**Malware Categorisation Using Machine Learning**

A dissertation submitted in partial fulfilment of

the requirements for the degree of

BACHELOR OF ENGINEERING in Computer Science

in

The Queen's University of Belfast

By

Jon Kyle

09/05/2017

Table of Contents

[Acknowledgements 4](#_Toc480458771)

[Abstract 4](#_Toc480458772)

[Introduction 4](#_Toc480458773)

[Literature Review and Problem Specification 5](#_Toc480458774)

[Dynamic Analysis 6](#_Toc480458775)

[Static Analysis 6](#_Toc480458776)

[PE File Structure 7](#_Toc480458777)

[Proposed Machine Learning Features 8](#_Toc480458778)

[File Entropy 8](#_Toc480458779)

[Ratio of Code to Initialised Data 8](#_Toc480458780)

[Major Image Version 8](#_Toc480458781)

[Number Of Sections 8](#_Toc480458782)

[Common DLL Imports 9](#_Toc480458783)

[Dimensionality Reduction Techniques 10](#_Toc480458784)

[Principle Component Analysis (PCA) 10](#_Toc480458785)

[Non-Negative Matrix Factorisation (NMF) 10](#_Toc480458786)

[Clustering Algorithms 11](#_Toc480458787)

[KMeans 11](#_Toc480458788)

[Mean Shift 12](#_Toc480458789)

[\*Explanation of which is going to be used and why 12](#_Toc480458790)

[System Requirements Specification 12](#_Toc480458791)

[System constraints 12](#_Toc480458792)

[Functional Requirements 12](#_Toc480458793)

[Non-functional requirements 13](#_Toc480458794)

[User Interface requirements 13](#_Toc480458795)

[User Characteristics 13](#_Toc480458796)

[Design 14](#_Toc480458797)

[System Architecture 14](#_Toc480458798)

[PE File Parser 14](#_Toc480458799)

[Feature Extraction 14](#_Toc480458800)

[User Interface 14](#_Toc480458801)

[Clustering and Categorisation 15](#_Toc480458802)

[User Interface Design 15](#_Toc480458803)

[Software Design 16](#_Toc480458804)

[Implementation 18](#_Toc480458805)

[Choice of languages/libraries 18](#_Toc480458806)

[Development Environment 19](#_Toc480458807)

[Key Implementation Decisions 19](#_Toc480458808)

[Implementation of Important Functions/Algorithms 22](#_Toc480458809)

[Calculation of Entropy 22](#_Toc480458810)

[Reading Values from PE Files 23](#_Toc480458811)

[Conversion of Byte Data to Decimal 24](#_Toc480458812)

[Getting Names of .dll Imports from a PE File 24](#_Toc480458813)

[Automation of Optimizing KMeans Parameters 25](#_Toc480458814)

[Implementation of individual components 25](#_Toc480458815)

[Testing 26](#_Toc480458816)

[Testing approach 26](#_Toc480458817)

[System Evaluation and Experimental Results 26](#_Toc480458818)

[Conclusion 26](#_Toc480458819)

[References 27](#_Toc480458820)

[Appendices 29](#_Toc480458821)

[Appendix 1. Meeting Minutes 29](#_Toc480458822)

[Appendix 2. PE32 Class Full UML 36](#_Toc480458823)

[Appendix 3. peparse.py Sample Command Line Output 37](#_Toc480458824)

# **Acknowledgements**

# **Abstract**

Malware categorization techniques play a critical role in safeguarding existing technical infrastructure. Malware comes in many forms and uses many different attack vectors to gain access to computers, to cause undesired and potentially harmful behaviors to occur.

This project presents a heuristic-based static analysis tool which extracts features from the binary representation of WIN32 Portable Executable files. These features are then analysed using a number of dimensionality reduction techniques including Principal Component Analysis and Non-Negative Matrix Factorisation, to maximize the variance within the data. This processed data is then used as an input to the KMeans clustering algorithm to categorise the files into clusters.

# **Introduction**

Malware can often cause harm to companies and in some cases individuals through the interception of network data, encryption or deletion of files, and in extreme cases can cause catastrophic failure on a hardware level through infecting or replacing existing firmware.

The symptoms of malware differ, depending on the malware type or family; As the symptoms differ, so does the potential risk to systems infected by the malware.

Categorization of malware allows us to make an approximation of the potential risk a file poses. Though there are many existing anti-virus scanners available, the results of these scanners can disagree when it comes to the categorization of a piece of malware, and in some rare cases an anti-virus may classify a file as safe, when in fact it is not.

Malware comes in many forms and employs many different propagation techniques to deliver payloads to vulnerable machines. Common propagation techniques include Virus, and Worm which self-replicate locally or across networks. Payloads can range from relatively benign adware, to ransomware and rootkits.

Since 2012 the number of network-capable devices has increased from 8.7 Billion to 28.4 Billion and is expected to increase to 50.1 Billion by 2020 [1], this poses a problem for Security Operation Centres as this increasing number of devices gives attackers an increasing number of targets and opportunities to exploit the networked nature of these devices for malicious purposes [2]. Regardless of platform, there is likely no perfect way of detecting or categorizing malicious files as the evasion and propagation techniques used by attackers increase in complexity, but as the techniques used by attackers evolve we need a similar evolution in detection techniques in order for Security Operations Centres to quickly and accurately assess the potential risk associated with a file and put the necessary mitigations and remediation in place.

For this project, we will be focusing on heuristic based static analysis of the malware file structure to determine its relative risk. Our goal is to investigate this method of malware categorization, which will involve data-mining a set of Windows PE files and make use of existing machine learning libraries and clustering algorithms to categorise these files.

# **Literature Review and Problem Specification**

**Exploration of Existing Malware Detection Methods**

Most anti-virus software looks at syntactic signatures of a file to determine whether it is malicious, these signatures can be made up of combinations, or patterns, of instructions which have been previously flagged as malicious. At a higher level, anti-virus software can look at the hash of a file and compare to a database of hash values from known malware samples. The disadvantage of the signature based approach most anti-virus software takes is that small changes to malware source code can result in a signature which may not have been seen before and therefore not in the most up-to-date database of signature definitions. Similarly, these signature based methods provide little protection from zero-day attacks. Metamorphic and polymorphic malware also causes issues for signature based detection systems as one piece of malware can have multiple signatures.

There has been some exploration into semantic-based detection systems as a way to combat the previously mentioned methods of evading syntactic signature-based systems. These systems seek to identify *what* an executable is doing rather than *how* it is doing it through use of abstractions and templates. [3]

## **Dynamic Analysis**

Dynamic analysis involves executing the malware in a sandboxed environment, often on a virtual machine. The run-time behaviors of the malware are then observed through the tracking of system events and network traffic. This approach to malware analysis excels as it removes a large amount of the guess work which would be present using static analysis because the malware is being run in a relatively realistic environment. [4]

Dynamic Analysis has its drawbacks, there is a non-zero chance of infecting other machines on the same network. Researchers have also observed malware which behaves differently when run in a virtualized environment[5], [6], which prevents accurate categorization.

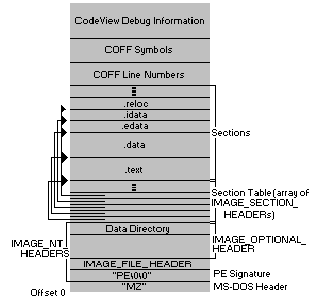
## **Static Analysis**

Static analysis is a technique used to inspect properties of a piece of malware without loading the executable into memory. It involves looking at the heuristics of the malware file for features which reveal behaviors the malware may exhibit if it were executed. The obvious advantage of this approach is that, since the malware is not being executed on the machine, then there is no chance of the malware spreading to other machines on the network and this also prevents damage to the host operating system. Static analysis also has limitations, such as its inability to fully analyze the behavior of a binary file which uses self-modifying code techniques or relies on data which is not static e.g. The current date/time. [7]. Accurate results can be computationally expensive to obtain. The processing time poses a problem for systems which rely on real-time detection of threats, although this is not the case for all static analysis systems as near real-time detection was observed with the PE Miner tool created by Shafiq et al. [8]

Static analysis has also been shown to be susceptible to reduction in effectiveness when used with obfuscated binaries [9],[10].

## **PE File Structure**

The Windows Portable Executable File Format is an extension of the Common Object File Format. There is a large amount of information available from an executable file including its target operating system, the date and time it was compiled and the DLLs it imports. A typical PE32 file consists of several sub-structures shown below.



**Fig1.** The PE File Format [11]

Each of the components of a PE File have a clearly designed specification which can be found at [11].

This specification forms the basis for our static analysis system, allowing us to create a parser which is able to extract information and features of the file quickly and conveniently. There are structural differences between 32 bit binaries and 64 bit binaries. For our solution we are targeting 32 bit binaries only.

## **Proposed Machine Learning Features**

### File Entropy

This is a number between 0 and 8 and represents the randomness of the data within the executable. According to the results proposed by Lyda and Hamrock [12], packed files and encrypted executables usually have a higher value for entropy when compared to non-packed and non-encrypted executables. Since a large proportion of malware use these techniques while benign files rarely use them we can use this entropy value as an indicator of whether a file has been packed or encrypted and therefore if it is benign or malicious. This indicator alone would obviously not be good enough to distinguish between a benign or malicious with 100% accuracy but it is a useful metric nonetheless.

### Ratio of Code to Initialised Data

This is the ratio of executable code to initialised data within the PE File. During my initial research, mining data from my dataset of malicious and benign files, I observed that benign files often have a large amount of data but a small amount of executable code in comparison. Conversely I observed that malicious files tend to have little to no data but a larger proportion of executable code. To normalise these observations and make them more meaningful I have chosen to express this ratio as a single machine learning feature as opposed to two distinct features consisting of the size of code and size of initialised data.

### Major Image Version

This number is the version of the executable. According to Raman [13] malicious files often have a value of zero and benign files often have a higher value. I observed the same pattern during my initial research and as such this feature has been the best indicator of whether a file is benign or malicious, that I have come across.

### Number Of Sections

This is the number of section headers in the section header table. As Yonts[14] observed, the number of sections in an executable can be a very good indicator of whether the file is benign or malicious. In general, benign files range from 0-10 sections while malicious files almost always have 3 or 4 sections. I was also able to observe this pattern when data mining my dataset.

### Common DLL Imports

The DLLs imported by an executable paint a very good picture of the functionality and behaviour of the executable. For example, if an executable is importing Wsock32.dll then you can surmise that the executable makes use of networking. Although there will be overlap between the imports of benign files and malicious files, we should be able to classify the malware by family using this information as behaviours differ between families. We will be using the same technique Shafiq et. Al. [8] proposed where we look for a particular import and set a flag to true or false depending on the whether it was present or not.

|  |  |
| --- | --- |
| **DLL Name** | **Function** |
| WS2\_32.dll | Networking |
| Wsock32.dll | Networking |
| Kernel32.dll | Memory Management/ IO operations |
| WININET.DLL | Networking |

## **Dimensionality Reduction Techniques**

Dimensionality reduction techniques seek to find the subset of data which best represents the whole collection of data for a particular object. That is, dimensionality reduction involves mapping a high-dimensionality dataset to a lower-dimension sub-space by filtering out data which is deemed non-essential. This has the benefit of providing an easier visualization of the whole data-set as it is not practical to represent anything with more than 3 dimensions on a graph. Low-dimensional datasets also improve performance when processing data with machine learning algorithms.[15]

### Principle Component Analysis (PCA)

Principle component Analysis is the most common form of dimensionality reduction and it achieves this though factorization of a matrix (our dataset) into two sub matrices, one of which is the Principle Component Model which best represents the data in the original matrix, this is the data which will be plotted on a graph, and the other is considered noise. [16]

### Non-Negative Matrix Factorisation (NMF)

Much like Principle Component Analysis, Non-Negative Matrix Factorisation involves factorizing a high-dimensionality matrix into two smaller matrices. Where this approach differs is that it operates under the constraint that the numerical data which makes up the data-set or matrix is non-negative. As a result of this constraint NMF can result in a different representation of the original matrix.[17]

## **Clustering Algorithms**

### KMeans

The K-Means algorithm divides provided data-points into K clusters such that the sum of squares within each cluster is minimized.[18][19] A crucial component of the KMeans algorithm is the number of clusters which the algorithm is instructed to create. Unlike other clustering algorithms such as MeanShift the KMeans algorithm does not decide the number of clusters internally based on the data.

#### Silhouette Metrics

There are several techniques which can be used to determine the optimal number of clusters, one of which is silhouette metric. This number is essentially a score for how well each data-point fits into its cluster. This metric is expressed as a score between -1 and 1. A score tending towards -1 would indicate that there may be to many or too few clusters and that the data does not belong in this cluster. A score tending towards 1 would indicate that the data has been clustered correctly. A score around 0 indicates that the data may fit more than one cluster [18].

|  |  |  |  |
| --- | --- | --- | --- |
| **Number Of Clusters** | **Decompositon** | **Clustering** | **Silhouette Score** |
| 2 | NMF | KMeans | 0.810315175 |
| 2 | PCA | KMeans | 0.704351353 |
| 3 | NMF | MeanShift | 0.660036554 |
| 3 | NMF | KMeans | 0.647568178 |
| 3 | PCA | KMeans | 0.63639525 |
| 4 | PCA | KMeans | 0.628463924 |
| 4 | NMF | KMeans | 0.626236348 |
| 4 | PCA | MeanShift | 0.617196567 |
| 5 | NMF | KMeans | 0.595266334 |
| 6 | PCA | KMeans | 0.569866616 |
| 6 | NMF | KMeans | 0.540762171 |
| 5 | PCA | KMeans | 0.510923939 |

#### Elbow Method

The Elbow Method is an alternative technique for determining the optimal number of clusters for a given dataset. This involves calculating the cost function for a given value of K starting with k = 2 and increasing K by 1 until the cost function of K drops drastically [18]. The cost function can be defined as the sum of squared errors, that is, the sum of the squared linear distance of each point to its nearest centroid. The reason the we see the cost of K decrease as K increases is that, as the number of centroids increases, the distances from each point to its nearest centroid decreases.



In both instances shown above it is obvious that the optimal number of clusters for the dataset analyzed is two, this is because the largest drop observed in the Cost Function of K (y-axis) occurs between 1 cluster and 2 clusters.

### Mean Shift

The Mean Shift algorithm is a form of hierarchical clustering algorithm. It works by iteratively shifting each data-point towards its nearest peak. These peaks are identified using a probability density function calculated from the given dataset. To start, each point is identified as a peak. We need to provide a kernel bandwidth value which will indirectly determine how many clusters will end up with. The kernel bandwidth is a radial distance from each point. If other points fall within this radial distance then the mean value for all of these points is used as a peak. This process is then repeated iteratively until all points shift towards their nearest peak. [20] This forms clusters of data-points.

# **\*Explanation of which is going to be used and why**

# **System Requirements Specification**

## **System constraints**

As previously mentioned, this project is only concerned with WIN32 binary files and not with their 64 bit equivalents. The system also does not make any attempt to decrypt or unpack malware files. This is due to the large number of implementations of packing algorithms and encryption algorithms used by attackers. It is unrealistic to have perfect conditions for each of these implementations. The extracted feature of file entropy will help combat this as this value gives a good indication of whether a file is malicious or not. Another possible approach to this problem is to attempt to unpack or decrypt the malware using existing tools before analysis with our solution.

## **Functional Requirements**

1. Provide methods for the extraction of all features outlined in the COFF specification [11]
2. Extract chosen features from a directory of WIN32 binary files
3. Provide a percentage measure for how accurate the clustering was.
4. Display the output of clustering algorithms in 2d and 3d graphs

## **Non-functional requirements**

If this project is a success it should:

1. Be able to distinguish between benign and malicious files with greater than 95% accuracy.
2. Be able to distinguish between different malware families with greater than 80% accuracy.
3. Provide an easy to use tool for static analysis of a PE32 file. i.e small setup time and cost with minimal configuration required to run.
4. Be platform independent for increased usability.
5. Handle errors gracefully and not crash
6. Be scalable to large datasets (1000+ files)
7. Data-mine a file in no more than 5 seconds

## **User Interface requirements**

The user interface for this tool should:

1. Allow the user to browse to a chosen directory
2. Display this directory for reference
3. Have a button for performing analysis on the specified directory.
4. Have tabs which switch between 2d and 3d plots
5. Allow the user to specify which clustering algorithms and dimensionality reduction algorithms to use
6. Show output from each of the selected algorithms
7. Show a percentage accuracy value for each of the selected algorithms

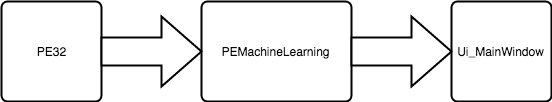
## **User Characteristics**

This tool is aimed at Security Operation Centres as well as InfoSec researchers in a professional environment.

Provides ability to compare an unknown file against a known dataset to determine its relative risk

# **Design**

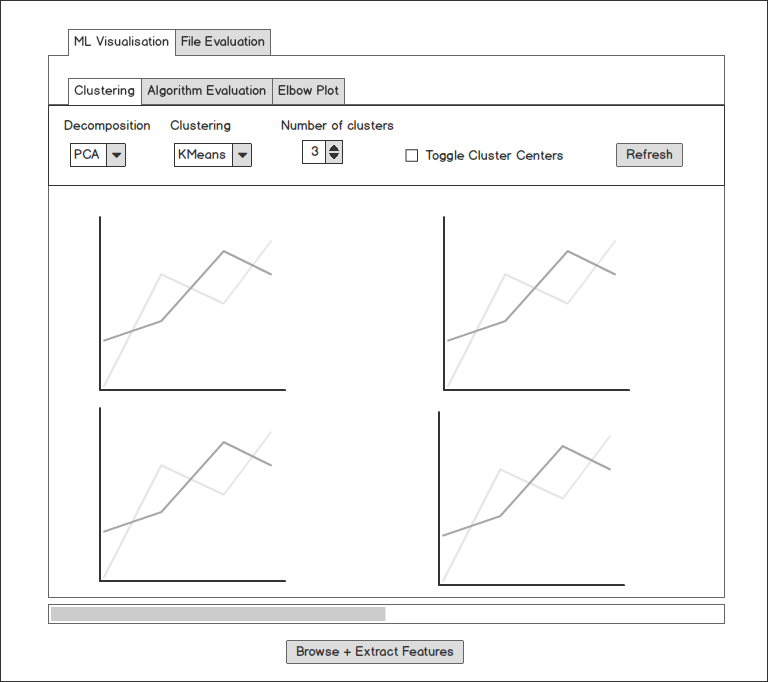
## **System Architecture**



The system consists of 3 classes. The PE32 class performs IO operations to read hexadecimal byte strings from Win32 PE Files. The PEMachineLearning class takes the data exposed by the PE32 class and performs dimensionality reduction, clustering and visualization operations. This class also handles interaction logic for the UI, attaching event listeners to UI components present in the UI\_MainWindow. The outputs from the operations carried out by the PEMachineLearning class are displayed through the UI\_MainWindow class.

## **User Interface Design**

The user interface is split up into 2 main tabs and 3 sub-tabs within the first main tab. Outside of the main area of the application there is a progress bar and a button which allows the user to browse to a directory.

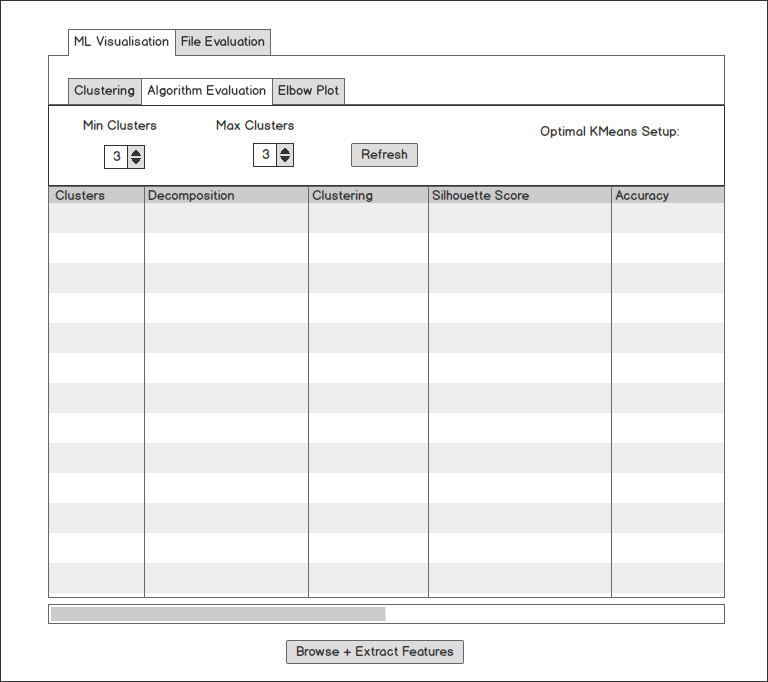


The first main tab is the visualization tab, this contains three sub-tabs for generating a number of different graphs and evaluating different combinations of decomposition and clustering algorithms on a dataset. The first sub-tab allows the user to choose a decomposition and clustering algorithm, as well as a number of clusters, which is only used for the KMeans algorithm. The user can toggle the display of cluster centers or centroids on the graphs plotted in this tab.

The four graphs are, starting from top-left clockwise :

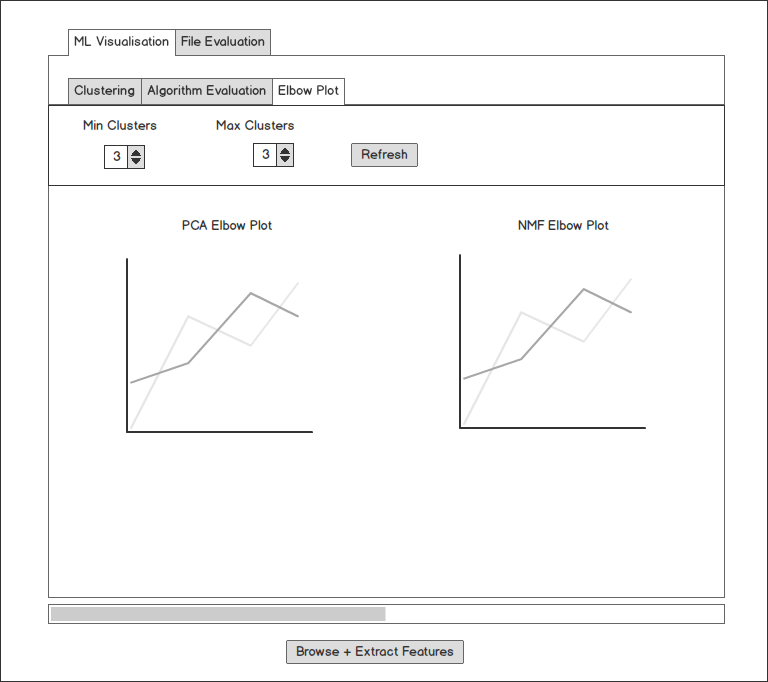
1. Decomposition Plot
2. Clustering Plot
3. Color Map Plot showing linear discriminants between clusters
4. Silhouette Plot depicting how well clustered the data is

When the plotting is complete a silhouette score will be made visible below the plots.



The seconds sub-tab consists of controls for specifying maximum and minimum numbers of clusters to evaluate algorithms between. This tab allows the user to quickly specify constraints for automatic evaluation of all possible decomposition and clustering algorithm combinations, and species the optimal KMeans parameters which can be used on the visualization page. The evaluation is presented as a table consisting of 5 columns:

1. Number of clusters
2. Decomposition Algorithm
3. Clustering Algorithm
4. Silhouette Score
5. Detection Accuracy



The third sub-tab provides the user with elbow plots for the KMeans algorithm for both supported decomposition algorithms. This can be used as an alternative method for determining the optimum KMeans parameters for the first tab.

**FILE EVAL**

## **Software Design**

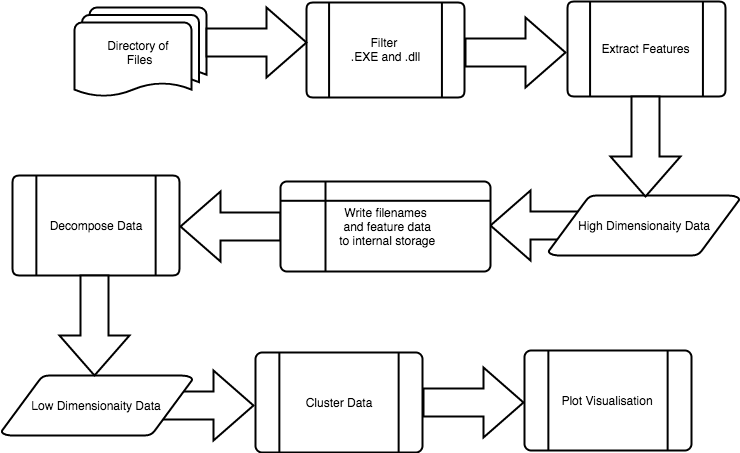


Figure !!! shows a high level version of the data-flow within the completed project. Each stage of this pipeline is explained in more detail below.

1. User browses to a directory of files. For each EXE or .dll file present in the directory, the tool creates an instance of the PE32 class.
2. For each PE32 object, extract our chosen machine learning features to a row in a CSV file.

|  |  |
| --- | --- |
| **Feature** | **Source** |
| File Entropy | PEFILE.GetFileEntropy() |
| Code/ Data Ratio | PEFILE.GetCodeDataRatio() |
| Major Image Version | PEFILE.GetMajorImageVersion() |
| Number Of Sections | PEFILE.NumberOfSections() |
| CommonDll Imports | PEFILE.GetImports() - map this list of imports to an array of true or false values for the presence of our chosen dlls. |

1. Perform decomposition on the high dimensionality data in the CSV file and project to a lower dimensional subspace suitable for plotting. Store this in a Numpy array.
2. Cluster data from array using chosen clustering algorithm and plot data points on a 2d Matplotlib scatter graph.



Example K=2 KMeans clustering for malicious (red) and benign(green) files.

1. (Optional) Use labels assigned by clustering algorithm in conjunction with filenames to determine the accuracy of the clusxstering.

**Clustering accuracy = Total number of correctly clustered files X 100**

**Total number of files analysed**

# **Implementation**

## **Choice of languages/libraries**

This project was implemented in Python 3.5. The reasoning for this is that Python is ideal for rapid prototyping; often requiring significantly less code than other object-orientated languages such as Java and C++ due to its dynamic typing. There are of course trade-offs with this choice, namely performance in comparison to C++ and other low level languages.

Due to the specialised requirements of this project, I made use of several libraries to aide in the rapid development of a working piece of software.

For the machine learning component of the project, I opted to use scikit-learn. This is an open source machine learning library for python which has implementations of common clustering algorithms such as KMeans and MeanShift. There are also several implementations for dimensionality reduction/decomposition available, namely Principle Component Analysis (PCA), Non-Negative Matrix Factorisation (NMF) and Fast Independent Component Analysis (FastICA). In my solution I make use of both PCA and NMF. To cluster the decomposed data, I made use of the KMeans and MeanShift algorithms.

I have chosen to use Matplotlib for data visualisation as it provides the ability to plot graphs in 2d and 3d with relatively little code. The data-mined features initially be too high-dimensional to plot on a graph, so we use PCA and NMF to project the data to a lower dimensional subspace which retains most of the variance from the initial dataset [15]. We then use the output of this as our input for the KMeans algorithm and then subsequently plot the clustered data using Matplotlib.

To implement a rudimentary GUI which displays the plotted graphs and evaluation of the clustering labels, I used PyQt which is a cross platform python binding used for rapid GUI development.

## **Development Environment**

Due to the malicious nature of the Windows PE files being analysed, my host operating system for my development environment was Mac OS X. This ensured that the malware files did not execute, as OS X uses the ELF format for executables, similar to a Linux environment.

I used Spyder as my Python IDE which is packaged with the Anaconda Python data-science platform. Anaconda comes with the Pylab package which includes Numpy and Matplotlib. My reasons for choosing the Anaconda Python platform were its ease of set-up and its ability to create multiple python environments with different versions of referenced libraries. It also comes bundled with scikit-learn and PyQt which minimises the time required to set-up a full development environment.

To create my GUI, I used the tools contained within QtCreator IDE. QtCreator is a cross platform C++ IDE which comes bundled with QtDesigner. QtDesigner is a WYSIWYG form design tool, which outputs a .ui file that can be used to generate a PyQt GUI class, using the pyuic5 command line tool. This solution to designing and creating a GUI minimised time spent creating complex layouts which was be better spent elsewhere in the development of the solution. It also allows for significantly more complex layouts than those possible with built in Python GUI libraries such as TKinter.

## **Key Implementation Decisions**

At the outset of the project it was decided that the implementation of machine learning algorithms and dimensionality reduction algorithms was out of scope of the project, subsequently, existing machine learning libraries were examined to determine their utility for what we wanted to accomplish in this project. The development efforts were largely focused in establishing a pipeline of operations which would give a useful visual representation of clusters within a given a dataset of benign and malicious files.

When the project was underway it was apparent that evaluation and comparison of multiple algorithms proved to be time consuming and that creating a tool based on a single algorithm was too rigid to be useful for any other experimentation with the dataset. As a result, elements were added to the UI to enable the tweaking of input parameters to the algorithms and also selection of which dimensionality reduction or clustering algorithms to use in the ****pipeline. Screenshots of these UI elements can be seen below.

To tackle the issue of algorithm evaluation I implemented a way of automatically evaluating each permutation of decomposition and clustering algorithms, within certain contraints set by the user similar to the tab in the previous screenshot. This tab finds the optimal setup for the KMeans algorithm based on the silhouette score, expressing how well the data fits the cluster labels generated for that particular permutation. This recommendation can be seen in the upper right-hand corner of the below screenshot.



This window allows the user to quickly evaluate which algorithm combinations produce the best clustering and, optionally, can also be used to determine how well the data has been clustered based on the detection rate of malware samples. The detection rate will only be displayed if the data-mined dataset is labeled, with the prefix of MAL for malicious files, and BEN for benign files, otherwise it will have a value of zero. It should be noted that because of how the filenames and extracted features are stored separately, the naming of a file has no effect on how the clustering algorithms determine which cluster is most appropriate for a particular file.

## **Mention Meanshift**

## **Implementation of Important Functions/Algorithms**

### Calculation of Entropy



Calculation of entropy is done by first initializing an empty integer array of length 256, with zeroes. The CalculateEntropy method allows for the entropy of sub-sections within the file to be calculated as well as the entropy value for the entire file. This array will hold a count of how many occurrences a particular byte value has between the startOffset and startOffset+length, the index of the count in the array is also the decimal value for that byte, allowing for an efficient iteration of this array later. The byte\_occurences array is then looped over and any byte values which have not occurred are ignored. For all byte values which have occurred, we calculate the frequency and then set the entropy value equal to: previous entropy value - frequency \* the base 256 log of the frequency \* 8. This results in an entropy value between 0 and 8. A higher value here represents more random data in the file within the constraints set by the caller.

### Conversion of Byte Data to Decimal



As the data required for decomposition and clustering needed to be in a numerical format, it was important to be able to convert between the hexadecimal byte strings being parsed from the PE file and their equivalent decimal representations. The simple method above was created to prevent the duplication of this conversion operation, utilizing pythons built in ability to convert between numbers in different base representations.

### Reading Values from PE Files



Due to the large number of times the tool needs to read values from a file, it was very important to write these functions in the most reusable way possible. The ReadByte method acts as the smallest component within the PE32 class, every call to the class to get a value will result in one or more calls to the ReadByte method. The ReadBytes method shows this in practice, where it loops until it has read the specified number of bytes, appending each to a local variable to be returned to the caller. The ReadBigEndianBytes and ReadLittleEndianBytes methods add another abstraction layer over the ReadBytes method, with the only difference in the returned result being, ReadBigEndianBytes use of pythons slicing functionality to reverse to order of the characters in the returned byte string. It should also be noted that these two methods accept parent offsets as well as offsets which are relative to the parent offset, this was a necessary step as the Windows PE File Specification[11] does not guarantee a value can be found at an offset from the start of the file, only that it can be found in relation to a previously calculated offset, usually the COFF header, Optional header or PE header.



The above screen shot shows just how flexible the previously discussed methods for reading bytes from a PE executable are. The only differences between calls to these methods were the input parameters, which were sourced from the PE File Specification[11].

### Getting Names of .dll Imports from a PE File



The above method provides a dynamic and flexible way of reading the string representation of an import name from a PE File. The method is called using the offset at which the imports entry in the ImportTable of the PE File can be found. This table sets out the relative virtual address(RVA) where each import can be found within the file. The method first reads the relative virtual address value which is found between startOffset and offset. It then begins to loop starting from the RVA until it hits a null byte, which is represented by *b’00’* in python. During each iteration of the loop it keeps a count of how many non-null values it has encountered, which is then used to read the entire section as a hexadecimal string. The count of the non-null bytes is used as a variable length in this call so the method does not have any issues reading import names of different lengths. This hexadecimal string is then decoded using the *bytearray.decode()* method. This returns the string representation of the import name.



Figure !!! shows how iterative calls can be made to the GetImportName method to return every import in the order they appear in the file. As you can see the index of the iteration, *i*, is multiplied by a constant, in this case 20, which is how many bytes are between each RVA value. This results in successive calls to the method having the exact startOffset value required to begin retrieving the bytes which make up the hexadecimal representation of the import name.

### Automation of Optimizing KMeans Parameters



As previously shown in Figure !!!!, the finished tool is able to provide a recommendation of the best KMeans parameters. The implementation of this functionality is relatively simple. A small part of the *populateTable()* method from the peMachineLearning class can be seen in Figure !!!. This code simply stores the best silhouette score and the decomposition algorithm, used to pre-process the data before performing the KMeans clustering, as two variables which are then used to updated the label shown in the top right corner of Figure !!!. This logic is carried out as the different permutations algorithms used are iteratively evaluated for the purposes of populating the table in the Algorithm Evaluation tab shown in Figure !!!

### Accuracy Check

****

The calculateAccuracy method takes a list of labels and a list of filenames as parameters in order to determine if the files were labeled correctly. The first step in calculating the accuracy is establishin which labels should be used for the comparison. For this reason, the method checks the first 3 letters of the first filename and sets the variables used for comparison accordingly. Once it has established which labels are the correct labels, it loops through all of the filenames and keeps a count of how many match the expected labels. This count is then returned as a percentage of the total number of files.

Decomposition and Clustering Plots

Color Plots

Silhouette Plots

## **Implementation of individual components**

### PE File Parser

The PE file parser component of this project has been implemented as a single wrapper class, PE32, which exposes properties of the PE File it is instantiated with. A full UML diagram of this class can be found in Appendix 2. For outputting these properties of a file I have also created a command line tool which, when given a path to an executable file, will print all fields which the wrapper class exposes for examination. The following command will run this tool ***python peparse.py {path-to-file}/{file-name}.exe****.* Appendix 3 contains a sample output from this tool.

### Feature Extraction

Feature extraction is accomplished by instantiating the PE32 class with all acceptable files in a given directory. The tool will only ever attempt to extract features from .EXE and .dll files as these are the only files which it was designed for. Once the PE32 class has been instantiated, calls are made to the methods on the PE32 class to get the values from the file. These values are then stored in a csv file, features.csv; The associated filenames are stored in a separate csv file, files.csv. The justification for storing the filenames separately, is that in later stages of the pipeline the finished tool uses, it is imperative that the data being input into the algorithms is numerical. When the filenames are required we only read from the files.csv and when we need data for our machine learning algorithms we only read from features.csv, it should be noted that the indexes are consistent between the files so the first piece of data in one file will correlate with the first line of data in the other etc.

### User Interface

The user interface for the finished tool is implemented in two classes, UI\_MainWindow and PeMachineLearning. The UI\_MainWindow acts as a superclass for PeMachineLearning, with PeMachineLearning inheriting all properties of the superclass. UI\_MainWindow acts as the setup for the UI and cannot be instantiated without being inherited, it handles the layout, naming and most of the display details of the UI components used. The PeMachineLearning class handles more specialized operations such as plotting graphs and performing IO operations, it uses its relationship with the superclass to display the results of these operations on the UI.

### Clustering and Categorisation

Clustering and categorisation are handled by the PeMachineLearning class. The methods have been designed so that they are flexible and reusable, accepting parameters for tweaking their behaviors and outputs, where appropriate.

**Expand**

**File Evaluation**

# **Testing**

## **Testing approach**

**Numerical difficulties**

# **System Evaluation and Experimental Results**

# **Conclusion**

# **References**

[1] Statista, “IoT: number of connected devices worldwide 2012-2020 | Statista,” 2015. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. [Accessed: 14-Apr-2017].

[2] D. Peraković, M. Periša, and I. Cvitić, “Analysis of the IoT impact on volume of DDoS attacks,” 2015.

[3] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, “Semantics-aware malware detection,” in *2005 IEEE SYMPOSIUM ON SECURITY AND PRIVACY, PROCEEDINGS*, 2005, pp. 32–46.

[4] M. Egele, T. Scholte, E. Kirda, and S. Barbara, “A survey on automated dynamic malware analysis techniques and tools,” *ACM Comput. Surv.*, vol. V, no. 2, pp. 1–49, 2011.

[5] B. Lau and V. Svajcer, “Measuring virtual machine detection in malware using DSD tracer,” *J Comput Virol*, vol. 6, pp. 181–195, 2010.

[6] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, “Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware,” in *Proceedings of the Seventh European Workshop on System Security*, 2014, p. 5:1--5:6.

[7] S. Gadhiya, K. Bhavsar, and P. D. Student, “Techniques for Malware Analysis,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 4, pp. 2277–128, 2013.

[8] M. Zubair Shafiq, S. Momina Tabish, F. Mirza, and M. Farooq, “‘PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime’ in Recent Advances in Intrusion Detection,” Springer Science + Business Media, 2009, pp. 121–141.

[9] M. Christodorescu and S. Jha, “Static analysis of executables to detect malicious patterns,” *SSYM’03 Proc. 12th Conf. USENIX Secur. Symp.*, vol. 12, pp. 12–12, 2003.

[10] A. Moser, C. Kruegel, and E. Kirda, “Limits of Static Analysis for Malware Detections,” *Acsac*, pp. 421–430, 2007.

[11] “Microsoft Portable Executable and Common Object File Format Specification,” 2015. [Online]. Available: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff\_v83.docx.

[12] R. Lyda and J. Hamrock, “Using Entropy Analysis to Find Encrypted and Packed Malware,” *IEEE Secur. Priv.*, vol. 5, pp. 40–45, 2007.

[13] K. Raman, “Selecting Features to Classify Malware,” *InfoSec Southwest 2012*, pp. 1–5, 2012.

[14] J. Yonts, “Attributes of Malicious Files,” 2012. [Online]. Available: https://uk.sans.org/reading-room/whitepapers/malicious/attributes-malicious-files-33979. [Accessed: 08-Dec-2016].

[15] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press, 2012.

[16] S. Wold, K. H. Esbensen, and P. Geladi, “Principal Component Analysis,” *Chemom. Intell. Lab. Syst.*, vol. 7439, no. August, pp. 37–52, 1987.

[17] D. Lee and H. Seung, “Algorithms for non-negative matrix factorization,” *Adv. Neural Inf. Process. Syst.*, no. 1, pp. 556–562, 2001.

[18] T. M. Kodinariya and P. R. Makwana, “Review on determining number of Cluster in K-Means Clustering,” *Int. J. Adv. Res. Comput. Sci. Manag. Stud.*, vol. 1, no. 6, pp. 2321–7782, 2013.

[19] J. A. Hartigan and M. A. Wong, “Algorithm AS 136: A K-Means Clustering Algorithm,” *Source J. R. Stat. Soc. Ser. C (Applied Stat.*, vol. 28, no. 1, pp. 100–108, 1979.

[20] Y. Cheng, “Mean Shift, Mode Seeking, and Clustering,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 17, no. 8, 1995.

# **Appendices**

## Appendix 1. Meeting Minutes





## Appendix 2. PE32 Class Full UML



## Appendix 3. peparse.py Sample Command Line Output

PE HEADER FEATURE EXTRACTION:

FILENAME: /Binaries/1.dll

FileSize: 163840 bytes

CodeDataRatio: 0.02631578947368421

\*\*\*FILE HEADER\*\*\*

Machine: 014c

NumberOfSections: 4

TimeDateStamp: 4d0e2fe6

PointerToSymbolTable: 00000000

NumberOfSymbols: 00000000

SizeOfOptionalHeader: 00e0

Characteristics: 210e

\*\*\*OPTIONAL HEADER STANDARD FIELDS\*\*\*

Magic: 010b

MajorLinkerVersion: 06

MinorLinkerVersion: 00

SizeOfCode: 00001000

SizeOfInitializedData: 00026000

SizeOfUninitializedData: 00000000

AddressOfEntryPoint: 000012fa

BaseOfCode: 00001000

BaseOfData: 00002000

\*\*\*OPTIONAL HEADER WINDOWS SPECIFIC\*\*\*

ImageBase: 10000000

SectionAlignment: 00001000

FileAlignment: 00001000

MajorOperatingSystemVersion: 0004

MinorOperatingSystemVersion: 0000

MajorImageVersion: 0000

MinorImageVersion: 0000

MajorSubsystemVersion: 0004

MinorSubsystemVersion: 0000

Win32VersionValue: 00000000

SizeOfImage: 00028000

SizeOfHeaders: 00001000

CheckSum: 00000000

Subsystem: 0002

DllCharacteristics: 0000

SizeOfStackReserve: 00100000

SizeOfStackCommit: 00001000

SizeOfHeapReserve: 00100000

SizeOfHeapCommit: 00001000

LoaderFlags: 00000000

NumberOfRvaAndSizes: 00000010

\*\*\*OPTIONAL HEADER DATA DICTIONARIES\*\*\*

ExportTableRVA: 000021b0

ExportTableSize: 00023e16

ImportTableRVA: 8284

ImportTableSize: 00000050

ResourceTableRVA: 00000000

ResourceTableSize: 00000000

ExceptionTableRVA: 00000000

ExceptionTableSize: 00000000

CertificateTableRVA: 00000000

CertificateTableSize: 00000000

BaseRelocationTableRVA: 00027000

BaseRelocationTableSize: 00000064

DebugRVA: 00000000

DebugSize: 00000000

ArchitectureRVA: 00000000

ArchitectureSize: 00000000

GlobalPtrRVA: 00000000

GlobalPtrSize: 00000000

TLSTableRVA: 00000000

TLSTableSize: 00000000

LoadConfigTableRVA: 00000000

LoadConfigTableSize: 00000000

BoundImportRVA: 00000000

BoundImportSize: 00000000

IATRVA: 00002000

IATSize: 0000005c

DelayImportDescriptorRVA: 00000000

DelayImportDescriptorSize: 00000000

CLRRuntimeHeaderRVA: 00000000

CLRRuntimeHeaderSize: 00000000

\*\*\*SECTION HEADERS\*\*\*

SECTION 0

Section 0 Name: .text

Section 0 VirtualSize: 0000039e

Section 0 VirtualAddress: 00001000

Section 0 SizeOfRawData: 00001000

Section 0 PointerToRawData: 00001000

Section 0 PointerToRelocations: 00000000

Section 0 PointerToLinenumbers: 00000000

Section 0 NumberOfRelocations: 0000

Section 0 NumberOfLinenumbers: 0000

Section 0 Characteristics: 60000020

Section 0 Entropy: 0.2375355877988614

Section 0 Md5: df7870bf6248d38832a649bc47dc28b1

SECTION 1

Section 1 Name: .rdata

Section 1 VirtualSize: 00023fc6

Section 1 VirtualAddress: 00002000

Section 1 SizeOfRawData: 00024000

Section 1 PointerToRawData: 00002000

Section 1 PointerToRelocations: 00000000

Section 1 PointerToLinenumbers: 00000000

Section 1 NumberOfRelocations: 0000

Section 1 NumberOfLinenumbers: 0000

Section 1 Characteristics: 40000040

Section 1 Entropy: 0.0032470259978673757

Section 1 Md5: 79bafdde51876f3e85f20d0596df7504

SECTION 2

Section 2 Name: .data

Section 2 VirtualSize: 0000006c

Section 2 VirtualAddress: 00026000

Section 2 SizeOfRawData: 00001000

Section 2 PointerToRawData: 00026000

Section 2 PointerToRelocations: 00000000

Section 2 PointerToLinenumbers: 00000000

Section 2 NumberOfRelocations: 0000

Section 2 NumberOfLinenumbers: 0000

Section 2 Characteristics: c0000040

Section 2 Entropy: 0.013294945794440415

Section 2 Md5: a505ae879dbdea80cb9c759f2cf47010

SECTION 3

Section 3 Name: .reloc

Section 3 VirtualSize: 00000204

Section 3 VirtualAddress: 00027000

Section 3 SizeOfRawData: 00001000

Section 3 PointerToRawData: 00027000

Section 3 PointerToRelocations: 00000000

Section 3 PointerToLinenumbers: 00000000

Section 3 NumberOfRelocations: 0000

Section 3 NumberOfLinenumbers: 0000

Section 3 Characteristics: 42000040

Section 3 Entropy: 0.03251353209648683

Section 3 Md5: 887c282d5df814c8d3f28966548c4ff0

\*\*\*IMPORTS\*\*\*

IMPORTING: KERNEL32.dll

IMPORTING: WS2\_32.dll

IMPORTING: MSVCRT.dll