Malware detection in portable executables using machine learning

Maj Amit Pathania(163054001) ,
Maj Manjunath Bilur(173054001)
Kapil Aggarwal(16305R010)

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

April 24, 2018

Contents

1 Abstract

Malware infections is one of most common security risks which can lead to loss and compromise of data and privacy and often leads to loss of money(Eg Ransomware). The Windows OS is one of most commonly used OS and also most suspectible to malware infections. In our project, we have surveyed and implemented machine learning based technique to classify file as benign or malicious with high accuracy and low computational power. Out of 56 raw features in our dataset, after modification, we have selected only 28 features. We applied various Machine learning algorithms and achieved accuracy upto 99%. Additionally, we analysed false positives and false negatives to understand how file header values control maliciousness of the file. Also, we tried to modify PE header values of few malicious files to check our classifier accuracy.

2 Introduction

Malware is defined as software or piece of code which can infiltrate or damage a system without the owner's consent. It is becoming increasingly difficult to detect malware using old signature based methods as most of current malwares are either polymorphic wherein each copy of virus uses a different key or are metamorphic wherein each new version uses non cryptographic obfuscation and thereby making it more difficult to get signature[2].

The basic idea of any machine learning task is to train the model based on some algorithm and perform classification or predict new values. Training is done on the input dataset and the model is built which is then subsequently used to make predictions.

Malware detection is classification problem. We can train a program to recognize whether a piece of software is a malware or not and thus we can then detect later that whether a given file is malware or not.

3 Motivation

Most of current malwares are difficult to detect using old **signature based methods** as they use various polymorphic or metamorphic approaches to generate new signatures and evade detection.Signature based detection has

high detection accuracy but is limited to the signature database, hence requires frequent updates.The major drawback is that it provides an attack window time (time between a vulnerability discovery to signature update or patching) equal to the sum of detection time, signature generation time and updating time.

Other approach is **behavior-based** also referred to as heuristics-based analysis. In this method, the actual behavior of malware is observed during its execution, looking for the signs of malicious behavior like modifying host files, registry keys, establishing suspicious connections. By itself, each of these actions cannot be a reasonable sign of malware, but their combination can raise the level of suspicion of the file. There is some threshold level of suspicion defined, and any malware exceeding this level raises an alert.Although this approach provides some level of effectiveness, it is not always accurate, since some features can have more weight than others.Non-signature based detection can raise false positive or false negative result.Non-signature based detection techniques can also detect unknown, zero-day and modern malware. To take these correlations into account and provide more accurate detection, machine learning methods can be used.

4 Our Contribution

The key identifier of malware is that it either changes the control flow or do change to the data structure. These changes effect memory access pattern by program. Our experimental evaluation focuses on Portable executable (PE) file's header fields values. We extracted all header data from PE data set. We categorized the weight of each feature so as to ensure that only important features that aid in our analysis are finally considered.The proposed learning model for malware detection has four major objectives as listed below:

- Feature set extraction from PE header.
- Select relevant headers and modify/derive few features thereby enhancing the detection capability of malware. In paper, they have considered only one classifier to find feature importance but we used two classifiers and combined their results to find relative importance.
- Application of machine learning techniques on the complete feature set and develop model for classification.

- Apply test data on the model and verify result.

Additionally, we analysed false positives and false negatives to understand file header values of malicious files and we also tried to modify PE header values of few malicious files to check our classifier accuracy.

Broadly, malware detection using machine learning involves following major steps:

- Training dataset is taken and a set of features are extracted based on training scores.
- The machine learning model is trained on this training dataset based on features selected in step1.
- The trained model is applied on test dataset and accuracy of model is calculated.

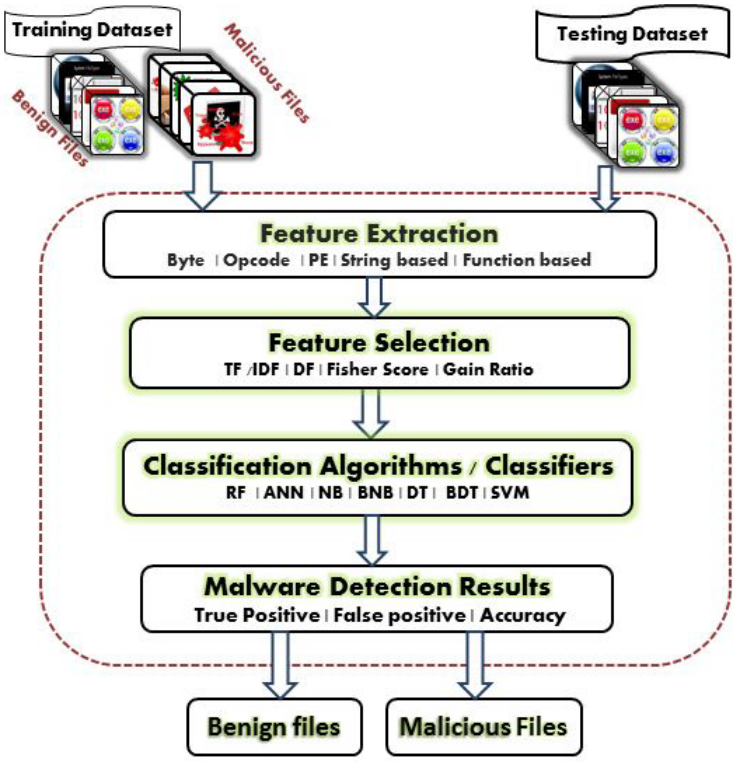

Figure 1: Framework for malware detection [3]

5 Objective

The objective of this project is to use machine learning to detect whether given file or program is malware or not based on PE file header. The aim is to implement and compare the performance of different machine learning algorithms (LDA,Random Forest, Neural network, adaboost and Decision tree).

6 Feature extraction

There are two common approaches to extract features from malware: static analysis and dynamic analysis. **Static analysis**[4] refers to extracting statistics from the meta-information of an executable without running the executable, such as a list of DLLs in the binary [5]. Features from static analysis are vulnerable to obfuscation techniques such as code transformation techniques, but have the advantage that malware never has to be executed. **Dynamic analysis**, on the other hand, runs an executable to extract features. Dynamic analysis, in principle,should be less vulnerable to obfuscation as it extracts features from the behavior of an executable[6]. Here, in our project, we are **using static analysis method by examining PE header information**.

6.1 PE File header

The Portable Executable (PE) format is a file format for executable, object code, DLLs, FON Font files, and others used in 32-bit and 64-bit versions of Windows operating systems. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code.This specification describes the structure of executable (image) files and object files under the Windows family of operating systems. PE executable contains two sections, which can be subdivided into several sections. One is Header and the other is Section.We can extract the different header's field values from the files using **pefile python module**. It is very efficient and well accepted Python module for PE file processing.

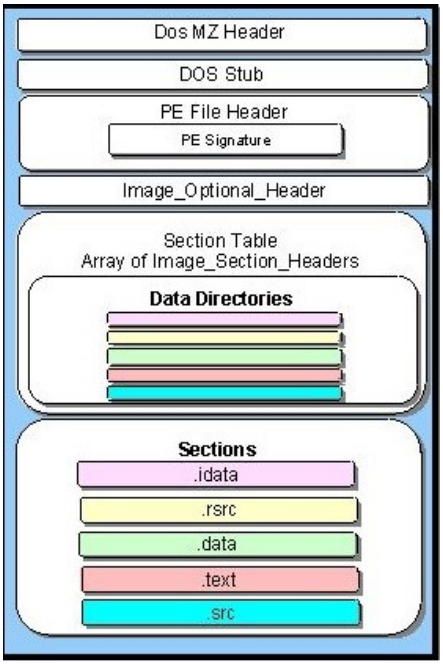

Figure 2: PE format

6.2 Feature selection

The dataset set consist of **138047** records.The dataset consists of **56** raw features out of which only **28** features were selected.Out of the selected **28** features, **5** were converted to Boolean and 1 was transformed to a value between 0 to 16.

6.2.1 Calculating relative feature importance

Generally, feature selection is performed before training and testing, and it reduces the feature set dimension as it selects only features with high discriminative value. We were interested in knowing the importance ranking of features in raw feature sets which further helps to understand the difference and relative importance of features. Among model based feature selection methods (wrapper methods) tree based methods are easier to apply and without much tuning, it can also model non-linear relations. We have used Random Forest Classifier with all other default settings (scikit-learn implementation) to get feature importance for raw feature set. It is meta es-

timator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.By default Gini impurity is used to measure the quality of a split.

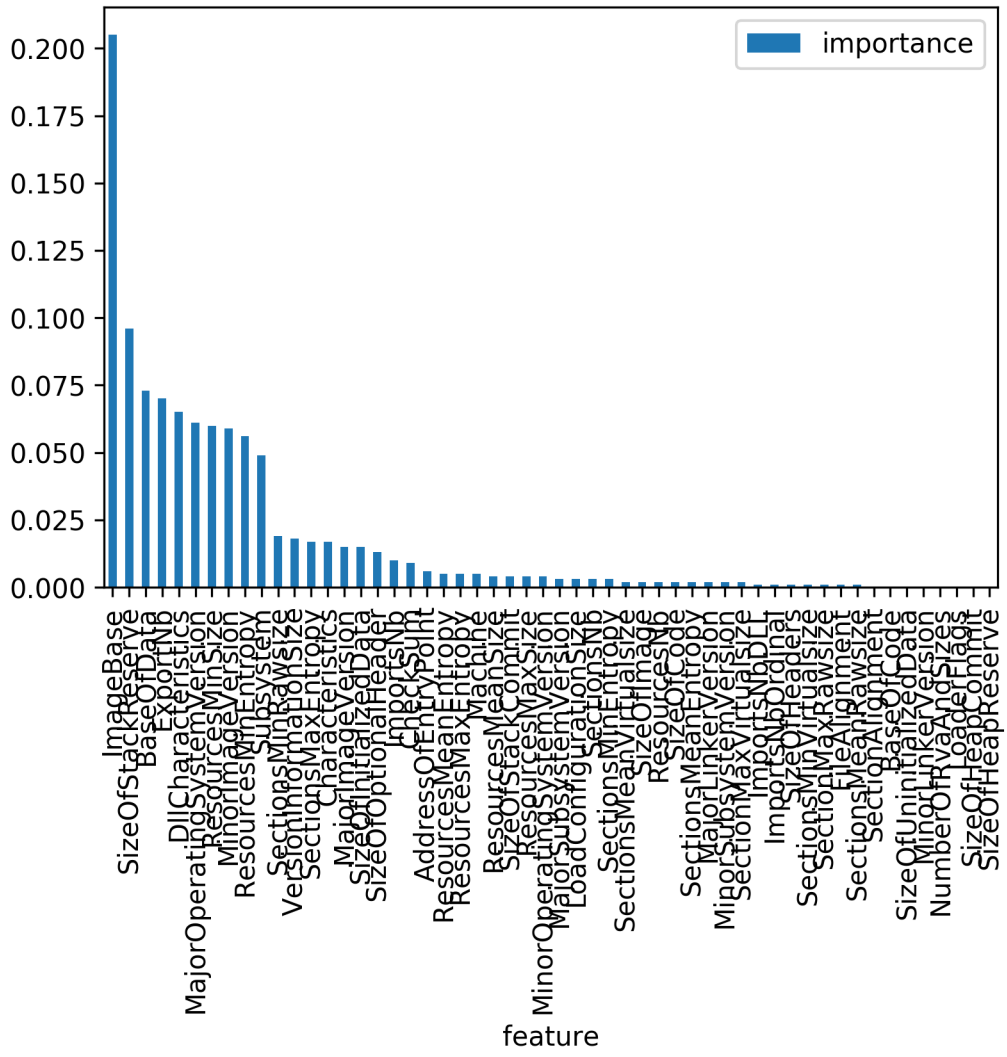

Figure 3: Feature importance using Random Forest

The top 15 features as given by Random Forest Classifier shown in figure below:

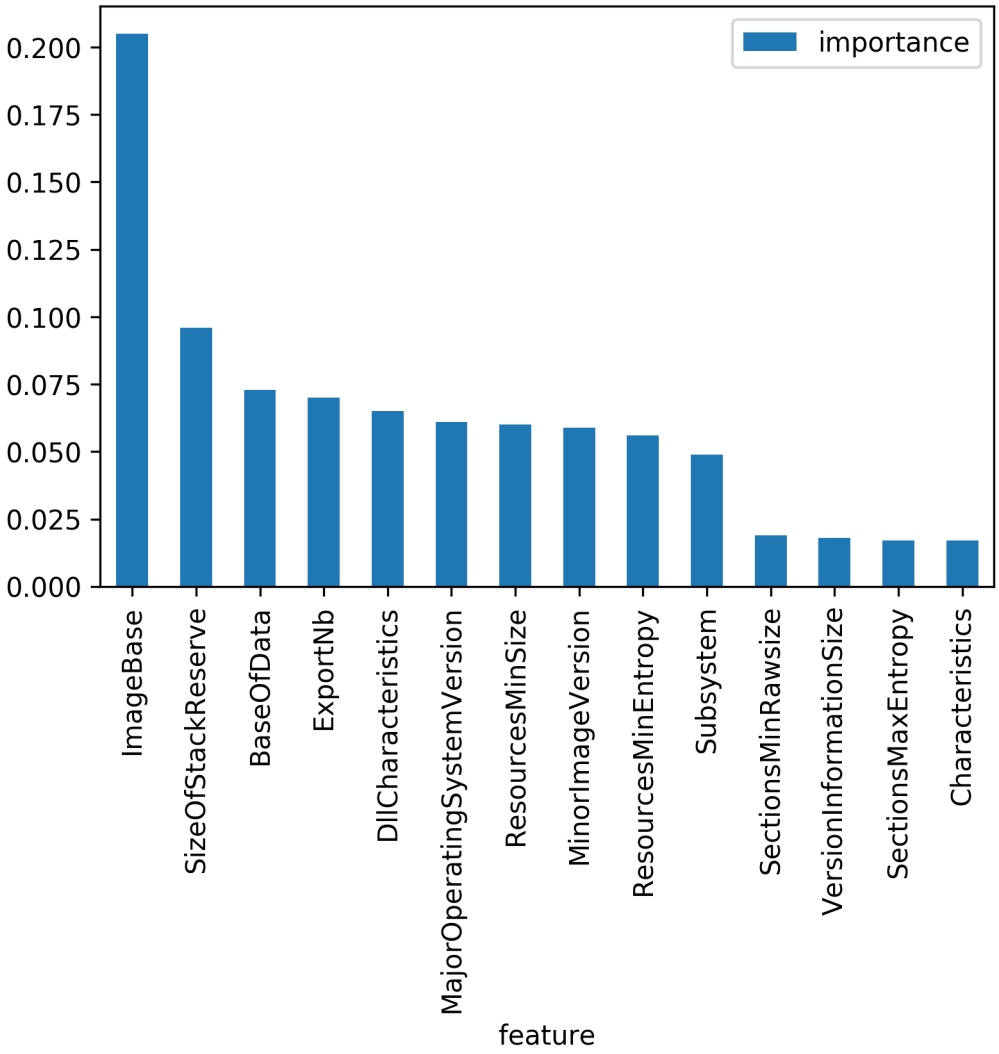

Figure 4: Top 15 features using Random Forest Classifier

We also tried Extra Trees Classifier with 250 trees and all other default settings (scikit-learn implementation) to get feature importance for original feature set. It is an ensemble classifier and by default Gini impurity is used to measure the quality of a split. The top 15 features as given by Extra trees Classifier shown in figure below:

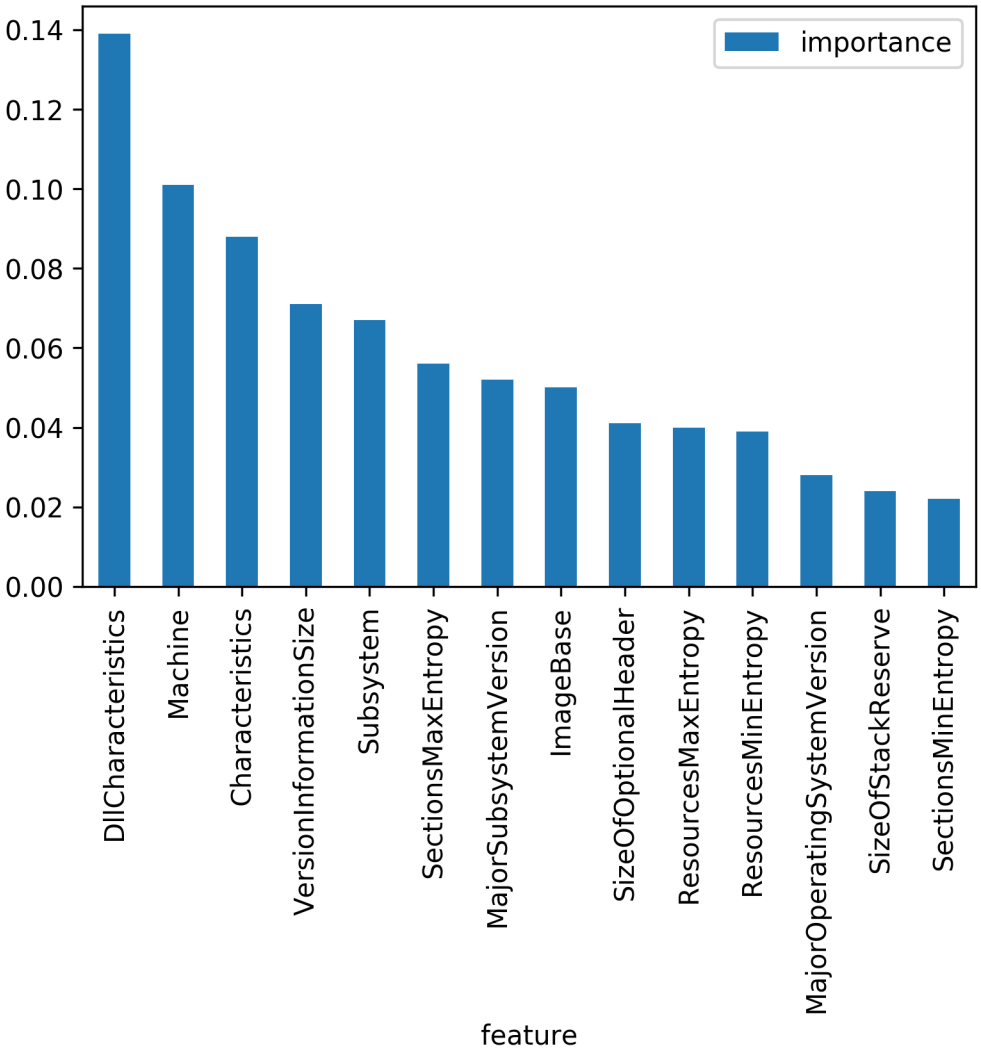

Figure 5: Top 15 features using Extra Trees Classifier

6.2.2 Feature analysis

- We combined the results obtained from Random forest and Extra Trees Classifier to select important features.
- Selected features like Characteristics , DLL Characteristics , Resources-MeanSize , ResourcesMeanEnropy as they are the property of each file

and describe the behaviour of the application.

- Feature like SizeOfInitializedData , SizeOfHeaders, SizeOfStackReserve , SizeOfHeapReserve describe the memory profile of an application.
- VersionInformationSize, Number of dll importes etc defines the others resources required

Reason for Dropping

- Features like Name, md5 , BaseOfData, check sum are dropped because these features are having different value for each record in dataset.
- Some of the features are having the values that are co-related like min , max and mean of the attributes so we drop those values which are low in relative importance as found out earlierResourcesMinSize, SectionMaxRawsize, etc.
- Some features are dropped because they were uniformly distributed among malicious and benign files like OS version , linker version etc and don't define the type of file.

6.2.3 Conversion of selected features[1]

- **SectionAlignment** is a field in the optional header and it is the alignment (in bytes) of sections that determine when they are loaded into memory. According to the Microsoft specification, it must be **greater than or equal to FileAlignment and the default is the page size of the architecture**. This field is treated as Boolean and is compared against the specifications. If extracted value follows the specification then this field will have a value 1 else will be assigned as value 0.
- The value of **FileAlignment** field of optional header is the factor (in bytes) that is used to align the raw data of sections in the image file. Microsoft specification states that this **value should be power of 2 (between 512 and 65536). Default value is 512** and if SectionAlignment is less than architecture's page size, then this value must match with SectionAlignment value. This field is converted to Boolean feature.

- **SizeOfImage** gives the size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a **multiple of SectionAlignment**.It is found that almost all benign samples follow the specification but approx 4% (3.87%) malware samples do not follow the specification. This field is also taken as Boolean feature.

- **SizeOfHeaders** has value that is equal to the combined size of MS-DOS stub, PE header and Section headers, and is **rounded to a multiple of FileAlignment**. This specification is also validated for malware and benign samples. Among benign samples, 16% did not follow the specification while 78% of malware samples did not follow the specification.

7 Performance evaluation

7.1 Dataset

In our project, we have used the dataset available online[7]. The collected dataset was divided into training and testing sets,containing 90% and 10% of the samples respectively in training and test set.The Training sample size is 124242and Test samples is 13805.

7.2 Comparing results obtained using different algorithms

Algorithm	Accuracy	Precision	Recall	FScore	TN	TP	FP	FN
Decision Tree	0.98	0.95	0.98	0.96	6971	2873	71	85
Naive Bayes	0.92	0.81	0.88	0.84	6962	2288	80	670
Neural Net	0.96	0.90	0.95	0.92	6861	2784	181	174
AdaBoost	0.98	0.94	0.97	0.96	6956	2866	86	92
LDA	0.89	0.72	0.86	0.79	6588	2354	454	604
Random Forest	0.99	0.97	0.99	0.98	7001	2922	41	36

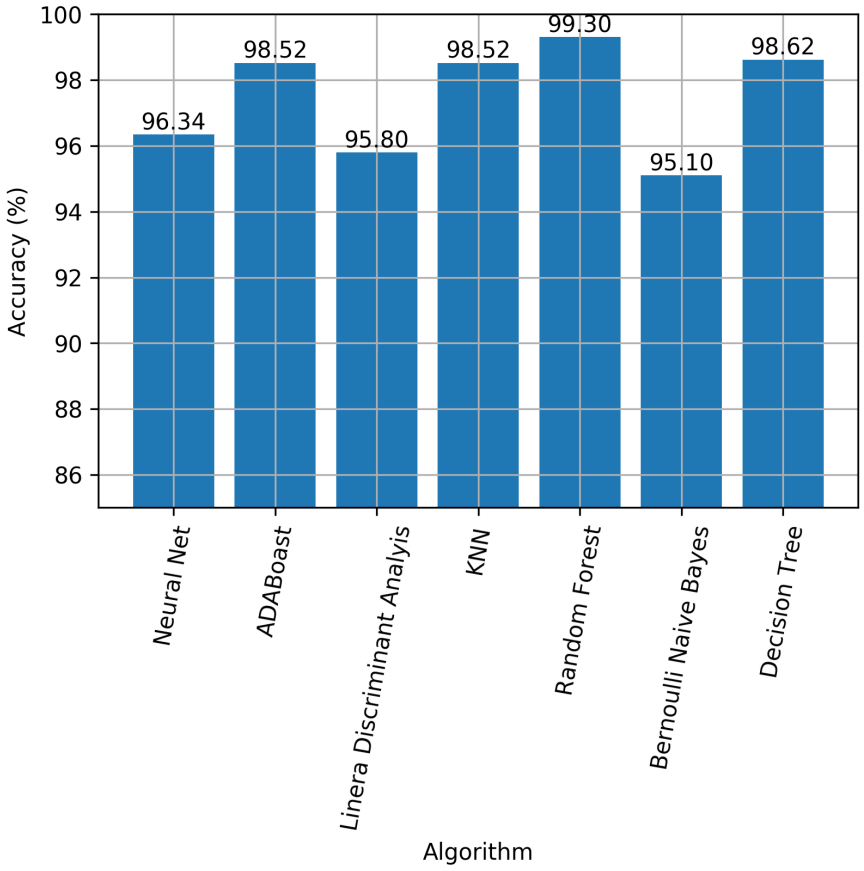

Figure 6: Comparison in performance of different algorithms

7.3 Inferences

- It was observed that there are some PE-file features that are specific to malware and others to benign programs. These features appear with different frequencies between these two categories of PE-files.
- **Performance with new feature set**. By dropping few headers and modifying few header values as explained in section 6.2, we were able to increase the accuracy of different models. Models like KNN and LDA which were earlier giving accuracy less than 90% showed more improvement.
- Random forest giving best performance. It was observed that Random

forest, Decision trees and Adaboost performed much better achieving accuracy more than 98%.

- Considering the detection time, we can conclude that our system is adequate for the real-time detection of malware.

- Analyzing false positives and false negatives. While analysing false positives, it was found that values of header fields like Characteristics, ImageBase, Version, SectionAlignment, FileAlignment followed standard values in malicious files. Hence, developer has ensured that virus PE header has standard values for these fields to skip detection.

- The detection is based on the features obtained from the executable file using the peinfo library in the python. Like every other techniques, it is possible to manipulate the behaviour of the malware detector by providing the false values in various attributes of the the executable files. We have done a comparative study by taking the 48 samples (40 malicious and 8 benign) and change their PE header attributes like **Characteristics, SizeOfInitializedData, SizeOfUninitializedData , BaseOfData , ImageBase and SectionsNb** with values from the benign samples for these attributes. We have observed the changes in the classification of the original and tainted samples.The differences in count are for original and tainted for the same model are as follow.

Classifier	Misclassified (out of 48)
Decision Tree	4
Naive Bayes	0
Adaboast	0
Neural Net	14
Random Forest	2

As expected, Naive Bayes, Adaboost and Random forest were more robust for such changes while neural networks performed worse.

8 Related work and critique

We referred and studied different papers as given in bibliography section but following papers were studied in details to understand the basic concept and

raw idea to implement our project.

- **A learning model to detect maliciousness of portable executable using integrated feature set**. The paper explains in details about the collection, prepossessing, feature selection and analysis using six machine learning techniques and are able to achieve accuracy to the tune of 60%-98%. However, paper didn't deliberate on reasons why few features were dropped and why classifier gave false positives or negatives on test data. The paper also does not highlight about the data set which deviated from the normal course. The data set was not tweaked and tested for abnormal PE header values.

- **A Chi-Square-Based Decision for Real-Time Malware Detection Using PE-File Feature** by Mohamed Belaoued and Smaine Mazouzi focused on reducing detect time in malware detection on PE files. They analyzed APIs (Application Programming Interfaces) and TPFs (Technical PE Features).They tried to find frequency of APIs and frequency of optional header field for malware and benign files.

- **Malware Detection Based on Multiple PE Headers Identification and Optimization for Specific Types of Files** by Filip Zatloukal and Jiri Znoj. They have proposed an approach for malware detection using multiple PE headers occurrences ie they try to detect malware by detecting multiple headers. But the important condition for this classification is that when malware connects itself (including PE header) to code , it do so without destroying header of previous file. Also, their work was restricted to executable files.

9 Challenges

- Finding a well defined good training dataset is very challenging as good classified dataset is essential to best train our model. We have used dataset available in public domain.

- Feature selection[1]. Selecting desired features and finding their dependencies is challenging. There is need to do empirical analysis to study header fields and reduce the dimensionality of a feature set. For this various papers using different techniques for feature selection were

referred and many trials were done taking different subset of features and hereby improving accuracy of models.

10 Conclusion

- The proposed technique used static analysis technique to extract the features which have low time and resource requirement than dynamic analysis. Good classification accuracy can be achieved by building malware classifier using header fields' value alone as the feature.
- The real-world scenario is unpredictable and can be different than experimental environment so to protect a sensitive system from malware, it is not advisable to use only header's values based classifier. For example, a carefully crafted malware would have a benign header and malicious payload hidden in the body of PE file.
- We have experimented with PE file format but as the proposed feature engineering method is very generic, it can be extended to other file formats such as image, pdf, audio and including mobile OS such as Android and iOS

References

[1] Ajit Kumar, K S Kuppusamy, and Aghila Gnanasekaran. A learning model to detect maliciousness of portable executable using integrated feature set. In *Journal of King Saud University - Computer and Information Sciences*, 01 2017.

[2] D. Gavriluţ, M. Cimpoeşu, D. Anton, and L. Ciortuz. Malware detection using machine learning. In *2009 International Multiconference on Computer Science and Information Technology*, pages 735–741, Oct 2009.

[3] Swapnaja Hiray Smita Ranveer. Comparative analysis of feature extraction methods of malware detection, June 2015.

[4] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[5] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.

[6] Christopher C. Lamb Timothy J. Draelos Justin E. Doak James B. Aimone Conrad D. James Michael R. Smith, Joe B. Ingram. Dynamic analysis of executables to detect and characterize malware. Nov 2017.

[7] Prateek lalwani. https://github.com/prk54/malware-detection-machine-learning-approach.

A Code to extract PE headers from .EXE file

```
import pefile
import os
import csv

rootdir = "/home/amitp/Desktop/cs741/peinfo/files"
output= "/home/amitp/Desktop/cs741/peinfo/raw_features"


f = open(output, 'wt')


for subdir,dirs,files in os.walk(rootdir):
    for file in files:
        input_file = rootdir + '/' + file
        try:
            pe =  pefile.PE(input_file)
            print(pe)
        except Exception as e:
            print ("Exception while loading file: ", e)
        else:
            try:
                features = pe
                #f.write(features)
                #writer.writerow(features)
            except Exception as e:
                print ("Exception while opening and writing CSV file: "

f.close()
```

B Code for malware detection from headers

```
# coding: utf-8

# In[103]:
```

```
import numpy as np
import matplotlib.pyplot as plt
import os
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split
from sklearn import metrics
from sklearn.cross_validation import cross_val_score
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB,BernoulliNB,MultinomialNB,Mu
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
from numpy import genfromtxt
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve
import pandas as pd
from matplotlib.pyplot import hist
from sklearn.metrics import confusion_matrix
from sklearn.metrics import recall_score
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.datasets import make_classification
from sklearn.ensemble import ExtraTreesClassifier


# In[104]:


data=pd.read_csv('data.csv')
#data2=pd.read_csv('data.csv')
```

```
# In[105]:


data=data.drop("Name",1)
#data2=data2.drop("Name",1)
data=data.drop("md5",1)
#data2=data2.drop("md5",1)
dataset= data.as_matrix();
shape=dataset.shape
X=dataset[:,:shape[1]-1]
Y=dataset[:, shape[1]-1]
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=0.1)


# In[106]:


#implementing Random forest for feature importance
RandForest=RandomForestClassifier(random_state=0)
RandForest.fit(X_train,Y_train)
data_x=data.drop('legitimate',1)
importances = pd.DataFrame({'feature':data_x.columns,'importance':np.rou
importances = importances.sort_values('importance',ascending=False).set_

print (importances)

# Plot the feature importances of the forest



ax=importances.iloc[0:14].plot.bar()
fig = ax.get_figure()
fig.savefig('importance2.png',dpi=300, format='png', bbox_inches='tight
#importances.plot.savefig(os.path.join('importance.png'), dpi=300, form
```

```
# In[107]:


forest = ExtraTreesClassifier(n_estimators=250,
                              random_state=0)

forest.fit(X_train, Y_train)
#importances = forest.feature_importances_
#std = np.std([tree.feature_importances_ for tree in forest.estimators_
 #            axis=0)
#indices = np.argsort(importances)[::-1]
#data_x=data.drop('legitimate',1)


# In[109]:


print(X_train.shape)
importances = pd.DataFrame({'feature':data_x.columns,'importance':np.rou
#print(data_x.columns)
#print(importances)
importances = importances.sort_values('importance',ascending=False).set_

print (importances)
# Print the feature ranking
ax=importances.iloc[0:14].plot.bar()
fig = ax.get_figure()
fig.savefig('importanceextratree.png',dpi=300, format='png', bbox_inches
#importances.plot.savefig(os.path.join('importance.png'), dpi=300, form




# In[66]:
```

```
data=data.drop("SectionsMinEntropy",1)
#data=data.drop("SectionsMaxEntropy",1)
data=data.drop("Machine",1)
data=data.drop("ResourcesMeanEntropy",1)
#data=data.drop("ResourcesMinEntropy",1)
data=data.drop("ResourcesMaxEntropy",1)
#data=data.drop("ResourcesMinSize",1)
#data=data.drop("ResourcesMaxSize",1)
data=data.drop("LoadConfigurationSize",1);
data=data.drop("SizeOfOptionalHeader",1);
data=data.drop("AddressOfEntryPoint",1);
#data=data.drop("BaseOfData",1);
data=data.drop("CheckSum",1)
data=data.drop("SectionsMinVirtualsize",1)
data=data.drop("SectionMaxVirtualsize",1)
data=data.drop("BaseOfCode",1)
data=data.drop("LoaderFlags",1)
#data=data.drop("Subsystem",1)
#data=data.drop("DllCharacteristics",1)
#data=data.drop("Characteristics",1)
data=data.drop("MinorLinkerVersion",1)
data=data.drop("MajorLinkerVersion",1)
#data=data.drop("MajorOperatingSystemVersion",1)
data=data.drop("MinorOperatingSystemVersion",1)
data=data.drop("MajorImageVersion",1)
#data=data.drop("MinorImageVersion",1)
data=data.drop("MajorSubsystemVersion",1)
data=data.drop("MinorSubsystemVersion",1)
data=data.drop("SizeOfCode",1)
data=data.drop("SectionsMinRawsize",1)
data=data.drop("SectionMaxRawsize",1)
data=data.drop("SectionsMeanEntropy",1)
data=data.drop("SectionsMeanVirtualsize",1)
data=data.drop("SizeOfHeapCommit",1)
data=data.drop("SizeOfStackCommit",1)
data=data.drop("SizeOfHeapReserve",1)
#ResourcesNb
```

```
data


# In[67]:


#data["SizeOfHeaders"].apply(int)
#data["FileAlignment"].apply(int)
def changeimagebase(ImageBase):
    result=0
    if ImageBase % (64*1024)==0 and ImageBase in [268435456,65536,41943
        result=1
    #print(result)
    return result;
#data["FileAlignment"].describe()


# In[68]:


def changesectionalignment(sectionalignment,filealignment):
    result=0;
    if sectionalignment>=filealignment :
        result=1
    return result



# In[69]:


def changefilealignment(sectionalignment,filealignment):
    result=0
    if sectionalignment >=512 :
        if filealignment % 2==0 and filealignment in range (512,65537):
            result=1
    else :
        if filealignment==sectionalignment :
```

```
            result=1
    return result


# In[70]:


def changeimagesize(sectionalignment,imagesize):
    return int(imagesize%sectionalignment==0)


# In[71]:


def changeheadersize(headersize,filealignment):
    result=0
    if(headersize>=filealignment):
        if(headersize%filealignment==0):
            result=1

    return result


# In[72]:


def changeNumberOfRvaAndSizes(NumberOfRvaAndSizes):
    if NumberOfRvaAndSizes >16 :
        return 16;
    return NumberOfRvaAndSizes


# In[73]:


def changeentropy(entropy):
    lowentropy=0
    highentropy=0
```

```
    if entropy>7 :
        highentropy=1
    if entropy <1 :
        lowentropy=1
    return lowentropy,highentropy


# In[74]:


data["NumberOfRvaAndSizes"]=data.apply(lambda x: changeNumberOfRvaAndSi
#data["ImageBase"]= data.apply(lambda x: changeimagebase(x['ImageBase'])
data["SizeOfHeaders"] = data.apply(lambda x: changeheadersize(x['SizeOfH
data['SectionAlignment'] = data.apply(lambda x: changesectionalignment(x
data['FileAlignment'] = data.apply(lambda x: changefilealignment(x['Sect
data['SizeOfImage'] = data.apply(lambda x: changeimagesize(x['SectionAli


# In[75]:


#check the size of dataset
print(data)
dataset= data.as_matrix();
#dataset2=data2.as_matrix();
print(np.shape(dataset))
print(dataset)


# In[76]:


#shuffle the dataset rows
np.random.shuffle(dataset)
np.random.shuffle(dataset)
np.random.shuffle(dataset)
np.random.shuffle(dataset)
```

```
np.random.shuffle(dataset)
np.random.shuffle(dataset)


# In[77]:


#seperate out input and output
shape=dataset.shape
X=dataset[:,:shape[1]-1]
Y=dataset[:, shape[1]-1]


# In[78]:


print(X)
print(Y)


# In[79]:


'''
validate_x=X[100000:,:]
validate_y=Y[100000:]
print(validate_x.shape)
print(validate_y.shape)
X=X[:100000,:]
Y=Y[:100000]
'''


# In[80]:


#split dataset into train and test
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=0.1)
```

```
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)


# In[81]:


#start implementing the classifiers and store their outputs
output={}


# In[82]:


#implementing K Nearest Neighbour
'''
KNN=KNeighborsClassifier(3)
KNN.fit(X_train, Y_train)
score=KNN.score(X_test, Y_test)
print ("KNN : ", score)
output["KNN"]=score
y_score = KNN.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))

'''


# In[83]:


#implementing Decision Tree
```

```
DTC= DecisionTreeClassifier(max_depth=5)
DTC.fit(X_train,Y_train)
score=DTC.score(X_test,Y_test)
print ("DTC : ", score)
output["Decision Tree"]=score
y_score = DTC.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))


# In[84]:


#implementing Linear Discriminant Analysis
'''
LDA= LinearDiscriminantAnalysis()
LDA.fit(X_train,Y_train)
score=LDA.score(X_test,Y_test)
print ("LDA : ", score)
output["Linera Discriminant Analyis"]=score
y_score = LDA.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))
'''


# In[85]:


#implementing Naive Bayes
```

```
BNB=BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=Tru
BNB.fit(X_train,Y_train)
score=BNB.score(X_test,Y_test)
print ("BNB : ", score)
output["Bernoulli Naive Bayes"]=score
y_score = BNB.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))


# In[86]:


#implementing Naive Bayes
MNB=MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
MNB.fit(X_train,Y_train)
score=MNB.score(X_test,Y_test)
print ("MNB : ", score)
#output["Multinomial Naive Bayes"]=score
y_score = MNB.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))


# In[87]:


#implementing AdaBoast
ADABoast=AdaBoostClassifier()
ADABoast.fit(X_train,Y_train)
```

```
score=ADABoast.score(X_test,Y_test)
print ("ADABoast : ", score)
output["ADABoast"]=score
y_score = ADABoast.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))


# In[88]:


#implementing Multilayer Perceptron
MLP=MLPClassifier(alpha=0.1)
'''
MLP=MLPClassifier(activation='relu', alpha=0.1, batch_size='auto',
        beta_1=0.9, beta_2=0.999, early_stopping=False,
        epsilon=1e-08, hidden_layer_sizes=(5, 2), learning_rate='constan
        learning_rate_init=0.001, max_iter=200, momentum=0.9,
        nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle=Tr
        solver='lbfgs', tol=0.0001, validation_fraction=0.1, verbose=Fal
        warm_start=False)'''
MLP.fit(X_train,Y_train)
score=MLP.score(X_test,Y_test)
print ("MLP : ", score)
output["Neural Net"]=score
y_score = MLP.predict(X_test)
precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))
```

```
# In[90]:


data=data.drop('legitimate',1)


# In[91]:


#implementing Random forest
RandForest=RandomForestClassifier(random_state=0)
RandForest.fit(X_train, Y_train)
'''
data_x=data.drop('legitimate',1)
importances = pd.DataFrame({'feature':data_x.columns,'importance':np.rou
importances = importances.sort_values('importance',ascending=False).set_

print (importances)
importances.plot.bar()
'''
score=RandForest.score(X_test, Y_test)
print ("Random Forest : ", score)
output["Random Forest"]=score
y_score = RandForest.predict(X_test)
incorrect=[]
mispredict=0
df=[]
for i in range(len(Y_test)): # assuming the lists are of the same length
    if Y_test[i]==0:
        if y_score[i]==0:
            s=1
        else:
            incorrect.append(i)
            mispredict=mispredict+1
            df.append(X_test[i])

cols=(list(data))
print(cols)
```

```
my_df = pd.DataFrame(df)
my_df.columns=cols

my_df.to_csv("error.csv", index=False,header=True)
print(mispredict)

precision = average_precision_score(Y_test, y_score)
recall=recall_score(Y_test, y_score, average='macro')
print("Precision : ",precision)
print("Recall : ",recall)
print("Fscore : " ,2*precision*recall/(precision+recall))
print( confusion_matrix(Y_test, y_score))



# In[92]:


my_df


# In[46]:


names=list(output.keys())
print(names)


# In[47]:


heights=list(output.values())
new_heights=[i * 100 for i in heights]
print(new_heights)


# In[48]:
```

```
Y = np.arange(len(names))
bar=plt.bar(Y, new_heights)
plt.xticks(Y, names, rotation=80)
plt.ylabel('Accuracy (%)')
plt.xlabel('Algorithm')
plt.axis([None,None , 85.0, 100.0])
plt.grid(True)
# Add counts above the two bar graphs
for rect in bar:
    height = rect.get_height()
    plt.text(rect.get_x() + rect.get_width()/2.0, height, '%.2f' % heigh
plt.savefig(os.path.join('test.png'), dpi=300, format='png', bbox_inches
plt.show()


# In[49]:


PX=X_test
PY=Y_test
pshape=PX.shape


# In[49]:


out=KNN.predict(PX)
print ("Error KNN" ,np.sum(np.abs(out-PY))/pshape[0]*100,"%")


# In[50]:


out=DTC.predict(PX)
print ("Error Decision Tree " ,np.sum(np.abs(out-PY))/pshape[0]*100,"%")
```

```
# In[51]:


out=LDA.predict(PX)
print ("Error Linear Discriminant Analysis " ,np.sum(np.abs(out-PY))/psh


# In[52]:


out=BNB.predict(PX)
print ("Error Bernoulli Naive Nayes " ,np.sum(np.abs(out-PY))/pshape[0]*


# In[53]:


out=MNB.predict(PX)
print ("Error Multinomial Naive Bayes " ,np.sum(np.abs(out-PY))/pshape[0


# In[54]:


out=ADABoast.predict(PX)
print ("Error ADA Boast " ,np.sum(np.abs(out-PY))/pshape[0]*100,"%")


# In[55]:


out=MLP.predict(PX)
print ("Error Neural Networks " ,np.sum(np.abs(out-PY))/pshape[0]*100,"%
```