1)
    dictimp['OriginalFirstThunk'] = OriginalFirstThunk
```

Figure 22 parser.py Local Dictionary Storing into Section Dictionary and Image Import Descriptor

Once the local section dictionary was constructed, that dictionary was then pushed into the header dictionary "dictheader." This can be seen in figure 22 above, an issue that arose was due to our final dictionary going to consist of four nested dictionaries. In an earlier implementation, "dictheader" was always being linked to "dictsect" as this was part of the four nested dictionaries. The problem was that due to "dictheader" constantly receiving the local section dictionaries, it would append itself, even previous instances under different keys with "dictsect." This resulted in each key in "dictsect" having

the same nested dictionary as "dictheader" was used for all. One solution to this problem was to create individual "dictheaders" with separate names for each section. However, since parser.py already consists of 30 dictionaries, we decided instead to make local temp copies of "dictheader" and save those individually into "dictsect." This is done using the deepcopy method from the python library copy, deepcopy ensures that the new dictionary is separate from the original. Some sources online suggest that the deepcopy method comes with a performance penalty when compared with just iterating and reassigning from the original dictionary to the new one [20]. In the second, if statement of figure 22, it showcases the code written incase a section repeats itself but must still be organized under the same section in "dictsect." In this example, dll is set to 1 once the last required element of a single image_import_descriptor is matched. Initially, dll is set to 0, but after the first iteration of this section loop, it gets set to 1. If the same section loop is called again with dll now set to 1, a separate key is created in "dictheader" for each iteration. Once a different section is read, the "dictheader" storing the various iteration is copied, and that copy is stored in "dictsect."

```
Listsect[final] = copy.deepcopy(dictsect)
dictfinal[entry] = Listsect[final]
final = final + 1
```

Figure 23 parser.py Final Dictionary Appending

After all lines in the header file have been parsed through the entire "dictsect" is assigned into the "dictfinal", with the key being the header file name, this is shown above in figure 23. Once again, the issue of dictionaries appending themselves is prevalent here, and instead of creating individual "dictsect" lists per file, we opted to utilize the deepcopy method again. The "dictfinal" dictionary will store all of the file dictionaries with their parsed information embedded into the four nested dictionary structure. In figure 24 below, the "dictfinal" is used in the python library JSON dump method to create the JSON file.

```
# creating json file
out_file = open("Malware.json", "w")
json.dump(dictfinal, out_file, indent=4)
out_file.close()
```

Figure 24 parser.py Creation of the JSON File from "dictfinal."

Figure 25 parser.py JSON Output

The output JSON file structure of parser.py can be seen in figure 25 above. Each indentation represents a nested dictionary from parser.py in JSON formatting. Parser.py was repeated for both the malware and the benign file collections and generated two JSON files, Malware.json and Benign.json. To help visualize the scope of these JSON files, Benign.json has just under 100 million lines in it, and Malware.json consists of an exuberant 128 million lines.

4.2.3 CSV Files Creation

With the JSON files processed, the task of creating the individual CSV files was now paramount. The scripts to perform such operations would be uniquely customized to their research paper as they all had variation in exactly which attributes they utilized. The primary objective for all these scripts was to be able to retrieve specific information from the JSON files and, if not present, represent that nonpresence with a 0.

```
column = ['Files', 'Class', 'SizeOfInitializedData', 'NumberOfSymbols', 'exportTableSize', 'IATSize', 'BoundImportSize',
          'BaseRelocationTableSize', 'LoadConfigTableSize',
          'e_cp', 'e_cblp', 'e_minalloc', 'e_ovno', 'KERNEL32', 'GDI32', 'USER32', 'COMCTL32', 'COMDLG32', 'WS2_32',
          'ADVAPI32', 'NETAPI32', 'OLEAUT32', 'OLE32', 'SHELL32', 'kernel32',
          'gdi32', 'user32', 'comctl32', 'comdlg32', 'ws2_32', 'advapi32', 'netapi32', 'oleaut32', 'ole32', 'shell32']
```

Figure 26 Baldangombo.py Column List

In the cases where papers only required a set number of attributes, the “columns” list would be constructed at the start of the script. This is visible in figure 26, seeing as that paper required only some static values from certain sections and the counting of the various DLL types that each file calls.

```
columntemp = copy.deepcopy(column)
columntemp[0] = file
columntemp[1] = 1  # This means malware!

if "DOS_HEADER" in sections:
    sectionstemp = sections["DOS_HEADER"]
    for headers, values in sectionstemp.items():
        if "e_cp" in values:
            x = int(values["e_cp"], 16)
            columntemp[9] = x
        if "e_cblp" in values:
            x = int(values["e_cblp"], 16)
            columntemp[10] = x
        if "e_minalloc" in values:
            x = int(values["e_minalloc"], 16)
            columntemp[11] = x
        if "e_ovno" in values:
            x = int(values["e_ovno"], 16)
            columntemp[12] = x
```

Figure 27 Baldangombo.py Data Collection

In figure 27 above showcases the standard data collection process for research papers with static attributes. “Columntemp” copies the column list from figure 26 and will systematically replace any

sections that it locates within the files JSON section. The first list element is set to the file name for organizations sake. The second is set to either zero or one depending on if the JSON list that is being utilized is the malware or benign version. This is used to tell the machine learning algorithm what class each file or row belongs to. The if statement visible in figure 27 shows the parsing of the JSON file to retrieve values in the "DOS_HEADER" section. If the values are located, they are assigned to their respective slot in the “columntemp” list. This is repeated for as many attributes as necessary, with figure 28 showcasing the process but with the DLL types that need counting. This is due to the paper requesting the number of times each DLL type is invoked.

```
if "Imported_Symbols" in sections:
    sectionstemp = sections["Imported_Symbols"]
    for headers, values in sectionstemp.items():
        if "KERNEL32" in values:
            for i in values["KERNEL32"]:
                j = j + 1
            columntemp[13] = j
            j = 0
        if "GDI32" in values:
            for i in values["GDI32"]:
                j = j + 1
            columntemp[14] = j
            j = 0
```

Figure 28 Baldangombo.py Data Collection Number of Times Invoked

```
list_columntemp = iter(columntemp)
# next(list_columntemp)
z = 2
for z, num in enumerate(list_columntemp):
    if num in column:
        columntemp[z] = 0

writer.writerow(columntemp)

columntemp.clear()
sectionstemp.clear()
```

Figure 29 Baldangombo.py Setting Information Not Available and Writing Rows

As discussed in section 4.1, any information that is not located by the CSV creating scripts must be classified as not available. This is done by inputting zero into that respective attribute value. Figure 29

shows that process by first skipping the first two list elements and then comparing between "comlumntemp" and "column" lists. Since "columntemp" is set to column at the start of parsing through the JSON file, if any of the list attributes in "columntemp" are the same as in "column" then the information has not been found. Zero is then overwritten into "columntemp" to indicate such an event. This exact process from figures 27 through 29 is repeated twice in the same script, once for the Malware.json and once for the Benign.json. The one difference is that the second list element, indicating class, of “columntemp” is set to either a zero or a one.

Files	Class	SizeOfInit	NumberO	exportTat	IATSize	BoundImp	BaseReloc	LoadConfi	e_cp	e_cblp	e_minallo	e_ovno	KERNEL32	GDI32
0b5ed486	1	0x2FE00	0x0	0x34	0x21C	0x0	0x1F8C	0x40	0x3	0x90	0x0	0x0	0	0
191cb75a	1	0x31000	0x0	0x0	0x344	0x0	0x50EC	0x40	0x3	0x90	0x0	0x0	100	8
531a1c20	1	0x31000	0x0	0x0	0x344	0x0	0x50EC	0x40	0x3	0x90	0x0	0x0	100	8
79cd27ce	1	0x31000	0x0	0x0	0x344	0x0	0x50EC	0x40	0x3	0x90	0x0	0x0	100	8
0dd7461b	1	0x5D800	0x0	0x0	0x0	0x0	0x10	0x0	0x2	0x50	0xF	0x1A	0	0
84b486c0	1	0x170400	0x0	0x0	0x14C	0x0	0x4854	0x40	0x3	0x90	0x0	0x0	82	0
db653fd2	1	0x31000	0x0	0x0	0x344	0x0	0x50EC	0x40	0x3	0x90	0x0	0x0	100	8
9f4c97a05	1	0x2000	0x0	0x0	0x1B4	0x20	0x0	0x0	0x3	0x90	0x0	0x0	0	0
6a8ecd7f8	1	0x2C9400	0x0	0x0	0x1BC	0x0	0x0	0x40	0x3	0x90	0x0	0x0	105	0
0b18f3905	1	0x1000	0x0	0x0	0x0	0x0	0x0	0x0	0x3	0x90	0x0	0x0	0	0
25748a7e7	1	0x2E800	0x0	0x0	0x2E4	0x0	0x0	0x40	0x3	0x90	0x0	0x0	77	8
51c24c1d5	1	0x31000	0x0	0x0	0x344	0x0	0x50EC	0x40	0x3	0x90	0x0	0x0	100	8
1e35223d2	1	0x31000	0x0	0x0	0x344	0x0	0x50EC	0x40	0x3	0x90	0x0	0x0	100	8
1cb315728	1	0x1B9800	0x0	0x0	0x1D8	0x0	0x0	0x40	0x3	0x90	0x0	0x0	102	0
866e9b837	1	0x160800	0x0	0x0	0x0	0x0	0x7E84	0x0	0x2	0x50	0xF	0x1A	0	0
2d216c762	1	0x48D000	0x0	0x0	0x1D4	0x0	0x0	0x40	0x3	0x90	0x0	0x0	0	0
2fefe7369	1	0x95800	0x0	0x0	0x134	0x0	0x0	0x40	0x3	0x90	0x0	0x0	73	0
65130fdc9	1	0x1D7A00	0x0	0x0	0x184	0x0	0x0	0x40	0x3	0x90	0x0	0x0	75	0
9bde42a9	1	0x79000	0x0	0x615	0x850	0x0	0x0	0x0	0x3	0x90	0x0	0x0	196	41

Figure 30 Baldangombo.py Part of the CSV Output File

The generation of CSV files with a dynamic attribute list presents a more significant challenge than the previously showcased script and example. The primary use case for this dynamically attributes list is if a research paper wishes to showcase if a file calls a variety of Flags. To do such a task with a static attributes list would require manual parsing of both JSON files to look for all the various Flag names. Instead, we modified the previous script to first parse through both JSON files. Each time a new/unique Flag name was located, it was added to a local list before being added to "column" list. Figure 31 below showcases part of that code.

```
for file, sections in data.items():
    # print(file)
    if "FILE_HEADER" in sections:
        sectionstemp = sections["FILE_HEADER"]
        for headers, values in sectionstemp.items():
            if "Flags" in values:
                Flags.extend(values["Flags"])
```

Figure 31 Markel.py

During the parsing of the JSON files to locate if the Flag name was present in each file, if located then a one would be added to “columntemp” in that index spot. This is shown in figure 32 below.

```
if "FILE_HEADER" in sections:
    sectionstemp = sections["FILE_HEADER"]
    for headers, values in sectionstemp.items():
        if "Flags" in values:
            for i in values["Flags"]:
                if (i in columntemp):
                    columntemp[columntemp.index(i)] = 1
```

Figure 32 Markel.py Locating Flag Name in JSON File

4.2.4 Machine Learning Scripts

The final scripts that were constructed were the scripts to execute the machine learning algorithms on the selected CSV file. Once again, there are a variety of variations to the scripts due to the actual machine learning algorithm used and the CSV file selected as the number of attributes would vary. Figure 33 below shows the entire Naïve Bayes machine learning algorithm for Zhao.csv.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, f1_score

dataset = pd.read_csv('//Users/cedricecran/Desktop/Zhao.csv')

X = dataset.iloc[:, 2:25].values #Attributes
y = dataset.iloc[:, 1].values #Class (benign or malware)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=25)

regressor = GaussianNB()
regressor.fit(X_train, y_train)
y_pred = regressor.predict(X_test)

print (confusion_matrix(y_test, y_pred))
print (classification_report(y_test, y_pred))
print (accuracy_score(y_test, y_pred))
```

Figure 33 ZhaoNaiveBayes.py

The python libraries used for all the machine learning scripts were pandas for CSV file reading and scikit-learn to split the data into training and testing sets, implement the machine learning algorithm and display the confusion matrix and accuracy score. The primary differences between the script variations were in the number of attributes, in figure 33 the number of attributes is 25, and the exact machine-learning algorithm to apply. In figure 33, this is what the variable "regressor" is set to; in this case, it is "GaussianNB()" for Naïve Bayes.

Chapter 5

Results and Evaluation

5.1 Results Analysis

Each research paper reviewed in this section will be evaluated based on the parameters set by the functional requirements of this dissertation. This includes the accuracy, the generated confusion matrix, false positives percentage, and the false negative percentage. Due to the scope of this dissertation is limited to the portable executable file header, only specific attributes within that scope of each research paper were utilized. This may be a source of inconsistency in the achieved results versus their reported results when comparing the results achieved overall papers; however, that inconsistency will decrease as the same restrictions were applied to all tested papers. For all results, the dataset was always split to a 70/30 ratio for training and testing with a random state modifier of 25. This should prevent the model from becoming overfit as the random state ensures a random seed is added to the number generator when deciding which 30% of the dataset will be used for testing.

5.1.1 Zhao

The explored attributes in the paper by Jingling Zhao et al. [10] were the frequency and various types of names all DLL's in a given file. Two machine learning algorithms were employed; these were Naïve Bayes and Support Vector Machines (SVM). The results by Jingling Zhao et al. utilizing the Naïve Bayes classification model are shown below in figure 33:

Category	Precision	Recall	F1 score
Malware	0.97	0.96	0.96
Benign	0.92	0.94	0.93
Mean	0.95	0.95	0.95

Figure 34 [10] Accuracy of Naïve Bayes classification

Our recorded results are shown below in figure 34.

```
[[6770   34]
 [5345 1531]]
              precision    recall  f1-score   support

           0       0.56      1.00      0.72      6804
           1       0.98      0.22      0.36      6876

    accuracy                           0.61     13680
   macro avg       0.77      0.61      0.54     13680
weighted avg       0.77      0.61      0.54     13680

0.6067982456140351
```

Figure 35 The recorded Confusion Matrix and Accuracy for the Naïve Bayes Classification

The recorded accuracy was (rounded) 60.68%, with a false positive percentage of 0.5% and a false negative score of 79%.

The posted results for the SVM classification model are shown below in figure 35:

Category	Precision	Recall	F1 score
Malware	0.98	0.95	0.96
Benign	0.90	0.95	0.92
Mean	0.94	0.95	0.94

Figure 36 [10] Accuracy of the SVM classification

The recorded results are shown below in figure 36.

```
[[6334  470]
 [1939 4937]]
              precision    recall  f1-score   support

           0       0.77      0.93      0.84      6804
           1       0.91      0.72      0.80      6876

    accuracy                           0.82     13680
   macro avg       0.84      0.82      0.82     13680
weighted avg       0.84      0.82      0.82     13680

0.8239035087719299
```

Figure 37 The recorded Confusion Matrix and Accuracy for SVM classification

The recorded accuracy was 82.39%, with a false positive percentage of 6.9% and a false negative score of 28.2%.

The loss in accuracy can be mainly accredited to the fact that only the features located within the portable executable header were utilized, not any of the string counts. These results showcase that Jingling Zhao et al. primary attributes for classification were the string's located in the files and not the DLL counts as measured in our testing.

5.1.2 Baldangombo

Usukhbayar Baldangombo et al. [12] utilized a combination of DLL count and raw PE header fields in their feature selection. In our testing, we utilized their exact features and DLL count technique but utilizing our balanced dataset. The results from the paper and our own are shown below.

Feature type	Classifier	DR (%)	FPR (%)	OA (%)
PE header&DLLs	Naïve-Bayes	95	8.9	72
	SVM	96	7.3	85
	J48	97.1	4.7	90

Figure 38 [12] Published Results

```
[[6782   22]
 [6694  182]]
              precision    recall  f1-score   support

           0       0.50      1.00      0.67      6804
           1       0.89      0.03      0.05      6876

    accuracy                           0.51     13680
   macro avg       0.70      0.51      0.36     13680
weighted avg       0.70      0.51      0.36     13680

0.5090643274853801
```

Figure 39 The recorded Confusion Matrix and Accuracy for the Naïve Bayes classification

The recorded accuracy was 50.9%, with a false positive percentage of 0.32% and a false negative score of 2.64%.

```
[[6636  168]
 [ 295 6581]]
              precision    recall  f1-score   support

           0       0.96      0.98      0.97      6804
           1       0.98      0.96      0.97      6876

    accuracy                           0.97     13680
   macro avg       0.97      0.97      0.97     13680
weighted avg       0.97      0.97      0.97     13680

0.9661549707602339
```

Figure 40 The recorded Confusion Matrix and Accuracy for the Decision Tree (CART version) classification

The recorded accuracy was 96.6%, with a false positive percentage of 2.47%, and a false negative score of 4.29%.

The SVM classifier was attempted utilizing our dataset but unfortunately proved to be computationally infeasible.

The collected results utilizing our dataset and the naïve Bayes classifier fell much short of the reported results by Usukhbayar Baldangombo et al. [12]. This could be due to the very bias dataset utilized by Usukhbayar Baldangombo et al. When comparing the results of the Decision Tree classifier, our accuracy percentage was 6.6% higher, and our false positive percentage was 2.23% lower. These results suggest that the Decision Tree classifier may handle dataset bias more efficiently than the Naive Bayes classifier.

5.1.3 Markel

Zane Markel and Michael Bilzor [13] utilized only raw data located in the portable executable header. In attempting to replicate their method as closely as possible, we only had access to their top ten extracted features measured by F-score. Along with these, there was a paper specific attribute named HighEntropy. HighEntropy is set to True if any section of a file has an entropy of greater than seven.

malprev	Naive Bayes	Decision Tree (CART)	Logistic Regression
(0.5, 0.5)	0.5127	0.9792	0.9456
(0.1, 0.1)	0.4640	0.9270	0.7941
(0.5, 0.1)	0.4804	0.9150	0.7905
(0.01, 0.01)	0.4250	0.7581	0.3023
(0.5, 0.01)	0.3342	0.5247	0.2873
(0.001, 0.001)	0.0493	0.4157	0.03124
(0.5, 0.001)	0.0697	0.1193	0.04857

Figure 41 [13] Published Results

The highest obtained accuracies for all three classifiers used were with a malprev, a tuple containing both test and training samples, of 0.5 for both samples.

```
[[6706   98]
 [6128  748]]
              precision    recall  f1-score   support

           0       0.52      0.99      0.68      6804
           1       0.88      0.11      0.19      6876

    accuracy                           0.54     13680
   macro avg       0.70      0.55      0.44     13680
weighted avg       0.70      0.54      0.44     13680

0.5448830409356725
```

Figure 42 The recorded Confusion Matrix and Accuracy for the Naïve Bayes classifier

The recorded accuracy was 54.48%, with a false positive percentage of 1.44%, and a false negative score of 89.12%.

```
[[6729   75]
 [ 109 6767]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      6804
           1       0.99      0.98      0.99      6876

    accuracy                           0.99     13680
   macro avg       0.99      0.99      0.99     13680
weighted avg       0.99      0.99      0.99     13680

0.9865497076023392
```

Figure 43 The recorded Confusion Matrix and Accuracy for the Decision Tree (CART version) classification

The recorded accuracy was 98.65%, with a false positive percentage of 2.67%, and a false negative score of 1.57%.

Zane Markel and Michael Bilzor [13], in addition to Naïve Bayes and the Decision Trees classifiers, also utilized Linear Regression. Linear Regression is generally used for machine learning regression problems, not classification. The paper also does not specify in which capacity this accuracy result was achieved as it would involve using another test sample and creating predictions based on the regression model built. Due to these circumstances, the linear regression was not implemented. The abysmal false negative percentage for the Naïve Bayes classifier is very high and very similar to our achieved results utilizing Ajit Kumar et al. [8] methodology. This could indicate that some of the selected features since there are only 11, do not allow for a proper distinction between our datasets benign and malicious files. This, however, is not supported by the data for the Decision Tree classifier, where the false-negative rate was only 1.57%.

5.1.4 Kumar

Ajit Kumar et al. [8] extracted raw portable executable header features and then derived features from the extracted raw data. The results of their classification algorithms are provided in both raw and integrated format. In this paper, with the focus being on raw header features and other papers not deriving integrated features, we have purposely only utilized the published raw feature set.

Classifier	Accuracy		Precision		Recall		F1-score	
	RawF	IntF	RawF	IntF	RawF	IntF	RawF	IntF
LR	77.06	78.12	0.81	0.80	0.77	0.78	0.76	0.77
LDA	91.71	92.45	0.92	0.93	0.92	0.92	0.92	0.92
RF	97.43	98.78	0.97	0.99	0.97	0.99	0.97	0.99
DT	96.47	97.12	0.96	0.97	0.96	0.97	0.96	0.97
NB	56.04	50.09	0.74	0.77	0.56	0.58	0.48	0.51
kNN	94.73	90.79	0.95	0.91	0.95	0.91	0.95	0.91

Figure 44 [8] Published Results

The reported results utilizing only the raw feature set are reported in figure 44 under "RawF."

```
[[4864 1940]
 [ 188 6688]]
              precision    recall  f1-score   support

           0       0.96      0.71      0.82      6804
           1       0.78      0.97      0.86      6876

    accuracy                           0.84     13680
   macro avg       0.87      0.84      0.84     13680
weighted avg       0.87      0.84      0.84     13680

0.8444444444444444
```

Figure 45 The recorded Confusion Matrix and Accuracy for the Linear Discriminant Analysis classification

The recorded accuracy was 84.44%, with a false positive percentage of 28.51 % and a false negative score of 2.73%.

```
[[6762   42]
 [  47 6829]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      6804
           1       0.99      0.99      0.99      6876

    accuracy                           0.99     13680
   macro avg       0.99      0.99      0.99     13680
weighted avg       0.99      0.99      0.99     13680

0.9934941520467836
```

Figure 46 The recorded Confusion Matrix and Accuracy for the Random Forests classifier

The recorded accuracy was 99.35%, with a false positive percentage of 0.62 % and a false negative score of 0.68%.

```
[[6738   66]
 [  68 6808]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      6804
           1       0.99      0.99      0.99      6876

    accuracy                           0.99     13680
   macro avg       0.99      0.99      0.99     13680
weighted avg       0.99      0.99      0.99     13680

0.9902046783625731
```

Figure 47 The recorded Confusion Matrix and Accuracy for the Decision Tree classification

The recorded accuracy was 99.0%, with a false positive percentage of 0.97 %, and a false negative score of 0.99%.

```
[[6749   55]
 [6094  782]]
              precision    recall  f1-score   support

           0       0.53      0.99      0.69      6804
           1       0.93      0.11      0.20      6876

    accuracy                           0.55     13680
   macro avg       0.73      0.55      0.44     13680
weighted avg       0.73      0.55      0.44     13680

0.5505116959064328
```

Figure 48 The recorded Confusion Matrix and Accuracy for the Naïve Bayes classifier

The recorded accuracy was 55.05%, with a false positive percentage of 0.80%, and a false negative score of 88.62%.

```
[[6619  185]
 [ 196 6680]]
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      6804
           1       0.97      0.97      0.97      6876

    accuracy                           0.97     13680
   macro avg       0.97      0.97      0.97     13680
weighted avg       0.97      0.97      0.97     13680

0.9721491228070176
```

Figure 49 The recorded Confusion Matrix and Accuracy for the k-Nearest Neighbor classifier

The recorded accuracy was 97.21%, with a false positive percentage of 2.72%, and a false negative score of 2.85%.

Utilizing our dataset with the methodology of Ajit Kumar et al. [8], we achieved better accuracy scores utilizing the Random Forests, Decision Trees, and k-Nearest Neighbor classifiers. The results of the Naïve Bayes classifier were very similar to Ajit Kumar et al. posted results. The false-negative score was once again exceptionally high at 89.62%, this is comparable to Zane Markel, and Michael Bilzor [13], whereby the achieved false-negative score for Naive Bayes utilizing our dataset was 89.12%. This may indicate a problem with utilizing the Naïve Bayes classifier with raw header data as both papers feature sets include much of that type of data. The accuracy result collected for Linear Discriminant Analysis fell 7% below the reported accuracy by Ajit Kumar et al. The use of a larger dataset is most likely the prime factor as to why some classifiers achieved overall higher performance.

5.2 Recombination

With the achieved results compiled, we decided to attempt the recombination of the two best performing tested papers. These were Zane Markel, and Michael Bilzor [13] and Ajit Kumar et al. [8], the classification algorithm used was the Random Forests classifier as it achieved the highest accuracy

out of all tested classifiers. The process of recombining the two papers was rather simple as the differences in utilized features were only minor. We utilized the entire feature set of Ajit Kumar et al. [8] and added the specific features that Zane Markel and Michael Bilzor [13] were utilizing. These added features were the “Size_Of_Code” attribute located in the optional header, all the possible “Flags” present in a files file header section, and the Zane Markel and Michael Bilzor specific attribute called "HighEntropy." The Random Forest classifier was set to 100 n estimators, and the data was split 70/30 with a random state of 25. This is the same configuration utilized during the collection of each tested paper. The results of our recombination are shown below in figure 50.

```
[[6770   34]
 [  43 6833]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      6804
           1       1.00      0.99      0.99      6876

    accuracy                           0.99     13680
   macro avg       0.99      0.99      0.99     13680
weighted avg       0.99      0.99      0.99     13680

0.9943713450292397
```

Figure 50 The recorded Confusion Matrix and Accuracy for the recombination

The recorded accuracy was 99.44%, with a false positive percentage of 0.50% and a false negative score of 0.62%. Comparing this to the Random Forest classifier utilizing Ajit Kumar et al. feature set, this is a 0.09% accuracy increase, a 0.12% false-positive percentage decrease, and a 0.06% false-negative percentage decrease.

5.3 Evaluation

Papers Tested Utilizing Our Dataset	Highest Reported Accuracy and Classifier used	Highest Achieved Accuracy and Classifier used
Jingling Zhao et al. [10]	98% SVM	82.38% SVM
Usukhbayar Baldangombo et al. [12]	90% DT	96.6% DT
Zane Markel and Michael Bilzor [13]	97.92% DT	98.65% DT
Ajit Kumar et al. [8]	97.43% RF	99.35% RF
Recombination of Zane Markel and Michael Bilzor [13] and Ajit Kumar et al. [8]	NA	**99.44% RF**

Table 4 Reported vs. Achieved Accuracy and its Classifier.

In order to evaluate the project, we will discuss the requirements proposed and to what degree they have been achieved.

ID	Type	Priority	Requirement
FR 1	Dataset	Must have	The dataset must have two separate sections of malware and benign files for testing purposes and all files must be validated to their respective category

The dataset constructed for this project includes an equal amount of confirmed benign and malicious files. The types of benign files are across the entire spectrum of Windows OS portable executable file types. They are guaranteed to be benign as they all were acquired from fresh installs of Windows OS. The malicious files were validated as being malicious and were then categorized by a custom script.

FR 2	Features	Must Have	Feature extraction and reduction must be executed on all files from the dataset used for testing purposes

During testing, all files in the dataset of 45,600 files were utilized for feature extraction. During feature extraction, reduction was introduced for direct inputting of the hexadecimal values out of the header.

FR 3	ML Technique	Must Have	Training of the various ML techniques must be performed on the training dataset

A total of six different machine learning classifiers were utilized in the testing of the various explored research papers.

FR 4	ML Technique	Must Have	The program must produce a confusion matrix stating the achieved accuracy, false positive, and false negative percentiles

The scripts developed for each paper's various classifiers all produced the confusion matrix, classification report, and achieved accuracy. The false positives and false negatives are present in the confusion matrix, and their percentages were calculated and stated in this report.

NFR 1	Dataset	Must Have	The dataset must contain at minimum 25,000 examples of recent malware designed for the Windows OS, in windows executable format

The dataset consists of 22,800 malicious and 22,800 benign files. The reason for the 22,800 limit is due to the desire to have a balanced dataset. As stated in section 3.1, over 27,000 suspected malicious files were collected. The collection of benign files from three versions of the Windows OS only resulted in 22,800.

NFR 2	ML Technique	Must Have	The program will be trained and run on a computer with at least quad-core CPU running above 2.0GHz

This requirement was in place to reduce the amount of computationally infeasible classifiers due to the size of the dataset. All testing was performed on a machine with an eight-core CPU running at 3.9 GHz. Even though this far exceeds the requirement, there was still one instance of computational infeasibility.

NFR 3	Evaluation	Must Have	Identification of a matrix for the evaluation of the research papers

The evaluation of all tested research papers was conducted from the output of the confusion matrix and accuracy scores displayed by the program. The primary evaluating factors were achieved accuracy, false positive, and false-negative rate.

A comparison between the achieved and reported results in table 4 shows a general trend of increased achieved accuracy to reported accuracy. This is most likely due to the non-bias dataset, as most of the tested papers were very malware biased in their datasets. Another possible explanation for the increase in accuracy could be due to the high percentage of Trojans in the malware dataset. It also highlights that the process utilized in this project was successful in acquiring the required features on a paper to paper basis. The achieved recombination result, although small in percentage, is still a positive sign that this set of features perform exceptionally well at training and testing classifiers. The increase in accuracy of 0.09% when tested with a dataset of our size would equate to roughly 41 additional files being correctly classified. If the dataset were to be scaled up to a million files, the increase in accuracy would equate to 900 files. Any scaled up solution would benefit from any additional accuracy percentages.

Chapter 6

Conclusion

This project has achieved the construction of a non-bias dataset through collecting malicious files from VirusShare [16] and benign from three separate Windows OS. The files were then tested to ensure they were of a readable, portable executable type. The project then successfully classified all malicious files utilizing reports from VirusTotal [17]. This led to the construction of a 45,600-file dataset that was then transformed into the JSON format. The necessary feature selection and CSV file creation were performed on the JSON files. Four research papers were then tested utilizing our extraction method and dataset to examine whether their approaches were practical. The recombination of two of the tested papers was performed and tested using the dataset to identify any performance changes. The dataset will be made publicly available but due to its sheer size of over 100gb this process is delayed. To receive access please contact the author of this report.

The significant challenges and limitations faced while undertaking this project were the possible inability of the python pefile module to extract the header from all types of portable executables. This issue came to light when not all the benign files could have their header read by the module. This could have limited the portable executable types in our dataset. Another possible limitation would be the word list classifier obtained from Naiyarah [18]. It is possible that the list is incomplete and therefore has led to the large number of files being classified as unknown. This leads directly to the next limitation, which is the Trojan rich malware dataset. Due to the overwhelming presence of probable trojan malware examples, the classifiers would have been able to overperform. This is due to the model created during training that would see mostly similar examples of malicious files during testing. The final challenge was concerning the efficiency of the parser.py script. The deepcopy function utilized could either be assisting efficiency or degrading it depending on the size of

the input file. Additional testing and experimenting must be carried out to determine its overall impact on the script's runtime efficiency.

The proposed future work for this project would be to increase the dataset size and improve the variety of malicious portable executable files. Although Trojans are the most common malware type, the increase in variety will result in a more real-world testing environment. Additionally, the usage of k-cross fold verification can be implemented when utilizing the classifiers. This will result in the produced results becoming averages over various runs of the classifier on different folds of the dataset. The project scope can also be expanded to include n-grams of the raw files, string counts, or API calls. This would, however, require changing the extraction method as these features are not located in the portable executable header. Our proposed approach achieved a higher accuracy then the four implemented papers. Future experimentation utilizing this approach is essential to determine its large-scale viability.

Chapter 7

Bibliography

[1] Kaspersky, "Digital Dangerscape: Kaspersky Lab Spotlights Cybersecurity Trends in the Middle East, Turkey and Africa," 30 April 2019. [Online]. Available: https://me-en.kaspersky.com/about/press-releases/2019_digital-dangerscape-kaspersky-lab-spotlights-cybersecurity-trends-in-the-middle-east-turkey-and-africa. [Accessed November 2019].

[2] L. Liu, W. Bao-sheng, Y. Bo and Z. Qiu-xi, "Automatic malware classification and new malware detection," Frontiers of Information Technology & Electronic Engineering, Changsha, 2017.

[3] G. R¨atsch, "A Brief Introduction into Machine Learning," Friedrich Miescher Laboratory of the Max Planck Society, T¨ubingen, 2004.

[4] S. Joshua and S. Hillary, "Chaptor 6: Understanding Machine Learning-Based Malware Detectors," in *Malware Data Science*, San Francisco, no starch press, 2018, pp. 89-117.

[5] K. Kris and M. Chad, "Practical Malware Analysis," Black Hat, 2014.

[6] S. Michael and H. Andrew, Practical Malware Analysis, San Fransisco: No Starch Press, 2012.

[7] S. S. Shai and B. D. Shai, Understanding Machine Learning:, New York: Cambridge University Press, 2014.

[8] K. Ajit, K. K.S. and A. G., "A learning model to detect maliciousness of portable executable using," ScienceDirect, Riyadh, 2017.

[9] K. S. Akash and J. Aruna, "Integrated Malware Analysis Using Machine Learning," in *International Conference on Telecommunication and Networks*, Noida, 2017.

[10] Z. Jingling, Z. Suoxing, L. Gohan and B. Cui, "Malware Detection Using Machine Learning Based," in *10.1109/ICCCN.2018.8487459*, 2018.

[11] I. Muhammad, H. D. Muhammad and I. Maliha, "Static and Dynamic Malware Analysis," in *IBCAST*, Islamabad, 2019.

[12] B. Usukhbayar, J. Nyamjav and H. Shi-Jinn, "A Static Malware Detection System Using Data Mining Methods," National University of Mongolia, 2013.

[13] Z. Markel and M. Bilzor, "Machine Learning Based Malware Detection," Defense Technical Information Center, Annapolis, Maryland, 2015.

[14] S. Hrushikesh, P. Sonali, S. Dewang, S. Lucky, S. Mayank and T. Kumar, "On the Design of Supervised Binary Classifiers for Malware Detection using Portable Executable Files," in *2019 IEEE 9th International Conference on Advanced Computing (IACC)*, Tiruchirappalli, India, 2019.

[15] B. Mohamed, G. Bouchra, B. Yasmine, D. Abdelouahid and B. Mahmoud, "Malware Detection System Based on an In-Depth Analysis of the Portable Executable Headers," in *International Conference on Machine Learning for Networking*, Paris, France, 2019.

[16] "VirusShare.com," [Online]. Available: https://virusshare.com/. [Accessed 22 November 2019].

[17] "Virus Total," [Online]. Available: https://www.virustotal.com/gui/home. [Accessed 22 November 2019].

[18] H. Naiyarah, "Malware Analysis and Detection Using Machine Learning Classifiers," Heriot-Watt University, Dubai, 2016.

[19] JSON, "Introducing JSON," [Online]. Available: https://www.json.org. [Accessed 22 04 2020].

[20] BPL, "Stackoverflow," 20 Febuary 2018. [Online]. Available: https://stackoverflow.com/questions/24756712/deepcopy-is-extremely-slow.

[21] "Dependency Walker 2.2," [Online]. Available: http://www.dependencywalker.com/. [Accessed 22 November 2019].

[22] "ODA," [Online]. Available: https://onlinedisassembler.com/odaweb/. [Accessed 22 November 2019].

[23] "Scrum.org," [Online]. Available: https://www.scrum.org/. [Accessed 22 November 2019].

[24] A. Saba and A. S. Nazar, "Detection of Malicious Executables Using Static and Dynamic Features of Portable Executable (PE) File," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, Zhangjiajie, China, 2016.

Chapter 8

Appendix

The following are the primary scripts utilized in this project. The scripts to perform the feature extraction, CSV file creation, and execute the machine learning classifier are set up to perform Zane Markel and Michael Bilzor [13] Decision Tree approach.

PEcheck.py

```
import os
import pefile
import shutil

directory = '//Volumes/MyBook/Benign/Windows_8'

newpath = '//Users/cedricecran/Desktop/Windows_8'

for entry in os.scandir(directory):
   try:
      pe = pefile.PE(entry, fast_load=False)

   except pefile.PEFormatError:
      continue

   shutil.copy(entry, newpath)

for file in os.listdir(newpath):
   filehead = os.path.join(newpath, file)
   newfilehead = os.path.join(newpath, file + "_windows8")
   os.rename(filehead, newfilehead)
```

VTScan.py

```
import requests
import os

url = 'https://www.virustotal.com/vtapi/v2/file/scan'

params = {'apikey':
'819e0686e3a8ac0ca16eca7e523111fb326fd2d318749fd787c2ee5b00dd2ccc'}

rootdir = '//Volumes/MyBook/WARNING_VIRUSES_KEEP_OUT/PEMalware'
```

```
for filename in os.listdir(rootdir):
	if filename.startswith('Virus'):
		filehead = os.path.join(rootdir, filename)
		files = {'file': (filehead, open(filehead, 'rb'))}
		response = requests.post(url, files=files, params=params)
		dictresponse = response.json()
		newname = os.path.join(rootdir, dictresponse.get('md5'))
		os.rename(filehead, newname)
		print(response.json())
		print (filename)
		continue
```

VTReport.py

```
import requests
import os
import json

url = 'https://www.virustotal.com/vtapi/v2/file/report'

rootdir = '//Volumes/MyBook/WARNING_VIRUSES_KEEP_OUT/PEMalware'


for filename in os.listdir(rootdir):
	if not filename.startswith('VT'):
		params = {'apikey':
'819e0686e3a8ac0ca16eca7e523111fb326fd2d318749fd787c2ee5b00dd2ccc',
'resource': filename}
		response = requests.get(url, params=params)
		writefile = open(filename + ".txt", "w")
		writefile.write(json.dumps(response.json()))
		writefile.close()
```

PEextract.py

```
import os
import pefile

directory = '//Users/cedricecran/Desktop/Windows_8'


for filename in os.listdir(directory):
	if not filename.endswith('.py'):
		filehead = os.path.join(directory, filename)
		pe = pefile.PE(filehead, fast_load=False)
		info = pe.dump_info()
		try:
			with open(filename + ".txt", 'w') as textfile:
```

```
                        textfile.write(info)
            except IOError:
                print("I/O error")
                print(filename)
```

Malwareclassifier.py

```
import os
import shutil
import re

#NAYARA's WORD LIST ------
#trojan = " troj_gen trojan trj "
#exploit = " exploit "
#worm = " worm "
#backdoor =" backdoor "
#virus = " zherkov rehenes badda funlove vampiro kunkka trafaret troxa bleah slugin
tolone hala bolzano relnek hidrag huhk hublo xpaj aliser polip madangel ipamor alman
jadtre neshta iparmor otwycal ramnit parite elkern expiro virut virtob virlock
virransom sality "
#adware = " adware "
#generic = " gen generic "
#potentiallyUnwanted = " pup pua unwanted solimba installcore "
#riskware = " riskware "
#rootkit = " rootkit "
#--------------------

rootdir = '//Users/cedricecran/Desktop/Classification'

countdict = {}

for filename in os.listdir(rootdir):
    if filename.endswith('.txt'):
        filehead = os.path.join(rootdir, filename)
        with open(filehead) as f:
            trojanreg = re.findall('troj_gen|trojan|trj', f.read(),
flags=re.IGNORECASE)
            exploitreg = re.findall('exploit', f.read(), flags=re.IGNORECASE)
            wormreg = re.findall('worm', f.read(), flags=re.IGNORECASE)
            backdoorreg = re.findall('backdoor', f.read(),
flags=re.IGNORECASE)
            virusreg =
re.findall('zherkov|rehenes|badda|funlove|vampiro|kunkka|trafaret|troxa|bleah|slugin|
tolone|hala|bolzano|relnek|hidrag|huhk|hublo|xpaj|aliser|polip|madangel|ipamor|alma
n|jadtre|neshta|iparmor|otwycal|ramnit|parite|elkern|expiro|virut|virtob|virlock|virra
nsom|sality|virus', f.read(), flags=re.IGNORECASE)
            adwarereg = re.findall('adware', f.read(), flags=re.IGNORECASE)
            genericreg = re.findall('gen|generic', f.read(),
flags=re.IGNORECASE)
```

```
                potentiallyUnwantedreg =
re.findall('pup|pua|unwanted|solimba|installcore', f.read(), flags=re.IGNORECASE)
                riskwarereg = re.findall('riskware', f.read(), flags=re.IGNORECASE)
                rootkitreg = re.findall('rootkit', f.read(), flags=re.IGNORECASE)

                trojancount = len(trojanreg)
                exploitcount = len(exploitreg)
                wormcount = len(wormreg)
                backdoorcount = len(backdoorreg)
                viruscount = len(virusreg)
                adwarecount = len(adwarereg)
                genericcount = len(genericreg)
                potentiallyUnwantedcount = len(potentiallyUnwantedreg)
                riskwarecount = len(riskwarereg)
                rootkitcount = len(rootkitreg)

                Adwaredir = '//Users/cedricecran/Desktop/Adware2/' + filename
                Backdoordir = '//Users/cedricecran/Desktop/Backdoor2/' +
filename
                Exploitdir = '//Users/cedricecran/Desktop/Exploit2/' + filename
                Genericdir = '//Users/cedricecran/Desktop/Generic2/' + filename
                Potendir =
'//Users/cedricecran/Desktop/PotentiallyUnwanted2/' + filename
                Riskwaredir = '//Users/cedricecran/Desktop/Riskware2' +
filename
                Rootkitdir = '//Users/cedricecran/Desktop/Rootkit2/' + filename
                Trojandir = '//Users/cedricecran/Desktop/Trojan2/' + filename
                Virusdir = '//Users/cedricecran/Desktop/Virus2/' + filename
                Wormdir = '//Users/cedricecran/Desktop/Worm2/' + filename
                Elsedir = '//Users/cedricecran/Desktop/Unclassified2/' +
filename


                countdict = {'unknown' : 0, 'trojan': trojancount, 'adware':
adwarecount, 'backdoor': backdoorcount, 'exploit': exploitcount, 'generic': genericcount,
'potendir': potentiallyUnwantedcount, 'riskware': riskwarecount, 'rootkit': rootkitcount,
'worm': wormcount, 'virus': viruscount}


                mosttype = max(countdict, key=countdict.get)

                if mosttype == 'trojan':
                        shutil.move(filehead, Trojandir)
                elif mosttype == 'adware':
                        shutil.move(filehead, Adwaredir)
                elif mosttype == 'backdoor':
                        shutil.move(filehead, Backdoordir)
                elif mosttype =='exploit':
                        shutil.move(filehead, Exploitdir)
```

```
                elif mosttype == 'generic':
                        shutil.move(filehead, Genericdir)
                elif mosttype == 'potendir':
                        shutil.move(filehead, Potendir)
                elif mosttype == 'riskware':
                        shutil.move(filehead, Riskwaredir)
                elif mosttype == 'rootkit':
                        shutil.move(filehead, Rootkitdir)
                elif mosttype == 'worm':
                        shutil.move(filehead, Wormdir)
                elif mosttype == 'virus':
                        shutil.move(filehead, Virusdir)
                elif mosttype == 'unknown':
                        shutil.move(filehead, Elsedir)

        countdict.clear()
```

Parsertest.py

```
import json
import os
import re
import copy

# the file to be converted
directory = '//Users/cedricecran/Desktop/MalwareDataset'

# intermediate and resultant dictionaries
# file
dictfinal = {}

# sections
dictsect = {}

# headers
dictheader = {}

# variable
dictdos = {}

dictnt = {}

dictfile = {}

dictopt = {}

dictpe = {}

dictdiexport = {}
```

```
dictdiimport = {}

dictdires = {}

dictdiexcep = {}

dictdisec = {}

dictdibase = {}

dictdidebug = {}

dictdicopy = {}

dictdiglob = {}

dictditls = {}

dictdiload = {}

dictdibound = {}

dictdiiat = {}

dictdidelay = {}

dictdicom = {}

dictres = {}

dictimp = {}

dicttls = {}

dictreloc = {}

dictdebug = {}

dictload = {}

Listdll = []

Listhighlow = []

Listabsolute = []

Listsect = [0] * 50000
```

```
Listimp = []

Listfiles = []

List = []

i = 0

f = 0

j = 0

dll = 0

e = 0

final = 0

basereloc = 0

typedebug = 0


class _RegExLib:
   #Creating the Regular Expressions
   #----------------------------------------------------------
   _reg_image_dos_header = re.compile('IMAGE_DOS_HEADER')
   _reg_e_magic = re.compile('e_magic:\s*([a-z0-9A-Z]*)')
   _reg_e_cblp = re.compile('e_cblp:\s*([a-z0-9A-Z]*)')
   _reg_e_cp = re.compile('e_cp:\s*([a-z0-9A-Z]*)')
   _reg_e_crlc = re.compile('e_crlc:\s*([a-z0-9A-Z]*)')
   _reg_e_cparhdr = re.compile('e_cparhdr:\s*([a-z0-9A-Z]*)')
   _reg_e_minalloc = re.compile('e_minalloc: \s*([a-z0-9A-Z]*)')
   _reg_e_maxalloc = re.compile('e_maxalloc:\s*([a-z0-9A-Z]*)')
   _reg_e_ss = re.compile('e_ss:\s*([a-z0-9A-Z]*)')
   _reg_e_sp = re.compile('e_sp:\s*([a-z0-9A-Z]*)')
   _reg_e_csum = re.compile('e_csum:\s*([a-z0-9A-Z]*)')
   _reg_e_ip = re.compile('e_ip:\s*([a-z0-9A-Z]*)')
   _reg_e_cs = re.compile('e_cs:\s*([a-z0-9A-Z]*)')
   _reg_e_lfarlc = re.compile('e_lfarlc:\s*([a-z0-9A-Z]*)')
   _reg_e_ovno = re.compile('e_ovno:\s*([a-z0-9A-Z]*)')
   _reg_e_oemid = re.compile('e_oemid:\s*([a-z0-9A-Z]*)')
   _reg_e_oeminfo = re.compile('e_oeminfo:\s*([a-z0-9A-Z]*)')
   _reg_e_lfanew = re.compile('e_lfanew:\s*([a-z0-9A-Z]*)')
   # ----------------------------------------------------------
   _reg_image_nt_headers = re.compile('IMAGE_NT_HEADERS')
   _reg_Signature = re.compile('Signature:\s*([a-z0-9A-Z]*)')
   # ----------------------------------------------------------
   _reg_image_file_header = re.compile('IMAGE_FILE_HEADER')
```

```
    _reg_Machine = re.compile('Machine:\s*([a-z0-9A-Z]*)')
    _reg_NumberOfSections = re.compile('NumberOfSections:\s*([a-z0-9A-Z]*)')
    _reg_TimeDateStamp = re.compile('TimeDateStamp:\s*([a-z0-9A-Z]*)')
    _reg_PointerToSymbolTable = re.compile('PointerToSymbolTable:\s*([a-z0-9A-Z]*)')
    _reg_NumberOfSymbols = re.compile('NumberOfSymbols:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfOptionalHeader = re.compile('SizeOfOptionalHeader:\s*([a-z0-9A-Z]*)')
    _reg_Characteristics = re.compile('Characteristics:\s*([a-z0-9A-Z]*)')
    _reg_Flags = re.compile('Flags:')
    _reg_InnerFlags = re.compile('[^Flags: ][^,\s][^\,]*[^,\s]*[^\n]')
    # -----------------------------------------------------
    _reg_optional_header = re.compile('IMAGE_OPTIONAL_HEADER')
    _reg_Magic = re.compile('Magic:\s*([a-z0-9A-Z]*)')
    _reg_MajorLinkerVersion = re.compile('MajorLinkerVersion:\s*([a-z0-9A-Z]*)')
    _reg_MinorLinkerVersion = re.compile('MinorLinkerVersion:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfCode = re.compile('SizeOfCode:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfInitializedData = re.compile('SizeOfInitializedData:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfUninitializedData = re.compile('SizeOfUninitializedData:\s*([a-z0-9A-
Z]*)')
    _reg_AddressOfEntryPoint = re.compile('AddressOfEntryPoint:\s*([a-z0-9A-Z]*)')
    _reg_BaseOfCode = re.compile('BaseOfCode:\s*([a-z0-9A-Z]*)')
    _reg_BaseOfData = re.compile('BaseOfData:\s*([a-z0-9A-Z]*)')
    _reg_ImageBase = re.compile('ImageBase:\s*([a-z0-9A-Z]*)')
    _reg_SectionAlignment = re.compile('SectionAlignment:\s*([a-z0-9A-Z]*)')
    _reg_FileAlignment = re.compile('FileAlignment:\s*([a-z0-9A-Z]*)')
    _reg_MajorOperatingSystemVersion =
re.compile('MajorOperatingSystemVersion:\s*([a-z0-9A-Z]*)')
    _reg_MinorOperatingSystemVersion =
re.compile('MinorOperatingSystemVersion:\s*([a-z0-9A-Z]*)')
    _reg_MajorImageVersion = re.compile('MajorImageVersion:\s*([a-z0-9A-Z]*)')
    _reg_MinorImageVersion = re.compile('MinorImageVersion:\s*([a-z0-9A-Z]*)')
    _reg_MajorSubsystemVersion = re.compile('MajorSubsystemVersion:\s*([a-z0-9A-
Z]*)')
    _reg_MinorSubsystemVersion = re.compile('MinorSubsystemVersion:\s*([a-z0-9A-
Z]*)')
    _reg_Reserved1 = re.compile('Reserved1:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfImage = re.compile('SizeOfImage:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfHeaders = re.compile('SizeOfHeaders:\s*([a-z0-9A-Z]*)')
    _reg_CheckSum = re.compile('CheckSum:\s*([a-z0-9A-Z]*)')
    _reg_Subsystem = re.compile('Subsystem:\s*([a-z0-9A-Z]*)')
    _reg_DllCharacteristics = re.compile('DllCharacteristics:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfStackReserve = re.compile('SizeOfStackReserve:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfStackCommit = re.compile('SizeOfStackCommit:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfHeapReserve = re.compile('SizeOfHeapReserve:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfHeapCommit = re.compile('SizeOfHeapCommit:\s*([a-z0-9A-Z]*)')
    _reg_LoaderFlags = re.compile('LoaderFlags:\s*([a-z0-9A-Z]*)')
    _reg_NumberOfRvaAndSizes = re.compile('NumberOfRvaAndSizes:\s*([a-z0-9A-Z]*)')
    # -----------------------------------------------------
    _reg_image_section_header = re.compile('IMAGE_SECTION_HEADER')
    _reg_Name = re.compile('Name:\s*([.a-z0-9A-Z]*)')
```

```
    _reg_Misc = re.compile('Misc:\s*([a-z0-9A-Z]*)')
    _reg_Misc_PhysicalAddress = re.compile('Misc_PhysicalAddress:\s*([a-z0-9A-Z]*)')
    _reg_Misc_VirtualSize = re.compile('Misc_VirtualSize:\s*([a-z0-9A-Z]*)')
    _reg_VirtualAddress = re.compile('VirtualAddress:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfRawData = re.compile('SizeOfRawData:\s*([a-z0-9A-Z]*)')
    _reg_PointerToRawData = re.compile('PointerToRawData:\s*([a-z0-9A-Z]*)')
    _reg_PointerToRelocations = re.compile('PointerToRelocations:\s*([a-z0-9A-Z]*)')
    _reg_PointerToLinenumbers = re.compile('PointerToLinenumbers:\s*([a-z0-9A-Z]*)')
    _reg_NumberOfRelocations = re.compile('NumberOfRelocations:\s*([a-z0-9A-Z]*)')
    _reg_NumberOfLinenumbers = re.compile('NumberOfLinenumbers:\s*([a-z0-9A-
Z]*)')
    _reg_Entropy = re.compile('Entropy:\s*([a-z0-9A-Z.]*)')
    _reg_MD5 = re.compile('MD5 {5}hash:\s*([a-z0-9A-Z]*)')
    _reg_SHA1 = re.compile('SHA-1 {3}hash:\s*([a-z0-9A-Z]*)')
    _reg_SHA256 = re.compile('SHA-256 {1}hash:\s*([a-z0-9A-Z]*)')
    _reg_SHA512 = re.compile('SHA-512 {1}hash:\s*([a-z0-9A-Z]*)')
    # ------------------------------------------------------
    _reg_image_directory_entry_export =
re.compile('IMAGE_DIRECTORY_ENTRY_EXPORT')
    _reg_image_directory_entry_import =
re.compile('IMAGE_DIRECTORY_ENTRY_IMPORT')
    _reg_image_directory_entry_resource =
re.compile('IMAGE_DIRECTORY_ENTRY_RESOURCE')
    _reg_image_directory_entry_exception =
re.compile('IMAGE_DIRECTORY_ENTRY_EXCEPTION')
    _reg_image_directory_entry_security =
re.compile('IMAGE_DIRECTORY_ENTRY_SECURITY')
    _reg_image_directory_entry_basereloc =
re.compile('IMAGE_DIRECTORY_ENTRY_BASERELOC')
    _reg_image_directory_entry_debug =
re.compile('IMAGE_DIRECTORY_ENTRY_DEBUG')
    _reg_image_directory_entry_copyright =
re.compile('IMAGE_DIRECTORY_ENTRY_COPYRIGHT')
    _reg_image_directory_entry_globalptr =
re.compile('IMAGE_DIRECTORY_ENTRY_GLOBALPTR')
    _reg_image_directory_entry_tls = re.compile('IMAGE_DIRECTORY_ENTRY_TLS')
    _reg_image_directory_entry_load_config =
re.compile('IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG')
    _reg_image_directory_entry_bound_import =
re.compile('IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT')
    _reg_image_directory_entry_iat = re.compile('IMAGE_DIRECTORY_ENTRY_IAT')
    _reg_image_directory_entry_delay_import =
re.compile('IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT')
    _reg_image_directory_entry_com_descriptor =
re.compile('IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR')
    _reg_image_directory_entry_reserved =
re.compile('IMAGE_DIRECTORY_ENTRY_RESERVED')
    _reg_Size = re.compile('Size:\s*([a-z0-9A-Z]*)')
    # ------------------------------------------------------
```

```
    _reg_image_import_descriptor = re.compile('IMAGE_IMPORT_DESCRIPTOR')
    _reg_OriginalFirstThunk = re.compile('OriginalFirstThunk:\s*([a-z0-9A-Z]*)')
    _reg_ForwarderChain = re.compile('ForwarderChain:\s*([a-z0-9A-Z]*)')
    _reg_FirstThunk = re.compile(' FirstThunk:\s*([a-z0-9A-Z]*)')
    _reg_DLL = re.compile('([a-z0-9A-Z]*32).dll.([a-z0-9A-Z]*)')
    # --------------------------------------------------
    _reg_image_resource_directory = re.compile('IMAGE_RESOURCE_DIRECTORY')
    # --------------------------------------------------
    _reg_image_tls_directory = re.compile('IMAGE_TLS_DIRECTORY')
    _reg_StartAddressOfRawData = re.compile('StartAddressOfRawData:\s*([a-z0-9A-
Z]*)')
    _reg_EndAddressOfRawData = re.compile('EndAddressOfRawData:\s*([a-z0-9A-Z]*)')
    _reg_AddressOfIndex = re.compile('AddressOfIndex:\s*([a-z0-9A-Z]*)')
    _reg_AddressOfCallBacks = re.compile('AddressOfCallBacks:\s*([a-z0-9A-Z]*)')
    _reg_SizeOfZeroFill = re.compile('SizeOfZeroFill:\s*([a-z0-9A-Z]*)')
    # --------------------------------------------------
    _reg_image_base_relocation = re.compile('IMAGE_BASE_RELOCATION')
    _reg_highlow = re.compile('([.a-z0-9A-Z]*) HIGHLOW')
    _reg_absolute = re.compile('([.a-z0-9A-Z]*) ABSOLUTE')
    # --------------------------------------------------
    _reg_image_load_config_directory = re.compile('IMAGE_LOAD_CONFIG_DIRECTORY')
    _reg_MajorVersion = re.compile('MajorVersion:\s*([a-z0-9A-Z]*)')
    _reg_MinorVersion = re.compile('MinorVersion:\s*([a-z0-9A-Z]*)')
    _reg_GlobalFlagsClear = re.compile('GlobalFlagsClear:\s*([a-z0-9A-Z]*)')
    _reg_GlobalFlagsSet = re.compile('GlobalFlagsSet:\s*([a-z0-9A-Z]*)')
    _reg_CriticalSectionDefaultTimeout =
re.compile('CriticalSectionDefaultTimeout:\s*([a-z0-9A-Z]*)')
    _reg_DeCommitFreeBlockThreshold =
re.compile('DeCommitFreeBlockThreshold:\s*([a-z0-9A-Z]*)')
    _reg_DeCommitTotalFreeThreshold =
re.compile('DeCommitTotalFreeThreshold:\s*([a-z0-9A-Z]*)')
    _reg_LockPrefixTable = re.compile('LockPrefixTable:\s*([a-z0-9A-Z]*)')
    _reg_MaximumAllocationSize = re.compile('MaximumAllocationSize:\s*([a-z0-9A-
Z]*)')
    _reg_VirtualMemoryThreshold = re.compile('VirtualMemoryThreshold:\s*([a-z0-9A-
Z]*)')
    _reg_ProcessHeapFlags = re.compile('ProcessHeapFlags:\s*([a-z0-9A-Z]*)')
    _reg_ProcessAffinityMask = re.compile('ProcessAffinityMask:\s*([a-z0-9A-Z]*)')
    _reg_CSDVersion = re.compile('CSDVersion:\s*([a-z0-9A-Z]*)')
    _reg_EditList = re.compile('EditList:\s*([a-z0-9A-Z]*)')
    _reg_SecurityCookie = re.compile('SecurityCookie:\s*([a-z0-9A-Z]*)')
    _reg_SEHandlerTable = re.compile('SEHandlerTable:\s*([a-z0-9A-Z]*)')
    _reg_SEHandlerCount = re.compile('SEHandlerCount:\s*([a-z0-9A-Z]*)')
    _reg_GuardCFCheckFunctionPointer =
re.compile('GuardCFCheckFunctionPointer:\s*([a-z0-9A-Z]*)')
    _reg_Reserved2 = re.compile('Reserved2:\s*([a-z0-9A-Z]*)')
    _reg_GuardCFFunctionTable = re.compile('GuardCFFunctionTable:\s*([a-z0-9A-Z]*)')
    _reg_GuardCFFunctionCount = re.compile('GuardCFFunctionCount:\s*([a-z0-9A-Z]*)')
    _reg_GuardFlags = re.compile('GuardFlags:\s*([a-z0-9A-Z]*)')
```

```
# --------------------------------------------------------
_reg_image_debug_directory = re.compile('IMAGE_DEBUG_DIRECTORY')
_reg_Type = re.compile('Type:\s*([a-z0-9A-Z]*)')
_reg_SizeOfData = re.compile('SizeOfData:\s*([a-z0-9A-Z]*)')
_reg_AddressOfRawData = re.compile('AddressOfRawData:\s*([a-z0-9A-Z]*)')
_reg_Typelist = re.compile('Type: [A-Za-z0-9]')
_reg_InnerType = re.compile('[^Type: ][^,\s][^\,]*[^,\s]*[^\n]')

def __init__(self, line):
   # check whether line has a positive match with all of the regular expressions
   self.image_dos_header = self._reg_image_dos_header.search(line)
   self.e_magic = self._reg_e_magic.search(line)
   self.e_cblp = self._reg_e_cblp.search(line)
   self.e_cp = self._reg_e_cp.search(line)
   self.e_crlc = self._reg_e_crlc.search(line)
   self.e_cparhdr = self._reg_e_cparhdr.search(line)
   self.e_minalloc = self._reg_e_minalloc.search(line)
   self.e_maxalloc = self._reg_e_maxalloc.search(line)
   self.e_ss = self._reg_e_ss.search(line)
   self.e_sp = self._reg_e_sp.search(line)
   self.e_csum = self._reg_e_csum.search(line)
   self.e_ip = self._reg_e_ip.search(line)
   self.e_cs = self._reg_e_cs.search(line)
   self.e_lfarlc = self._reg_e_lfarlc.search(line)
   self.e_ovno = self._reg_e_ovno.search(line)
   self.e_oemid = self._reg_e_oemid.search(line)
   self.e_oeminfo = self._reg_e_oeminfo.search(line)
   self.e_lfanew = self._reg_e_lfanew.search(line)
   self.image_nt_headers = self._reg_image_nt_headers.search(line)
   self.Signature = self._reg_Signature.search(line)
   self.Machine = self._reg_Machine.search(line)
   self.image_file_header = self._reg_image_file_header.search(line)
   self.NumberOfSections = self._reg_NumberOfSections.search(line)
   self.TimeDateStamp = self._reg_TimeDateStamp.search(line)
   self.PointerToSymbolTable = self._reg_PointerToSymbolTable.search(line)
   self.NumberOfSymbols = self._reg_NumberOfSymbols.search(line)
   self.SizeOfOptionalHeader = self._reg_SizeOfOptionalHeader.search(line)
   self.Characteristics = self._reg_Characteristics.search(line)
   self.Flags = self._reg_Flags.search(line)
   self.InnerFlags = self._reg_InnerFlags.findall(line)
   self.optional_header = self._reg_optional_header.search(line)
   self.Magic = self._reg_Magic.search(line)
   self.MajorLinkerVersion = self._reg_MajorLinkerVersion.search(line)
   self.MinorLinkerVersion = self._reg_MinorLinkerVersion.search(line)
   self.SizeOfCode = self._reg_SizeOfCode.search(line)
   self.SizeOfInitializedData = self._reg_SizeOfInitializedData.search(line)
   self.SizeOfUninitializedData = self._reg_SizeOfUninitializedData.search(line)
   self.AddressOfEntryPoint = self._reg_AddressOfEntryPoint.search(line)
   self.BaseOfCode = self._reg_BaseOfCode.search(line)
```

```
        self.BaseOfData = self._reg_BaseOfData.search(line)
        self.ImageBase = self._reg_ImageBase.search(line)
        self.SectionAlignment = self._reg_SectionAlignment.search(line)
        self.FileAlignment = self._reg_FileAlignment.search(line)
        self.MajorOperatingSystemVersion =
self._reg_MajorOperatingSystemVersion.search(line)
        self.MinorOperatingSystemVersion =
self._reg_MinorOperatingSystemVersion.search(line)
        self.MajorImageVersion = self._reg_MajorImageVersion.search(line)
        self.MinorImageVersion = self._reg_MinorImageVersion.search(line)
        self.MajorSubsystemVersion = self._reg_MajorSubsystemVersion.search(line)
        self.MinorSubsystemVersion = self._reg_MinorSubsystemVersion.search(line)
        self.Reserved1 = self._reg_Reserved1.search(line)
        self.SizeOfImage = self._reg_SizeOfImage.search(line)
        self.SizeOfHeaders = self._reg_SizeOfHeaders.search(line)
        self.CheckSum = self._reg_CheckSum.search(line)
        self.Subsystem = self._reg_Subsystem.search(line)
        self.DllCharacteristics = self._reg_DllCharacteristics.search(line)
        self.SizeOfStackReserve = self._reg_SizeOfStackReserve.search(line)
        self.SizeOfStackCommit = self._reg_SizeOfStackCommit.search(line)
        self.SizeOfHeapReserve = self._reg_SizeOfHeapReserve.search(line)
        self.SizeOfHeapCommit = self._reg_SizeOfHeapCommit.search(line)
        self.LoaderFlags = self._reg_LoaderFlags.search(line)
        self.NumberOfRvaAndSizes = self._reg_NumberOfRvaAndSizes.search(line)
        self.image_section_header = self._reg_image_section_header.search(line)
        self.Name = self._reg_Name.search(line)
        self.Misc = self._reg_Misc.search(line)
        self.Misc_PhysicalAddress = self._reg_Misc_PhysicalAddress.search(line)
        self.Misc_VirtualSize = self._reg_Misc_VirtualSize.search(line)
        self.VirtualAddress = self._reg_VirtualAddress.search(line)
        self.SizeOfRawData = self._reg_SizeOfRawData.search(line)
        self.PointerToRawData = self._reg_PointerToRawData.search(line)
        self.PointerToRelocations = self._reg_PointerToRelocations.search(line)
        self.PointerToLinenumbers = self._reg_PointerToLinenumbers.search(line)
        self.NumberOfRelocations = self._reg_NumberOfRelocations.search(line)
        self.NumberOfLinenumbers = self._reg_NumberOfLinenumbers.search(line)
        self.Entropy = self._reg_Entropy.search(line)
        self.MD5 = self._reg_MD5.search(line)
        self.SHA1 = self._reg_SHA1.search(line)
        self.SHA256 = self._reg_SHA256.search(line)
        self.SHA512 = self._reg_SHA512.search(line)
        self.image_directory_entry_export =
self._reg_image_directory_entry_export.search(line)
        self.image_directory_entry_import =
self._reg_image_directory_entry_import.search(line)
        self.image_directory_entry_resource =
self._reg_image_directory_entry_resource.search(line)
        self.image_directory_entry_exception =
self._reg_image_directory_entry_exception.search(line)
```

```
        self.image_directory_entry_security =
self._reg_image_directory_entry_security.search(line)
        self.image_directory_entry_basereloc =
self._reg_image_directory_entry_basereloc.search(line)
        self.image_directory_entry_debug =
self._reg_image_directory_entry_debug.search(line)
        self.image_directory_entry_copyright =
self._reg_image_directory_entry_copyright.search(line)
        self.image_directory_entry_globalptr =
self._reg_image_directory_entry_globalptr.search(line)
        self.image_directory_entry_tls = self._reg_image_directory_entry_tls.search(line)
        self.image_directory_entry_load_config =
self._reg_image_directory_entry_load_config.search(line)
        self.image_directory_entry_bound_import =
self._reg_image_directory_entry_bound_import.search(line)
        self.image_directory_entry_iat = self._reg_image_directory_entry_iat.search(line)
        self.image_directory_entry_delay_import =
self._reg_image_directory_entry_delay_import.search(line)
        self.image_directory_entry_com_descriptor =
self._reg_image_directory_entry_com_descriptor.search(line)
        self.image_directory_entry_reserved =
self._reg_image_directory_entry_reserved.search(line)
        self.Size = self._reg_Size.search(line)
        self.image_import_descriptor = self._reg_image_import_descriptor.search(line)
        self.OriginalFirstThunk = self._reg_OriginalFirstThunk.search(line)
        self.ForwarderChain = self._reg_ForwarderChain.search(line)
        self.FirstThunk = self._reg_FirstThunk.search(line)
        self.DLL = self._reg_DLL.search(line)
        self.image_resource_directory = self._reg_image_resource_directory.search(line)
        self.image_load_config_directory =
self._reg_image_load_config_directory.search(line)
        self.MajorVersion = self._reg_MajorVersion.search(line)
        self.MinorVersion = self._reg_MinorVersion.search(line)
        self.GlobalFlagsClear = self._reg_GlobalFlagsClear.search(line)
        self.GlobalFlagsSet = self._reg_GlobalFlagsSet.search(line)
        self.CriticalSectionDefaultTimeout =
self._reg_CriticalSectionDefaultTimeout.search(line)
        self.DeCommitFreeBlockThreshold =
self._reg_DeCommitFreeBlockThreshold.search(line)
        self.DeCommitTotalFreeThreshold =
self._reg_DeCommitTotalFreeThreshold.search(line)
        self.LockPrefixTable = self._reg_LockPrefixTable.search(line)
        self.MaximumAllocationSize = self._reg_MaximumAllocationSize.search(line)
        self.VirtualMemoryThreshold = self._reg_VirtualMemoryThreshold.search(line)
        self.ProcessHeapFlags = self._reg_ProcessHeapFlags.search(line)
        self.ProcessAffinityMask = self._reg_ProcessAffinityMask.search(line)
        self.CSDVersion = self._reg_CSDVersion.search(line)
        self.EditList = self._reg_EditList.search(line)
        self.SecurityCookie = self._reg_SecurityCookie.search(line)
```

```
        self.SEHandlerTable = self._reg_SEHandlerTable.search(line)
        self.SEHandlerCount = self._reg_SEHandlerCount.search(line)
        self.GuardCFCheckFunctionPointer =
self._reg_GuardCFCheckFunctionPointer.search(line)
        self.Reserved2 = self._reg_Reserved2.search(line)
        self.GuardCFFunctionTable = self._reg_GuardCFFunctionTable.search(line)
        self.GuardCFFunctionCount = self._reg_GuardCFFunctionCount.search(line)
        self.GuardFlags = self._reg_GuardFlags.search(line)
        self.image_tls_directory = self._reg_image_tls_directory.search(line)
        self.StartAddressOfRawData = self._reg_StartAddressOfRawData.search(line)
        self.EndAddressOfRawData = self._reg_EndAddressOfRawData.search(line)
        self.AddressOfIndex = self._reg_AddressOfIndex.search(line)
        self.AddressOfCallBacks = self._reg_AddressOfCallBacks.search(line)
        self.SizeOfZeroFill = self._reg_SizeOfZeroFill.search(line)
        self.image_base_relocation = self._reg_image_base_relocation.search(line)
        self.highlow = self._reg_highlow.search(line)
        self.absolute = self._reg_absolute.search(line)
        self.image_debug_directory = self._reg_image_debug_directory.search(line)
        self.Type = self._reg_Type.search(line)
        self.SizeOfData = self._reg_SizeOfData.search(line)
        self.AddressOfRawData = self._reg_AddressOfRawData.search(line)
        self.Typelist = self._reg_Typelist.search(line)
        self.InnerType = self._reg_InnerType.findall(line)


for entry in os.listdir(directory):
    if entry.endswith('.txt'):
        filehead = os.path.join(directory, entry)
        with open(filehead) as file:
            for line in file:
                line = line.strip()
                reg_match = _RegExLib(line)

                if reg_match.image_dos_header:
                    i = 1

                if reg_match.e_magic and i == 1:
                    e_magic = reg_match.e_magic.group(1)
                    dictdos['e_magic'] = e_magic

                if reg_match.e_cblp and i == 1:
                    e_cblp = reg_match.e_cblp.group(1)
                    dictdos['e_cblp'] = e_cblp

                if reg_match.e_cp and i == 1:
                    e_cp = reg_match.e_cp.group(1)
                    dictdos['e_cp'] = e_cp

                if reg_match.e_crlc and i == 1:
```

```
    e_crlc = reg_match.e_crlc.group(1)
    dictdos['e_crlc'] = e_crlc

if reg_match.e_cparhdr and i == 1:
    e_cparhdr = reg_match.e_cparhdr.group(1)
    dictdos['e_cparhdr'] = e_cparhdr

if reg_match.e_minalloc and i == 1:
    e_minalloc = reg_match.e_minalloc.group(1)
    dictdos['e_minalloc'] = e_minalloc

if reg_match.e_maxalloc and i == 1:
    e_maxalloc = reg_match.e_maxalloc.group(1)
    dictdos['e_maxalloc'] = e_maxalloc

if reg_match.e_ss and i == 1:
    e_ss = reg_match.e_ss.group(1)
    dictdos['e_ss'] = e_ss

if reg_match.e_sp and i == 1:
    e_sp = reg_match.e_sp.group(1)
    dictdos['e_sp'] = e_sp

if reg_match.e_csum and i == 1:
    e_csum = reg_match.e_csum.group(1)
    dictdos['e_csum'] = e_csum

if reg_match.e_ip and i == 1:
    e_ip = reg_match.e_ip.group(1)
    dictdos['e_ip'] = e_ip

if reg_match.e_cs and i == 1:
    e_cs = reg_match.e_cs.group(1)
    dictdos['e_cs'] = e_cs

if reg_match.e_lfarlc and i == 1:
    e_lfarlc = reg_match.e_lfarlc.group(1)
    dictdos['e_lfarlc'] = e_lfarlc

if reg_match.e_ovno and i == 1:
    e_ovno = reg_match.e_ovno.group(1)
    dictdos['e_ovno'] = e_ovno

if reg_match.e_oemid and i == 1:
    e_oemid = reg_match.e_oemid.group(1)
    dictdos['e_oemid'] = e_oemid

if reg_match.e_oeminfo and i == 1:
    e_oeminfo = reg_match.e_oeminfo.group(1)
```

```
    dictdos['e_oeminfo'] = e_oeminfo

if reg_match.e_lfanew and i == 1:
    e_lfanew = reg_match.e_lfanew.group(1)
    dictdos['e_lfanew'] = e_lfanew
    dictheader['IMAGE_DOS_HEADERS'] = dictdos
    tempheader = copy.deepcopy(dictheader)
    dictsect['DOS_HEADER'] = tempheader
    dictheader.clear()

# --------------------------------

if reg_match.image_nt_headers:
    i = 2

if reg_match.Signature and i == 2:
    Signature = reg_match.Signature.group(1)
    dictnt['Signature'] = Signature
    dictheader['IMAGE_NT_HEADERS'] = dictnt
    tempnt = copy.deepcopy(dictheader)
    dictsect['NT_HEADERS'] = tempnt
    dictheader.clear()

# --------------------------------

if reg_match.image_file_header:
    i = 3

if reg_match.Machine and i == 3:
    Machine = reg_match.Machine.group(1)
    dictfile['Machine'] = Machine

if reg_match.NumberOfSections and i == 3:
    NumberOfSections = reg_match.NumberOfSections.group(1)
    dictfile['NumberOfSections'] = NumberOfSections

if reg_match.TimeDateStamp and i == 3:
    TimeDateStampFile = reg_match.TimeDateStamp.group(1)
    dictfile['TimeDateStampFile'] = TimeDateStampFile

if reg_match.PointerToSymbolTable and i == 3:
    PointerToSymbolTable = reg_match.PointerToSymbolTable.group(1)
    dictfile['PointerToSymbolTable'] = PointerToSymbolTable

if reg_match.NumberOfSymbols and i == 3:
    NumberOfSymbols = reg_match.NumberOfSymbols.group(1)
    dictfile['NumberOfSymbols'] = NumberOfSymbols

if reg_match.SizeOfOptionalHeader and i == 3:
```

```
    SizeOfOptionalHeader = reg_match.SizeOfOptionalHeader.group(1)
    dictfile['SizeOfOptionalHeader'] = SizeOfOptionalHeader

if reg_match.Characteristics and i == 3:
    Characteristics = reg_match.Characteristics.group(1)
    dictfile['Characteristics'] = Characteristics

if reg_match.Flags and i == 3:
    if reg_match.InnerFlags:
        Flags = reg_match.InnerFlags
    dictfile['Flags'] = Flags
    dictheader['IMAGE_FILE_HEADER'] = dictfile
    tempfile = copy.deepcopy(dictheader)
    dictsect['FILE_HEADER'] = tempfile
    dictheader.clear()

# --------------------------------

if reg_match.optional_header:
    i = 4

if reg_match.Magic and i == 4:
    Magic = reg_match.Magic.group(1)
    dictopt['Magic'] = Magic

if reg_match.MajorLinkerVersion and i == 4:
    MajorLinkerVersion = reg_match.MajorLinkerVersion.group(1)
    dictopt['MajorLinkerVersion'] = MajorLinkerVersion

if reg_match.MinorLinkerVersion and i == 4:
    MinorLinkerVersion = reg_match.MinorLinkerVersion.group(1)
    dictopt['MinorLinkerVersion'] = MinorLinkerVersion

if reg_match.SizeOfCode and i == 4:
    SizeOfCode = reg_match.SizeOfCode.group(1)
    dictopt['SizeOfCode'] = SizeOfCode

if reg_match.SizeOfInitializedData and i == 4:
    SizeOfInitializedData = reg_match.SizeOfInitializedData.group(1)
    dictopt['SizeOfInitializedData'] = SizeOfInitializedData

if reg_match.SizeOfUninitializedData and i == 4:
    SizeOfUninitializedData = reg_match.SizeOfUninitializedData.group(1)
    dictopt['SizeOfUninitializedData'] = SizeOfUninitializedData

if reg_match.AddressOfEntryPoint and i == 4:
    AddressOfEntryPoint = reg_match.AddressOfEntryPoint.group(1)
    dictopt['AddressOfEntryPoint'] = AddressOfEntryPoint
```

```
        if reg_match.BaseOfCode and i == 4:
            BaseOfCode = reg_match.BaseOfCode.group(1)
            dictopt['BaseOfCode'] = BaseOfCode

        if reg_match.BaseOfData and i == 4:
            BaseOfData = reg_match.BaseOfData.group(1)
            dictopt['BaseOfData'] = BaseOfData

        if reg_match.ImageBase and i == 4:
            ImageBase = reg_match.ImageBase.group(1)
            dictopt['ImageBase'] = ImageBase

        if reg_match.SectionAlignment and i == 4:
            SectionAlignment = reg_match.SectionAlignment.group(1)
            dictopt['SectionAlignment'] = SectionAlignment

        if reg_match.FileAlignment and i == 4:
            FileAlignment = reg_match.FileAlignment.group(1)
            dictopt['FileAlignment'] = FileAlignment

        if reg_match.MajorOperatingSystemVersion and i == 4:
            MajorOperatingSystemVersion =
reg_match.MajorOperatingSystemVersion.group(1)
            dictopt['MajorOperatingSystemVersion'] = MajorOperatingSystemVersion

        if reg_match.MinorOperatingSystemVersion and i == 4:
            MinorOperatingSystemVersion =
reg_match.MinorOperatingSystemVersion.group(1)
            dictopt['MinorOperatingSystemVersion'] = MinorOperatingSystemVersion

        if reg_match.MajorImageVersion and i == 4:
            MajorImageVersion = reg_match.MajorImageVersion.group(1)
            dictopt['MajorImageVersion'] = MajorImageVersion

        if reg_match.MinorImageVersion and i == 4:
            MinorImageVersion = reg_match.MinorImageVersion.group(1)
            dictopt['MinorImageVersion'] = MinorImageVersion

        if reg_match.MajorSubsystemVersion and i == 4:
            MajorSubsystemVersion = reg_match.MajorSubsystemVersion.group(1)
            dictopt['MajorSubsystemVersion'] = MajorSubsystemVersion

        if reg_match.MinorSubsystemVersion and i == 4:
            MinorSubsystemVersion = reg_match.MinorSubsystemVersion.group(1)
            dictopt['MinorSubsystemVersion'] = MinorSubsystemVersion

        if reg_match.Reserved1 and i == 4:
            Reserved1 = reg_match.Reserved1.group(1)
            dictopt['Reserved1'] = Reserved1
```

```
if reg_match.SizeOfImage and i == 4:
   SizeOfImage = reg_match.SizeOfImage.group(1)
   dictopt['SizeOfImage'] = SizeOfImage

if reg_match.SizeOfHeaders and i == 4:
   SizeOfHeaders = reg_match.SizeOfHeaders.group(1)
   dictopt['SizeOfHeaders'] = SizeOfHeaders

if reg_match.CheckSum and i == 4:
   CheckSum = reg_match.CheckSum.group(1)
   dictopt['CheckSum'] = CheckSum

if reg_match.Subsystem and i == 4:
   Subsystem = reg_match.Subsystem.group(1)
   dictopt['Subsystem'] = Subsystem

if reg_match.DllCharacteristics and i == 4:
   DllCharacteristics = reg_match.DllCharacteristics.group(1)
   dictopt['DllCharacteristics'] = DllCharacteristics

if reg_match.SizeOfStackReserve and i == 4:
   SizeOfStackReserve = reg_match.SizeOfStackReserve.group(1)
   dictopt['SizeOfStackReserve'] = SizeOfStackReserve

if reg_match.SizeOfStackCommit and i == 4:
   SizeOfStackCommit = reg_match.SizeOfStackCommit.group(1)
   dictopt['SizeOfStackCommit'] = SizeOfStackCommit

if reg_match.SizeOfHeapReserve and i == 4:
   SizeOfHeapReserve = reg_match.SizeOfHeapReserve.group(1)
   dictopt['SizeOfHeapReserve'] = SizeOfHeapReserve

if reg_match.SizeOfHeapCommit and i == 4:
   SizeOfHeapCommit = reg_match.SizeOfHeapCommit.group(1)
   dictopt['SizeOfHeapCommit'] = SizeOfHeapCommit

if reg_match.LoaderFlags and i == 4:
   LoaderFlags = reg_match.LoaderFlags.group(1)
   dictopt['LoaderFlags'] = LoaderFlags

if reg_match.NumberOfRvaAndSizes and i == 4:
   NumberOfRvaAndSizes = reg_match.NumberOfRvaAndSizes.group(1)
   dictopt['NumberOfRvaAndSizes'] = NumberOfRvaAndSizes
   dictheader['IMAGE_OPTIONAL_HEADER'] = dictopt
   tempopt = copy.deepcopy(dictheader)
   dictsect['OPTIONAL_HEADER'] = tempopt
   dictheader.clear()
```

```
# --------------------------------

if reg_match.image_section_header:
   i = 5

if reg_match.Name and i == 5:
   Name = reg_match.Name.group(1)
   dictpe['Name'] = Name

if reg_match.Misc and i == 5:
   Misc = reg_match.Misc.group(1)
   dictpe['Misc'] = Misc

if reg_match.Misc_PhysicalAddress and i == 5:
   Misc_PhysicalAddress = reg_match.Misc_PhysicalAddress.group(1)
   dictpe['Misc_PhysicalAddress'] = Misc_PhysicalAddress

if reg_match.Misc_VirtualSize and i == 5:
   Misc_VirtualSize = reg_match.Misc_VirtualSize.group(1)
   dictpe['Misc_VirtualSize'] = Misc_VirtualSize

if reg_match.VirtualAddress and i == 5:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictpe['VirtualAddress'] = VirtualAddress

if reg_match.SizeOfRawData and i == 5:
   SizeOfRawData = reg_match.SizeOfRawData.group(1)
   dictpe['SizeOfRawData'] = SizeOfRawData

if reg_match.PointerToRawData and i == 5:
   PointerToRawData = reg_match.PointerToRawData.group(1)
   dictpe['PointerToRawData'] = PointerToRawData

if reg_match.PointerToRelocations and i == 5:
   PointerToRelocations = reg_match.PointerToRelocations.group(1)
   dictpe['PointerToRelocations'] = PointerToRelocations

if reg_match.PointerToLinenumbers and i == 5:
   PointerToLinenumbers = reg_match.PointerToLinenumbers.group(1)
   dictpe['PointerToLinenumbers'] = PointerToLinenumbers

if reg_match.NumberOfRelocations and i == 5:
   NumberOfRelocations = reg_match.NumberOfRelocations.group(1)
   dictpe['NumberOfRelocations'] = NumberOfRelocations

if reg_match.NumberOfLinenumbers and i == 5:
   NumberOfLinenumbers = reg_match.NumberOfLinenumbers.group(1)
   dictpe['NumberOfLinenumbers'] = NumberOfLinenumbers
```

```
if reg_match.Characteristics and i == 5:
   Characteristics = reg_match.Characteristics.group(1)
   dictpe['Characteristics'] = Characteristics

if reg_match.Flags and i == 5:
   if reg_match.InnerFlags:
      Flags = reg_match.InnerFlags
   dictpe['Flags'] = Flags

if reg_match.Entropy and i == 5:
   Entropy = reg_match.Entropy.group(1)
   dictpe['Entropy'] = Entropy

if reg_match.MD5 and i == 5:
   MD5 = reg_match.MD5.group(1)
   dictpe['MD5'] = MD5

if reg_match.SHA1 and i == 5:
   SHA1 = reg_match.SHA1.group(1)
   dictpe['SHA-1'] = SHA1

if reg_match.SHA256 and i == 5:
   SHA256 = reg_match.SHA256.group(1)
   dictpe['SHA-256'] = SHA256

if reg_match.SHA512 and i == 5:
   SHA512 = reg_match.SHA512.group(1)
   dictpe['SHA-512'] = SHA512
   List.insert(j, copy.deepcopy(dictpe))
   dictheader[Name] = List[j]

   j = j + 1

# --------------------------------

if reg_match.image_directory_entry_export:
   temppe = copy.deepcopy(dictheader)
   dictsect['PE_SECTIONS'] = temppe
   dictheader.clear()
   i = 6

if reg_match.VirtualAddress and i == 6:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdiexport['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 6:
   Size = reg_match.Size.group(1)
   dictdiexport['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_EXPORT'] = dictdiexport
```

```
if reg_match.image_directory_entry_import:
   i = 7

if reg_match.VirtualAddress and i == 7:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdiimport['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 7:
   Size = reg_match.Size.group(1)
   dictdiimport['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_IMPORT'] = dictdiimport

if reg_match.image_directory_entry_resource:
   i = 8

if reg_match.VirtualAddress and i == 8:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdires['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 8:
   Size = reg_match.Size.group(1)
   dictdires['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_RESOURCE'] = dictdires

if reg_match.image_directory_entry_exception:
   i = 9

if reg_match.VirtualAddress and i == 9:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdiexcep['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 9:
   Size = reg_match.Size.group(1)
   dictdiexcep['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_EXCEPTION'] = dictdiexcep

if reg_match.image_directory_entry_security:
   i = 10

if reg_match.VirtualAddress and i == 10:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdisec['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 10:
   Size = reg_match.Size.group(1)
   dictdisec['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_SECURITY'] = dictdisec
```

```
if reg_match.image_directory_entry_basereloc:
   i = 11

if reg_match.VirtualAddress and i == 11:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdibase['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 11:
   Size = reg_match.Size.group(1)
   dictdibase['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_BASERELOC'] = dictdibase

if reg_match.image_directory_entry_debug:
   i = 12

if reg_match.VirtualAddress and i == 12:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdidebug['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 12:
   Size = reg_match.Size.group(1)
   dictdidebug['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_DEBUG'] = dictdidebug

if reg_match.image_directory_entry_copyright:
   i = 13

if reg_match.VirtualAddress and i == 13:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdicopy['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 13:
   Size = reg_match.Size.group(1)
   dictdicopy['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_COPYRIGHT'] = dictdicopy

if reg_match.image_directory_entry_globalptr:
   i = 14

if reg_match.VirtualAddress and i == 14:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdiglob['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 14:
   Size = reg_match.Size.group(1)
   dictdiglob['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_GLOBALPTR'] = dictdiglob

if reg_match.image_directory_entry_tls:
```

```
    i = 15

if reg_match.VirtualAddress and i == 15:
    VirtualAddress = reg_match.VirtualAddress.group(1)
    dictditls['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 15:
    Size = reg_match.Size.group(1)
    dictditls['Size'] = Size
    dictheader['IMAGE_DIRECTORY_ENTRY_TLS'] = dictditls

if reg_match.image_directory_entry_load_config:
    i = 16

if reg_match.VirtualAddress and i == 16:
    VirtualAddress = reg_match.VirtualAddress.group(1)
    dictdiload['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 16:
    Size = reg_match.Size.group(1)
    dictdiload['Size'] = Size
    dictheader['IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG'] = dictdiload

if reg_match.image_directory_entry_bound_import:
    i = 21

if reg_match.VirtualAddress and i == 21:
    VirtualAddress = reg_match.VirtualAddress.group(1)
    dictdibound['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 21:
    Size = reg_match.Size.group(1)
    dictdibound['Size'] = Size
    dictheader['IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT'] = dictdibound

if reg_match.image_directory_entry_iat:
    i = 17

if reg_match.VirtualAddress and i == 17:
    VirtualAddress = reg_match.VirtualAddress.group(1)
    dictdiiat['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 17:
    Size = reg_match.Size.group(1)
    dictdiiat['Size'] = Size
    dictheader['IMAGE_DIRECTORY_ENTRY_IAT'] = dictdiiat

if reg_match.image_directory_entry_delay_import:
    i = 18
```

```
if reg_match.VirtualAddress and i == 18:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdidelay['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 18:
   Size = reg_match.Size.group(1)
   dictdidelay['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT'] = dictdidelay

if reg_match.image_directory_entry_com_descriptor:
   i = 19

if reg_match.VirtualAddress and i == 19:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictdicom['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 19:
   Size = reg_match.Size.group(1)
   dictdicom['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR'] = dictdicom

if reg_match.image_directory_entry_reserved:
   i = 20

if reg_match.VirtualAddress and i == 20:
   VirtualAddress = reg_match.VirtualAddress.group(1)
   dictres['VirtualAddress'] = VirtualAddress

if reg_match.Size and i == 20:
   Size = reg_match.Size.group(1)
   dictres['Size'] = Size
   dictheader['IMAGE_DIRECTORY_ENTRY_RESERVED'] = dictres
   tempdir = copy.deepcopy(dictheader)
   dictsect['Directories'] = tempdir
   dictheader.clear()
# --------------------------------
if reg_match.image_import_descriptor:
   i = 22
   if dll == 1:
      dictimp[DLLType] = Listdll
      Listimp.insert(e, copy.deepcopy(dictimp))
      dictheader[Name] = Listimp[e]
      Listdll.clear()
      dictimp.clear()
      e = e + 1
      dll = 0
   else:
      e = 0
```

```
if reg_match.OriginalFirstThunk and i == 22:
   OriginalFirstThunk = reg_match.OriginalFirstThunk.group(1)
   dictimp['OriginalFirstThunk'] = OriginalFirstThunk

if reg_match.Characteristics and i == 22:
   Characteristics = reg_match.Characteristics.group(1)
   dictimp['Characteristics'] = Characteristics

if reg_match.TimeDateStamp and i == 22:
   TimeDateStamp = reg_match.TimeDateStamp.group(1)
   dictimp['TimeDateStamp'] = TimeDateStamp

if reg_match.ForwarderChain and i == 22:
   ForwarderChain = reg_match.ForwarderChain.group(1)
   dictimp['ForwarderChain'] = ForwarderChain

if reg_match.Name and i == 22:
   Name = reg_match.Name.group(1)
   dictimp['Name'] = Name

if reg_match.FirstThunk and i == 22:
   FirstThunk = reg_match.FirstThunk.group(1)
   dictimp['FirstThunk'] = FirstThunk
   dll = 1

if reg_match.DLL and i == 22:
   DLL = reg_match.DLL.group(2)
   DLLType = reg_match.DLL.group(1)
   Listdll.append(DLL)

# --------------------------------

if reg_match.image_resource_directory:
   if dll == 1:
      dictimp[DLLType] = Listdll
      Listimp.insert(e, copy.deepcopy(dictimp))
      dictheader[Name] = Listimp[e]
      tempimp = copy.deepcopy(dictheader)
      dictsect['Imported_Symbols'] = tempimp
      dictheader.clear()
      dll = 0

# --------------------------------

if reg_match.image_tls_directory:
   i = 23
   if dll == 1:
      dictimp[DLLType] = Listdll
```

```
        Listimp.insert(e, copy.deepcopy(dictimp))
        dictheader[Name] = Listimp[e]
        tempimp = copy.deepcopy(dictheader)
        dictsect['Imported_Symbols'] = tempimp
        dictheader.clear()
        dll = 0

if reg_match.StartAddressOfRawData and i == 23:
    StartAddressOfRawData = reg_match.StartAddressOfRawData.group(1)
    dicttls['StartAddressOfRawData'] = StartAddressOfRawData

if reg_match.EndAddressOfRawData and i == 23:
    EndAddressOfRawData = reg_match.EndAddressOfRawData.group(1)
    dicttls['EndAddressOfRawData'] = EndAddressOfRawData

if reg_match.AddressOfIndex and i == 23:
    AddressOfIndex = reg_match.AddressOfIndex.group(1)
    dicttls['AddressOfIndex'] = AddressOfIndex

if reg_match.AddressOfCallBacks and i == 23:
    AddressOfCallBacks = reg_match.AddressOfCallBacks.group(1)
    dicttls['AddressOfCallBacks'] = AddressOfCallBacks

if reg_match.SizeOfZeroFill and i == 23:
    SizeOfZeroFill = reg_match.SizeOfZeroFill.group(1)
    dicttls['SizeOfZeroFill'] = SizeOfZeroFill

if reg_match.Characteristics and i == 23:
    Characteristics = reg_match.Characteristics.group(1)
    dicttls['Characteristics'] = Characteristics
    dictheader['IMAGE_TLS_DIRECTORY'] = dicttls
    temptls = copy.deepcopy(dictheader)
    dictsect['TLS'] = temptls
    dictheader.clear()

# --------------------------------

if reg_match.image_base_relocation:
    i = 24
    if dll == 1:
        dictimp[DLLType] = Listdll
        Listimp.insert(e, copy.deepcopy(dictimp))
        dictheader[Name] = Listimp[e]
        tempimp = copy.deepcopy(dictheader)
        dictsect['Imported_Symbols'] = tempimp
        dictheader.clear()
        dll = 0

if reg_match.highlow and i == 24:
```

```
            highlow = reg_match.highlow.group(1)
            Listhighlow.append(highlow)
            basereloc = 1

        if reg_match.absolute and i == 24:
            absolute = reg_match.absolute.group(1)
            Listabsolute.append(absolute)
            basereloc = 1

        # --------------------------------

        if reg_match.image_load_config_directory:
            i = 25
            if dll == 1:
                dictimp[DLLType] = Listdll
                Listimp.insert(e, copy.deepcopy(dictimp))
                dictheader[Name] = Listimp[e]
                tempimp = copy.deepcopy(dictheader)
                dictsect['Imported_Symbols'] = tempimp
                dictheader.clear()
                dll = 0

        if reg_match.Size and i == 25:
            Size = reg_match.Size.group(1)
            dictload['Size'] = Size

        if reg_match.TimeDateStamp and i == 25:
            TimeDateStamp = reg_match.TimeDateStamp.group(1)
            dictload['TimeDateStamp'] = TimeDateStamp

        if reg_match.MajorVersion and i == 25:
            MajorVersion = reg_match.MajorVersion.group(1)
            dictload['MajorVersion'] = MajorVersion

        if reg_match.MinorVersion and i == 25:
            MinorVersion = reg_match.MinorVersion.group(1)
            dictload['MinorVersion'] = MinorVersion

        if reg_match.GlobalFlagsClear and i == 25:
            GlobalFlagsClear = reg_match.GlobalFlagsClear.group(1)
            dictload['GlobalFlagsClear'] = GlobalFlagsClear

        if reg_match.GlobalFlagsSet and i == 25:
            GlobalFlagsSet = reg_match.GlobalFlagsSet.group(1)
            dictload['GlobalFlagsSet'] = GlobalFlagsSet

        if reg_match.CriticalSectionDefaultTimeout and i == 25:
            CriticalSectionDefaultTimeout =
reg_match.CriticalSectionDefaultTimeout.group(1)
```

```
            dictload['CriticalSectionDefaultTimeout'] = CriticalSectionDefaultTimeout

        if reg_match.DeCommitFreeBlockThreshold and i == 25:
            DeCommitFreeBlockThreshold =
reg_match.DeCommitFreeBlockThreshold.group(1)
            dictload['DeCommitFreeBlockThreshold'] = DeCommitFreeBlockThreshold

        if reg_match.DeCommitTotalFreeThreshold and i == 25:
            DeCommitTotalFreeThreshold =
reg_match.DeCommitTotalFreeThreshold.group(1)
            dictload['DeCommitTotalFreeThreshold'] = DeCommitTotalFreeThreshold

        if reg_match.LockPrefixTable and i == 25:
            LockPrefixTable = reg_match.LockPrefixTable.group(1)
            dictload['LockPrefixTable'] = LockPrefixTable

        if reg_match.MaximumAllocationSize and i == 25:
            MaximumAllocationSize = reg_match.MaximumAllocationSize.group(1)
            dictload['MaximumAllocationSize'] = MaximumAllocationSize

        if reg_match.VirtualMemoryThreshold and i == 25:
            VirtualMemoryThreshold = reg_match.VirtualMemoryThreshold.group(1)
            dictload['VirtualMemoryThreshold'] = VirtualMemoryThreshold

        if reg_match.ProcessHeapFlags and i == 25:
            ProcessHeapFlags = reg_match.ProcessHeapFlags.group(1)
            dictload['ProcessHeapFlags'] = ProcessHeapFlags

        if reg_match.ProcessAffinityMask and i == 25:
            ProcessAffinityMask = reg_match.ProcessAffinityMask.group(1)
            dictload['ProcessAffinityMask'] = ProcessAffinityMask

        if reg_match.CSDVersion and i == 25:
            CSDVersion = reg_match.CSDVersion.group(1)
            dictload['CSDVersion'] = CSDVersion

        if reg_match.Reserved1 and i == 25:
            Reserved1 = reg_match.Reserved1.group(1)
            dictload['Reserved1'] = Reserved1

        if reg_match.EditList and i == 25:
            EditList = reg_match.EditList.group(1)
            dictload['EditList'] = EditList

        if reg_match.SecurityCookie and i == 25:
            SecurityCookie = reg_match.SecurityCookie.group(1)
            dictload['SecurityCookie'] = SecurityCookie

        if reg_match.SEHandlerTable and i == 25:
```

```
            SEHandlerTable = reg_match.SEHandlerTable.group(1)
            dictload['SEHandlerTable'] = SEHandlerTable

        if reg_match.SEHandlerCount and i == 25:
            SEHandlerCount = reg_match.SEHandlerCount.group(1)
            dictload['SEHandlerCount'] = SEHandlerCount

        if reg_match.GuardCFCheckFunctionPointer and i == 25:
            GuardCFCheckFunctionPointer =
reg_match.GuardCFCheckFunctionPointer.group(1)
            dictload['GuardCFCheckFunctionPointer'] = GuardCFCheckFunctionPointer

        if reg_match.Reserved2 and i == 25:
            Reserved2 = reg_match.Reserved2.group(1)
            dictload['Reserved2'] = Reserved2

        if reg_match.GuardCFFunctionTable and i == 25:
            GuardCFFunctionTable = reg_match.GuardCFFunctionTable.group(1)
            dictload['GuardCFFunctionTable'] = GuardCFFunctionTable

        if reg_match.GuardCFFunctionCount and i == 25:
            GuardCFFunctionCount = reg_match.GuardCFFunctionCount.group(1)
            dictload['GuardCFFunctionCount'] = GuardCFFunctionCount

        if reg_match.GuardFlags and i == 25:
            GuardFlags = reg_match.GuardFlags.group(1)
            dictload['GuardFlags'] = GuardFlags
            dictheader['IMAGE_LOAD_CONFIG_DIRECTORY'] = dictload
            tempload = copy.deepcopy(dictload)
            dictsect['LOAD_CONFIG'] = tempload
            dictheader.clear()

        # --------------------------------

        if reg_match.image_debug_directory:
            i = 26
            if dll == 1:
                dictimp[DLLType] = Listdll
                Listimp.insert(e, copy.deepcopy(dictimp))
                dictheader[Name] = Listimp[e]
                tempimp = copy.deepcopy(dictheader)
                dictsect['Imported_Symbols'] = tempimp
                dictheader.clear()
                dll = 0

        if reg_match.Characteristics and i == 26:
            Characteristics = reg_match.Characteristics.group(1)
            dictdebug['Characteristics'] = Characteristics
```

```
    if reg_match.TimeDateStamp and i == 26:
        TimeDateStamp = reg_match.TimeDateStamp.group(1)
        dictdebug['TimeDateStamp'] = TimeDateStamp

    if reg_match.MajorVersion and i == 26:
        MajorVersion = reg_match.MajorVersion.group(1)
        dictdebug['MajorVersion'] = MajorVersion

    if reg_match.MinorVersion and i == 26:
        MinorVersion = reg_match.MinorVersion.group(1)
        dictdebug['MinorVersion'] = MinorVersion

    if reg_match.Type and i == 26:
        if typedebug == 0:
            Type = reg_match.Type.group(1)
            dictdebug['Type'] = Type
            typedebug = 1

    if reg_match.SizeOfData and i == 26:
        SizeOfData = reg_match.SizeOfData.group(1)
        dictdebug['SizeOfData'] = SizeOfData

    if reg_match.AddressOfRawData and i == 26:
        AddressOfRawData = reg_match.AddressOfRawData.group(1)
        dictdebug['AddressOfRawData'] = AddressOfRawData

    if reg_match.PointerToRawData and i == 26:
        PointerToRawData = reg_match.PointerToRawData.group(1)
        dictdebug['PointerToRawData'] = PointerToRawData

    if reg_match.Typelist and i == 26:
        if reg_match.InnerType:
            Type = reg_match.InnerType
        dictdebug['Types'] = Type
        dictheader['IMAGE_DEBUG_DIRECTORY'] = dictdebug
        tempdebug = copy.deepcopy(dictdebug)
        dictsect['Debug_information'] = tempdebug
        dictheader.clear()


    # --------------------------------

if basereloc == 1:
    dictreloc['highlow'] = Listhighlow
    dictreloc['absolute'] = Listabsolute
    dictheader['IMAGE_BASE_RELOCATION'] = dictreloc
    tempreloc = copy.deepcopy(dictheader)
    dictsect['Base_relocations'] = tempreloc
    dictheader.clear()
```

```
        Listsect[final] = copy.deepcopy(dictsect)
        dictfinal[entry] = Listsect[final]
        final = final + 1
        dictsect.clear()
        dictdos.clear()
        dictnt.clear()
        dictfile.clear()
        dictopt.clear()
        dictpe.clear()
        dictdiexport.clear()
        dictdiimport.clear()
        dictdires.clear()
        dictdiexcep.clear()
        dictdisec.clear()
        dictdibase.clear()
        dictdidebug.clear()
        dictdicopy.clear()
        dictdiglob.clear()
        dictditls.clear()
        dictdiload.clear()
        dictdibound.clear()
        dictdiiat.clear()
        dictdidelay.clear()
        dictdicom.clear()
        dictres.clear()
        dictimp.clear()
        dicttls.clear()
        dictreloc.clear()
        dictdebug.clear()
        dictload.clear()
        Listdll.clear()
        Listhighlow.clear()
        Listabsolute.clear()
        Listfiles.clear()
        Listimp.clear()
        f = 0
        j = 0
        dll = 0
        e = 0
        basereloc = 0
        typedebug = 0

# creating and writing to the json file
out_file = open("Malware.json", "w")
json.dump(dictfinal, out_file, indent=4)
out_file.close()
```

Markel.py

```
import json
import csv
import copy

malware_file = '//Users/cedricecran/Desktop/Malware.json'

benign_file = '//Users/cedricecran/Desktop/Benign.json'

Flags = []

column = ['Files', 'Class', 'SizeOfInitializedData',
'BaseOfData', 'SizeOfCode',
'MajorOperatingSystemVersion', 'MinorOperatingSystemVersion',
'SizeOfStackReserve', 'HighEntropy']

j = 0

with open(malware_file) as f:
   data = json.load(f)

with open(benign_file) as r:
   data2 = json.load(r)


for file, sections in data.items():
   if "FILE_HEADER" in sections:
      sectionstemp = sections["FILE_HEADER"]
      for headers, values in sectionstemp.items():
         if "Flags" in values:
            Flags.extend(values["Flags"])

for file, sections in data2.items():
   if "FILE_HEADER" in sections:
      sectionstemp = sections["FILE_HEADER"]
      for headers, values in sectionstemp.items():
         if "Flags" in values:
            Flags.extend(values["Flags"])


Flags = list(dict.fromkeys(Flags))

column.extend(Flags)

with open('Markel.csv', 'w', newline='') as csvfile:
   writer = csv.writer(csvfile)
   writer.writerow(column)
```

```
with open('Markel.csv', 'a') as csvfile:
    writer = csv.writer(csvfile)
    for file, sections in data.items():
        columntemp = copy.deepcopy(column)
        columntemp[0] = file
        columntemp[1] = 1  # This means malware!

        if "OPTIONAL_HEADER" in sections:
            sectionstemp = sections["OPTIONAL_HEADER"]
            for headers, values in sectionstemp.items():
                if "SizeOfInitializedData" in values:
                    x = int(values["SizeOfInitializedData"], 16)
                    columntemp[2] = x
                if "BaseOfData" in values:
                    x = int(values["BaseOfData"], 16)
                    columntemp[3] = x
                if "SizeOfCode" in values:
                    x = int(values["SizeOfCode"], 16)
                    columntemp[4] = x
                if "MajorOperatingSystemVersion" in values:
                    x = int(values["MajorOperatingSystemVersion"], 16)
                    columntemp[5] = x
                if "MinorOperatingSystemVersion" in values:
                    x = int(values["MinorOperatingSystemVersion"], 16)
                    columntemp[6] = x
                if "SizeOfStackReserve" in values:
                    x = int(values["SizeOfStackReserve"], 16)
                    columntemp[7] = x

        if "PE_SECTIONS" in sections:
            sectionstemp = sections["PE_SECTIONS"]
            for headers, values in sectionstemp.items():
                if "Entropy" in values:
                    x = float(values["Entropy"])
                    if x > 7.0:
                        columntemp[8] = 1

        if "FILE_HEADER" in sections:
            sectionstemp = sections["FILE_HEADER"]
            for headers, values in sectionstemp.items():
                if "Flags" in values:
                    for i in values["Flags"]:
                        if (i in columntemp):
                            columntemp[columntemp.index(i)] = 1



        list_columntemp = iter(columntemp)
        z = 2
```

```
    for z, num in enumerate(list_columntemp):
        if num in column:
            columntemp[z] = 0

    writer.writerow(columntemp)

    columntemp.clear()
    sectionstemp.clear()

for file, sections in data2.items():
    columntemp = copy.deepcopy(column)
    columntemp[0] = file
    columntemp[1] = 0  # This means benign!

    if "OPTIONAL_HEADER" in sections:
        sectionstemp = sections["OPTIONAL_HEADER"]
        for headers, values in sectionstemp.items():
            if "SizeOfInitializedData" in values:
                x = int(values["SizeOfInitializedData"], 16)
                columntemp[2] = x
            if "BaseOfData" in values:
                x = int(values["BaseOfData"], 16)
                columntemp[3] = x
            if "SizeOfCode" in values:
                x = int(values["SizeOfCode"], 16)
                columntemp[4] = x
            if "MajorOperatingSystemVersion" in values:
                x = int(values["MajorOperatingSystemVersion"], 16)
                columntemp[5] = x
            if "MinorOperatingSystemVersion" in values:
                x = int(values["MinorOperatingSystemVersion"], 16)
                columntemp[6] = x
            if "SizeOfStackReserve" in values:
                x = int(values["SizeOfStackReserve"], 16)
                columntemp[7] = x

    if "PE_SECTIONS" in sections:
        sectionstemp = sections["PE_SECTIONS"]
        for headers, values in sectionstemp.items():
            if "Entropy" in values:
                x = float(values["Entropy"])
                if x > 7.0:
                    columntemp[8] = 1

    if "FILE_HEADER" in sections:
        sectionstemp = sections["FILE_HEADER"]
        for headers, values in sectionstemp.items():
            if "Flags" in values:
                for i in values["Flags"]:
```

```
            if (i in columntemp):
               columntemp[columntemp.index(i)] = 1


    list_columntemp = iter(columntemp)
    z = 2
    for z, num in enumerate(list_columntemp):
       if num in column:
          columntemp[z] = 0

    writer.writerow(columntemp)

    columntemp.clear()
    sectionstemp.clear()
```

MarkelDT.py

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn import tree

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, f1_score


dataset = pd.read_csv('//Users/cedricecran/Desktop/Markel.csv')


X = dataset.iloc[:, 2:27].values #Attributes

y = dataset.iloc[:, 1].values #Class (benign or malware)


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=25)


regressor = tree.DecisionTreeClassifier()

regressor.fit(X_train, y_train)

y_pred = regressor.predict(X_test)
```

```
print (confusion_matrix(y_test, y_pred))

print (classification_report(y_test, y_pred))

print (accuracy_score(y_test, y_pred))
```