# Malware Detection

# 19 Amaranath Balaji Prithivirajan

## Introduction

## Overview:

Malware is a term that is formed by the combination of the word’s malicious software. These are computer programs that are written with specific intent tailored to infect personal or industrial computers/networks to compromise their securities and access the data. The malware industry has grown more sophisticated in the recent years, with well-funded and organized attacks increasing in number year by year. These attacks have wreaked havoc and damaged several important real-world systems. The number of detected malware variants rose by 62% in 2020(Symantec), with yearly rise in malware attacks more intelligent techniques must be developed to tackle this issue.

## Background and Motivation:

There are real-time malware detection products present in around 160M computers worldwide and inspect 600M computers monthly. The data generated from these computers could be post about tens of millions of daily data points which could be post as a potential dataset that could be further analyzed as a machine learning problem.

One of the main reasons that malwares in these high volumes of different files evade detection is that the malware authors introduce polymorphism to malicious components. This this means malware belonging to the same family are modified and/or obfuscated so that they look like many different files.

## Goal:

To effectively analyze and classify large number of files we need to group them into their respective families, such grouping criteria could be used by computers to detect different signatures of malware coming from the same family and group them together.

# Description of Dataset:

The malware dataset is almost half a terabyte when uncompressed. It consists of a set of known malware files representing a mix of 9 different families. Each malware file has an identifier, a 20-character hash value uniquely identifying the file, and a class label, which is an integer representing one of the 9 family names to which the malware may belong. For each file, the raw data contains the hexadecimal representation of the file’s binary content, without the header. The dataset also includes a metadata manifest, which is a log containing various metadata information extracted from the binary, such as function calls, strings, etc. This was generated using the IDA disassembler tool. The original question posed to participants was to classify malware to one of the 9 classes. The dataset can be downloaded from the competition website.

[https://www.kaggle.com/c/malware-classification/data [1](https://www.kaggle.com/c/malware-classification/data%20%20%5b1)]

# Methodology

## Data Overview

* For every malware we have two files of the extension .asm and. bytes
* The total dataset consist of 200 GB of data out of which 40 GB of data is .bytes and 160 GB of data is .asm
* Lots of data for a single computer, which processed using a multi-processing was employed which will be explained in the sections below, using the multiple cores in a CPU. The entire pre-processing and analysis were done on an Intel(R) Xeon Platinum 8276 CPU @ 2.20 GHz (CascadeLake) with 56 CPUs
* There are 9 types of malware present in these files, the classes of which can be obtained from a separate class file also given in the dataset. The individual classes present are as follows:
  + Ramnit
  + Lollipop
  + Kelihos\_ver3
  + Vundo
  + Simda
  + Tracur
  + Kelihos\_ver1
  + Obfuscator.ACY
  + Gatak

## Problem mapping:

Dataset understanding is the first step in our analysis before pre-processing as the insights needed to capture features are gathered through the preliminary a lot of this insight is given in the research paper cited at [1]. The asm file contains assembly language code [2] and the bytes file contain byte code at the system level [3]. The general flow of a computer program is as shown in the below diagram. Diagram

Description automatically generated

The asm and bytes files are generated during the flow of compilation and execution of the computer program.

**A single byte files content looks like this:**

00401000 00 00 80 40 40 28 00 1C 02 42 00 C4 00 20 04 20

00401010 00 00 20 09 2A 02 00 00 00 00 8E 10 41 0A 21 01

00401020 40 00 02 01 00 90 21 00 32 40 00 1C 01 40 C8 18

00401030 40 82 02 63 20 00 00 09 10 01 02 21 00 82 00 04

00401040 82 20 08 83 00 08 00 00 00 00 02 00 60 80 10 80

00401050 18 00 00 20 A9 00 00 00 00 04 04 78 01 02 70 90

00401060 00 02 00 08 20 12 00 00 00 40 10 00 80 00 40 19

00401070 00 00 00 00 11 20 80 04 80 10 00 20 00 00 25 00

00401080 00 00 01 00 00 04 00 10 02 C1 80 80 00 20 20 00

00401090 08 A0 01 01 44 28 00 00 08 10 20 00 02 08 00 00

004010A0 00 40 00 00 00 34 40 40 00 04 00 08 80 08 00 08

004010B0 10 00 40 00 68 02 40 04 E1 00 28 14 00 08 20 0A

004010C0 06 01 02 00 40 00 00 00 00 00 00 20 00 02 00 04

004010D0 80 18 90 00 00 10 A0 00 45 09 00 10 04 40 44 82

004010E0 90 00 26 10 00 00 04 00 82 00 00 00 20 40 00 00

004010F0 B4 00 00 40 00 02 20 25 08 00 00 00 00 00 00 00

00401100 08 00 00 50 00 08 40 50 00 02 06 22 08 85 30 00

00401110 00 80 00 80 60 00 09 00 04 20 00 00 00 00 00 00

00401120 00 82 40 02 00 11 46 01 4A 01 8C 01 E6 00 86 10

00401130 4C 01 22 00 64 00 AE 01 EA 01 2A 11 E8 10 26 11

00401140 4E 11 8E 11 C2 00 6C 00 0C 11 60 01 CA 00 62 10

00401150 6C 01 A0 11 CE 10 2C 11 4E 10 8C 00 CE 01 AE 01

00401160 6C 10 6C 11 A2 01 AE 00 46 11 EE 10 22 00 A8 00

00401170 EC 01 08 11 A2 01 AE 10 6C 00 6E 00 AC 11 8C 00

00401180 EC 01 2A 10 2A 01 AE 00 40 00 C8 10 48 01 4E 11

00401190 0E 00 EC 11 24 10 4A 10 04 01 C8 11 E6 01 C2 00

**A single asm file’s content looks like this:**

.text:00401000 assume es:nothing, ss:nothing, ds:\_data, fs:nothing, gs:nothing

.text:00401000 56 push esi

.text:00401001 8D 44 24 08 lea eax, [esp+8]

.text:00401005 50 push eax

.text:00401006 8B F1 mov esi, ecx

.text:00401008 E8 1C 1B 00 00 call ??0exception@std@@QAE@ABQBD@Z ; std::exception::exception(char const \* const &)

.text:0040100D C7 06 08 BB 42 00 mov dword ptr [esi], offset off\_42BB08

.text:00401013 8B C6 mov eax, esi

.text:00401015 5E pop esi

.text:00401016 C2 04 00 retn 4

.text:00401016 ; ---------------------------------------------------------------------------

.text:00401019 CC CC CC CC CC CC CC align 10h

.text:00401020 C7 01 08 BB 42 00 mov dword ptr [ecx], offset off\_42BB08

.text:00401026 E9 26 1C 00 00 jmp sub\_402C51

.text:00401026 ; ---------------------------------------------------------------------------

.text:0040102B CC CC CC CC CC align 10h

.text:00401030 56 push esi

.text:00401031 8B F1 mov esi, ecx

.text:00401033 C7 06 08 BB 42 00 mov dword ptr [esi], offset off\_42BB08

.text:00401039 E8 13 1C 00 00 call sub\_402C51

.text:0040103E F6 44 24 08 01 test byte ptr [esp+8], 1

.text:00401043 74 09 jz short loc\_40104E

.text:00401045 56 push esi

.text:00401046 E8 6C 1E 00 00 call ??3@YAXPAX@Z ; operator delete(void \*)

.text:0040104B 83 C4 04 add esp, 4

.text:0040104E

.text:0040104E loc\_40104E: ; CODE XREF: .text:00401043j

.text:0040104E 8B C6 mov eax, esi

.text:00401050 5E pop esi

.text:00401051 C2 04 00 retn 4

.text:00401051 ; ---------------------------------------------------------------------------

This is a single datapoint for our analysis we used the byte files alone. There are 10868 files present in both the different formats. For this analysis we used only the byte files. The byte files contain header which is an address in the beginning and then hexadecimal representation of the binary content that is present in the file. Let us convert this to a bag of words representation and count the number of times a word here a hexadecimal value such as the E3 for example has occurred in the file. So, for each file we will count the frequency of the 255 different hexadecimal words occurring in the file. This gives us the Bag of Words representation. We will use this representation to compute a baseline random model for classification and then compare it with other different models to evaluate the performance. Most of the knowledge and techniques to pre-process and understand data have been obtained through the discussion forums in Kaggle with the help of the suggestions provided by domain experts. [5]

So, this is a multiclass classification problem that is going to use the data from bytes files present in the dataset as NLP data and the feature representation we are going to use is the Bag of Words representation.

## Data pre-processing:

The train folder of the initial dataset contains both the asm and byte files mixed together in a single folder we initially run a python script to separate this out into two different folders one called asm and the other called bytes. Now each of the two separate folders contains 10868 files. We will be doing our initial model building and feature extraction using only the bytes file. This is considering the time at hand and the dataset size at our disposal the preprocessing script for the bytes file is relatively simple in terms of both runtime and space complexity also the size of the total bytes files is about 40GB while in addition to complexities of the preprocessing script the entire size of the asm files is about 160GB and requires parallel processing of file read and write operations to complete in any reasonable amount of time.

## Bytes file text pre-processing:

Each line of the byte file starts with a header address[4]. This address is followed by a hexadecimal representation which contains a set of words. Each byte file contains lines similar to these that encompass the hexadecimal representation of the program that has been written.

A single hexadecimal number in the line, is a compact representation of the values which could range from 0-255.[3] Each of the bytes file contains a header in the starting of the first line and then there are hexadecimal representation of the contents of the program. The hexadecimal representations are then split into unigram representation where each word is split by the space. There are 255 unique possible combinations for each file.

For each line the initial address is removed, then each hexadecimal is considered as a single word and its corresponding frequency of occurrence is calculated for each byte file. This is done for all the different possible hexadecimal combinations. This is done for all the different files present in the bytes folder. The simple approach here is to run through the files using some tokenizer like the CountVectorizer[6] available in sci-kit learn. However, this approach is not feasible since it requires the iterable to be loaded onto the memory before it could preprocess. Moreover, a simple script that gets the Bag of words feature representation can be easily built out for our dataset.

This script generated a csv file which contains each observation as a file the column is named as Id then the corresponding count representations for the different hexadecimal words are arranged together as the individual features. The class labels for the different files are loaded and a merge operation is then performed between these two datasets. The size of the different byte files is also calculated and added as an additional feature representation.

The we finally, normalize the count values present in the different feature columns so that there is a nice spread of the values between a range of 0 – 1.[7]

# Result and Analysis

## Class Distributions and Univariate Analysis:

The class data associated with each file is plotted in a histogram to understand the distribution of different classes in our dataset. Since, this is a multiclass classification problem an understanding whether the data is balanced or not would be helpful to further in our analysis. The histogram is plotted using seaborn with the percentage of each class present in the overall dataset.

Chart, bar chart, histogram

Description automatically generated

Some observations made from the above histogram were that the given problem is an imbalanced class problem. The number of data points present in some classes clearly dominate others. Here classes 2,3 and 1 clearly have higher number of points in the dataset than a class like 5 and certainly the other classes as well. But class 5 has comparatively a low number of points. To further our understanding of the data lets carry out a univariate analysis on the size feature of the dataset. The plot that is done is a box plot to show the size value distributions for different classes of the data present in our dataset.

Chart, box and whisker chart

Description automatically generated

It can be seen that although the number of data points for class 5 was low in the histogram their size is comparatively high compared to the other classes present in the dataset. While the size of the class 2 data points can be attributed to the number of the points present in the overall makeup of the dataset. The other sizes of the other classes are relatively smaller in comparison

## Multivariate Analysis:

This is an analysis that is carried on only on the hexadecimal words present in the merged final dataset. The class and the size of the file attribute features have been dropped before applying the algorithm for the analysis. Since this is a high dimensional analysis, we will be using the state-of-the-art technique t-distributed Stochastic Neighborhood Embedding technique (t-SNE). This is a dimensionality reduction technique that helps us go from high dimensional to lower dimensional data much like Principal Component Analysis but t-SNE helps us to understand the data better when visualized. As it should be done the analysis is carried out with different values of perplexities with high number of iterations to check until the points in our data converge. The t-SNE plots obtained for different values of perplexities are shown below.

**T-SNE Plot with perplexity value 50:**

Chart, scatter chart, bubble chart

Description automatically generated

**T-SNE Plot with perplexity value 30:**

**Chart, scatter chart

Description automatically generated**

The following inferences can be made from the t-SNE plots above. While the inferences were made with the following considerations in mind, the information about the sparsity/density of points in a cluster should not be read from the t-SNE plots. The distance between the different clusters may not be actually representative because of the nature of t-SNE plots. This is caused by the use of the t-distribution, in t-SNE which has been an improvement over SNE plots to solve the ‘crowding’ problem [8].

There are clear groupings of certain clusters that could be visibly seen from the dataset. These are present around the periphery while there is a certain overlap of clusters of the different classes that could be visibly seen in the center, this gives us an overview of the possibility of how certain classes could be separated from each other while most have characteristics that overlap each other in this multiclass classification problem.

## Modeling:

For the purposes of modeling a baseline model is required to which the results of other models could be compared to. Similarly, a visual representation of the performance metric would be helpful we would use precision matrix for the purposes of monitoring the performance visually. All the models would be using multi-class log loss as their respective loss metrics. The data is split into 80 percent train and 20 percent test data and within this 80 percent train data we split it again into a 20 percent cross-validation data. So, the entire split is 64 percent train set, 16 percent cross validation set, 20 percent test set.

## Random Model:

A random model is simply put a set of random predictions of multi-class values of probabilities put together in a single vector. This is generated for each data point. This is generated by first assigning a random number to a vector of size 9. But the sum of all the probabilities in the vector should sum to one so we add up all the individual values in the vector and divide this value for each individual value in the vector. This gives us the individual class probabilities needed for each data point. We repeat the same for all the points in the dataset. Then we calculate the log loss for all the generated points and original vectors in the dataset. This gives us the baseline log loss for us to compare against the other models. We also calculate the baseline precision matrix and visualize so that this can be used as a baseline. The baseline log loss value for cross validation data is 2.487 and the loss value for test data is 2.498. The precision matrix is as shown below. The number of misclassified points is about 88.9 percent.

**Precision Matrix for Random Model:**

A screenshot of a computer

Description automatically generated with medium confidence

## K- Nearest Neighbors:

The k-nearest neighbors are the first model that is used to compare the performance the model after the random model. The hyper parameter that is in play is the k, which is the number of points that each data point should consider as its neighbor. There are number of values of this hyperparameter which is tried out in steps of 2 ranging from 1 through 15. The model is fit but for the multiclass log loss needs the values as probabilities hence we get probabilities of the class predictions given out by the models using the CalibratedclassifierCV [9] module present in scikit learn. We use the ‘sigmoid’ method for this calibrated classifier, which refers that we had used logistic regression. We then use this set of probability predictions to calculate the multiclass log loss The best log loss that could be obtained was for a value of 3 for k , the corresponding log loss obtained was 0.203. The corresponding train loss for the k value of 3 is 0.11 and the cross validation loss for the k value of 3 is 0.203 and test loss for k value of 3 is 0.205. The number of misclassified points is about 4.22 percent which is a significant improvement over our random model. The multiclass log loss for different values of the hyperparameter are calculated and then their plotted as shown below.

Chart, line chart

Description automatically generated

The precision matrix is also calculated and visualized below.

Graphical user interface, table

Description automatically generated

## Logistic Regression:

The logistic regression model is a model that is another model that is being experimented for the classification problem from a geometric perspective. This model uses only the L2 regularization technique or the otherwise called the ridge regression technique. Different values for the regularization is tried out for the L2 regularization. Ranging from 0.00001 to 10000 in steps of powers of 10 from -5 to 4. Then the class output of the logistic regression model is then fed to the calibrated classifier for calibration of class values to probability values as explained in the previous k-NN algorithm. This will help us get the multi-class probabilities vector for the different classes, this is done for all the data points present in the dataset. We then calculate the multiclass log loss. This procedure is done for the different values of the L2 regularization the sci-kit learn module has parameter named C which is the inverse of the regularization strength for the different values mentioned above. The plots are plotted for the different C values of the model after calibration and calculation of their respective Multiclass log errors. The best value for C that is the inverse of regularization constant value that could be obtained is 1000, the corresponding log loss is about 0.884, this gives the cross-validation error. The training error for the log loss is about 0.858. The test error is about 0.852 for the c value of 1000. The number of misclassified points is about 27.27 percent which has increased. The plot is as shown below

Chart

Description automatically generated

The precision matrix is also calculated and visualized below.

Application

Description automatically generated with low confidence

## 

## Random Forest:

Random Forest is the model that is tried from the standpoint of being an algorithm from the ensemble technique. Random Forest is falls under one of folds of ensemble techniques called Bagging which is short for Bootstrap sampling and aggregation. The common aggregation technique that is used would be majority vote and usually the models would be any one which consist of low bias and high variance as the random forest models by default can help reduce the variance present in the models. Here the hyperparameter of the model would be the number of estimators which is the number of models used in the technique so this would contain a total of seven values ranging from 10 to 3000. Random forest model also does not give out the vector of class probabilities as needed by the multi class log loss hence it is required to do model calibration. This is done using the Calibrated classifier module available in sci-kit learn we use the logistic regression technique for this module. Thus, the predictions for different hyperparameter values are calculated and their respective log losses are calculated after calibration. The hyperparameter that gives us the best estimate is the number of base estimators is of size 50, the corresponding cross validation error is 0.0881. The train error for the same hyperparameter value is 0.026 and the test log loss is 0.087. It could be observed that the model has slightly overfit during training. The number of misclassified points is reduced to 2.023 percent. The graph is as shown below.

Chart

Description automatically generated

Then the precision matrix is also calculated and visualized as shown below.

Graphical user interface, application, table

Description automatically generated with medium confidence

## 

## XG Boost:

XG Boost is another model that is tried in the ensemble models category. This comes under the ensemble technique called Boosting. Although there are a number of other hyper parameters that could be potentially tuned we tune only the number of base estimators. Here the hyperparameter of the model would be the number of estimators which is the number of models used in the technique so this would contain a total of seven values ranging from 10 to 2000. To calculate multi-class log loss, it is required XG Boost to gives out a vector of class probabilities. So, model calibration is needed we calibrate the output of the model using the calibrated classifier CV module available in sci-kit learn. This does the calibration of the output to the respective class probabilities in a vector, for each data point. We do this for the different values of the base estimators and after calibration we calculate the multiclass log loss. The best value of the estimators is 2000 the corresponding log loss value is 0.0804 during cross validation. The train error for the same hyperparameter is 0.022 and the test error is 0.0828.The number of misclassified points is about 1.51 percent. Then the multi-class log loss for different values of base estimators is plotted as shown below again the log loss error.

Chart

Description automatically generated with medium confidence

The precision matrix is also calculated and visualized as shown below.

Graphical user interface, application

Description automatically generated

## XG Boost with parameters obtained through Random Search:

The last modeling paradigm is repeated but this time the parameters are obtained by performing a random search. The different parameters searched are learning rate, number of estimators, maximum depth of a tree, sub sample ratio of column when constructing each tree and sub sample ratio of training instance. A list of values for each of these parameters are passed along with model obtained from the last step before calibration and is passed to the randomized search module available in the scikit learn. The best parameters for the model is obtained through random search and then multiclass log loss is calculate with this model for the training, cross validation and test data. The best parameters obtained are 0.5 for subsample, 100 for the n\_estimators, 5 for max\_depth, 0.15 for learning\_rate and 0.5 for colsample\_by\_tree. The corresponding loss values are given below the train loss is 0.0215, cross validation loss is 0.0806 and test loss is 0.0789.

## Future Work:

## Asm file pre-processing:

The asm files give to us require domain knowledge to convert them to a meaningful dataset. Also, the size of the entire asm files present in the dataset is close to 160 GB and hence processing them without multi-processing take an irrational amount of time to complete. Hence a deeper dive to obtain the knowledge about the different components that make up the asm files was carried out [10]. The different components that make up the asm file is as follows headers, opcodes, keywords and registers. We have the asm files for x86 architecture CPUs and this gives specific values these different categories can take [11]. The overall idea is to get a bag of words similar to the previous case of byte files but using the values of the four different categories present in the asmFiles.

## Parallel processing:

The next part is to generate the bag of words for the given data as efficiently as possible. This would entail multi-processing code. The data is split into folders of twenty different chunks each averaging a size of 6.8 GB. Then the asm file in each of these folders is parallel processed using one processor each on a server. Each of these processes generate a csv file that contains the corresponding output that is dumped in an output folder. These csv files are then merged together to get a single output csv file. The multiprocessing is done using the multiprocessing library in python. The data pre-processing code is attached along with this.

Chart, box and whisker chart

Description automatically generated

## Conclusion:

We could build more effective models and gather more useful insights by performing analysis on the asmFiles and building more effective model with the pre-processed asm files data. The entire training process could be greatly accelerated with the use of MPI and MPI enabled libraries like the daal4py which could be explored in the future. These libraries enable parallel processing during model training. The current model training or parameter search for the asm file takes a significant amount of time which will be a bottle neck during consecutive iteration.

# References:

1. https://arxiv.org/pdf/1802.10135.pdf
2. https://en.wikipedia.org/wiki/Assembly\_language
3. <https://en.wikipedia.org/wiki/Bytecode>.
4. https://www.jrebel.com/blog/java-bytecode-tutorial
5. <https://www.kaggle.com/c/malware-classification/discussion>
6. <https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html>
7. <https://stackoverflow.com/questions/26414913/normalize-columns-of-pandas-data-frame/29651514#29651514>
8. <https://distill.pub/2016/misread-tsne/>
9. <https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html>
10. <http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>
11. <https://en.wikipedia.org/wiki/X86_instruction_listings>
12. https://docs.python.org/3/library/codecs.html