VIRUS DETECTION USING DATA MINING

Submitted in partial fulfilment

Of

Mini project in Bachelor of Technology

Submitted by

1. **JAYA SHANKAR REDDY KOMMURU 411529**
2. **VISHNU NAGINENI 411551**
3. **SAI SHIVA KUMAR KATHRI 411535**
4. **V. PRATHYUSH 411576**

Under the Supervision of

Mr. BKSP Kumar Raju Alluri

NATIONAL INSTITUTE OF TECHNOLOGY,

Andhra Pradesh.

SUMMARY

- **The project consists of two types of modules. One is the Implemented modules and other is the Theoretical Modules. There are 6 Implemented modules and 3 Theoretical Modules.**
- **The Implementation Modules use two different features for classification of malware and non-malware files, namely hex dump and System call sequences.**
- **The hex dump feature is used in Static Analysis where different classification algorithms of data mining are applied and their accuracy is being compared.**
- **Once identifying that the virus is present, the same hex dump features are used to identify which type of virus has injected that program.**
- **The System call sequences feature is used in Dynamic Analysis where clustering Algorithms are applied for the non-malware data set. And any outlier detected is treated as a Malware.**
- **Instead of considering all the sequences, the top k malware sequences are found using the TKS Algorithm. Now these top k sequences are compared with a given sequence and found out that whether it is malicious or genuine.**
- **Considering an assumption that the malware sequences are complete, from a given sequence, all the in between calls are counted apart from the malware calls. The trace having huge count is more probable to be malicious.**
- **Considering another assumption that there are fixed number of genuine activities for an Operating System, all the in between calls of a sequence are appended and checked for genuine activities. More the genuity, more the probability of it being non malicious.**
- **Theoretical model of integrating the static and dynamic model is given.**
- **Two different tools are explored for dynamic Analysis, namely Cuckoo sandbox and Intel’s PIN tool.**
- **PIN tool is used for detecting ROP.**
- **Cuckoo sandbox is used for extracting run time features other than System call Sequences.**

DATA MINING:

Data mining is the computing process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems. It is an essential process where intelligent methods are applied to extract data patterns. It is an interdisciplinary subfield of computer science. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use. Aside from the raw analysis step, it involves database and data management aspects, data pre-processing, model and inference considerations, interestingness metrics, complexity considerations, post-processing of discovered structures, visualization, and online updating. Data mining is the analysis step of the "knowledge discovery in databases" process, or KDD.

The actual data mining task is the semi-automatic or automatic analysis of large quantities of data to extract previously unknown, interesting patterns such as groups of data records (CLUSTER ANALYSIS), unusual records (ANAMOLY DETECTION), and dependencies (ASSOCIATION RULE MINING,SEQUENTIAL PATTERN MINING). This usually involves using database techniques such as SPATIAL INDICES. These patterns can then be seen as a kind of summary of the input data, and may be used in further analysis or, for example, in machine learning and predictive analytics. For example, the data mining step might identify multiple groups in the data, which can then be used to obtain more accurate prediction results by a DECISION SUPPORT SYSTEM. Neither the data collection, data preparation, nor result interpretation and reporting is part of the data mining step, but do belong to the overall KDD process as additional steps.

METHODS IN DATA MINING:

1. Preprocessing techniques.
2. Association Mining.
3. Classification Mining.
4. Cluster analysis.
5. Sequence Mining.

MALWARE

The term malware is used when referring to malicious software, which is designed to disrupt or gain unauthorized access to a system. There are various types of malware, out of which our project concentrates on virus.

Virus: It is a malicious code which gets attached to normal files in a computer and remains idle until the files are opened/executed. Once executed, the virus gets multiplied and attaches to other non-infected files and destroys the functionality of the computer.

Hence, it is very essential to differentiate between a malware and a cleanware once an unknown file has entered into the system inorder to ensure it will not perform any malicious activity.

This project aims to perform malware detection, where models are trained on different kinds of data sets. These models are trained using different data mining algorithms and tested using cross validation and percentage-split.

METHODS USED IN PROJECT:

❖ **Preprocessing techniques:**

- Hex dump generation.
- String to word vector.
- n grams tokeniser.

❖ **Classification Mining:**

- Naive Bayes**:** The Bayesian Classification represents a supervised learning method as well as a statistical method for classification. Assumes an underlying probabilistic model and it allows us to capture uncertainty about the model in a principled way by determining probabilities of the outcomes. It can solve diagnostic and predictive problems.
- Naive Bayes Multinomial: The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts.
- NaiveBayesMultinomial Text: Multinomial Naive Bayes is a specialized version of Naive Bayes that is designed more for text documents. Whereas simple naive Bayes would model a document as the presence and absence of particular words, multinomial naive bayes explicitly models the word counts and adjusts the underlying calculations to deal with in.
- NaivebayesMultinomial updatable : This version of Naïve Bayes takes the input increamentally and keeps updating the model for each tuple.
- Naive Bayes Updateable :This version is similar to the above one.
- J-48: It is java implementation of C 4.5 in weka.
- Random Forest: Random Forests are a combination of tree predictors where each tree depends on the values of a random vector sampled independently with the same distribution for all trees in the forest. The basic principle is that a group of "weak learners" can come together to form a "strong learner".
- Random Tree: This version comes under the Random Forest Algorithm.

❖ **Clustering:**

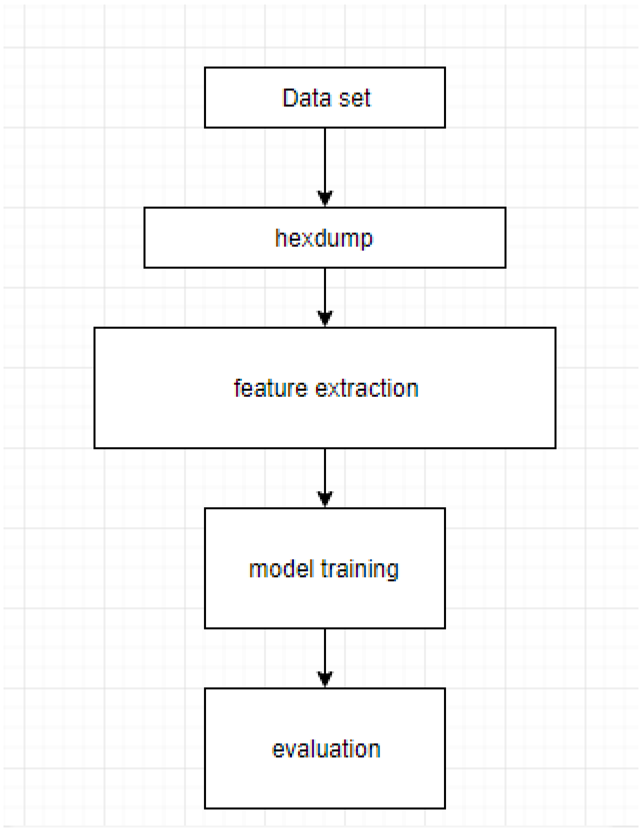
- Agglomerative Hierarchical Clustering (single, complete, average, mean, center, median): Agglomerative Hierarchical clustering -This algorithm works by grouping the data one by one on the basis of the nearest distance measure of all the pairwise distance between the data point.
- Simple K means: Forms K clusters. All the tuples in a cluster have same behavior. K is the input.
- Density Based Clustering: Density based clustering algorithm has played a vital role in finding non-linear shapes structure based on the density. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is most widely used density based algorithm. It uses the concept of density reachability and density connectivity.

❖ **Sequence Mining:**

- TKS Algorithm: TKS is an algorithm for discovering the top-k most frequent sequential patterns in a sequence database. The input of TKS is a sequence database and a user-specified parameter named *k* (a positive integer representing the desired number of patterns to be found).

IMPLEMENTED MODULES:

MODULE 1: STATIC ANALYSIS



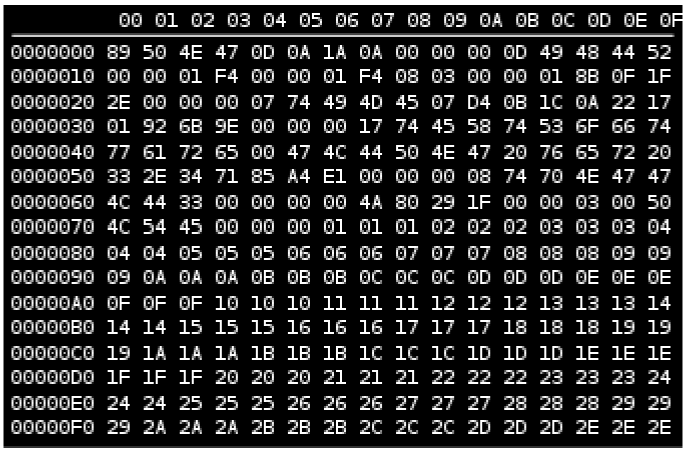
Static Analysis: Without executing the malware, many features like strings embedded in binary,PE header, disassembly can be extracted from a file using debugging tools. The dataset which we considered contains directly the hex dump.

Data set: https://www.kaggle.com/c/malware-classification/data

Description: This data set contains only malware files. The files consist of **hex dump** of different malwares of windows.

Non-malware files are also needed. So, around 100 windows executable files are taken and hex dump of them is generated using a C++ program.

Feature used: Hex dump is the hexadecimal representation of the instructions of an executable.



```
        00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000000 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52
0000010 00 00 01 F4 00 00 01 F4 08 03 00 00 01 8B 0F 1F
0000020 2E 00 00 00 07 74 49 4D 45 07 D4 0B 1C 0A 22 17
0000030 01 92 6B 9E 00 00 00 17 74 45 58 74 53 6F 66 74
0000040 77 61 72 65 00 47 4C 44 50 4E 47 20 76 65 72 20
0000050 33 2E 34 71 85 A4 E1 00 00 00 08 74 70 4E 47 47
0000060 4C 44 33 00 00 00 00 4A 80 29 1F 00 00 03 00 50
0000070 4C 54 45 00 00 00 01 01 01 02 02 02 03 03 03 04
0000080 04 04 05 05 05 06 06 06 07 07 07 08 08 08 09 09
0000090 09 0A 0A 0A 0B 0B 0B 0C 0C 0C 0D 0D 0D 0E 0E 0E
00000A0 0F 0F 0F 10 10 10 11 11 11 12 12 12 13 13 13 14
00000B0 14 14 15 15 15 16 16 16 17 17 17 18 18 18 19 19
00000C0 19 1A 1A 1A 1B 1B 1B 1C 1C 1C 1D 1D 1D 1E 1E 1E
00000D0 1F 1F 1F 20 20 20 21 21 21 22 22 22 23 23 23 24
00000E0 24 24 25 25 25 26 26 26 27 27 27 28 28 28 29 29
00000F0 29 2A 2A 2A 2B 2B 2B 2C 2C 2C 2D 2D 2D 2E 2E 2E
```

Fig1: Hex dump of a file. The starting 4 bytes of each line is the memory address followed by the hex dump.

N Gram:

An N-gram is a sequence of bytes of fixed or variable length, extracted from the hexadecimal dump of an executable program. Here, we are using n-grams as a general term for both overlapping and non-overlapping byte sequences.

EXAMPLE:

Hex Dump: 89 50 4E 47 0D 0A 1A

If N=2 (known as bigrams), then the n grams would be:

89 50

50 4E

4E 47

47 0D

0D 0A

A 1A

If N=3 (known as trigrams), then the n grams would be :

89 50 4E

50 4E 47

4E 47 0D

47 0D 0A

0D 0A 1A

If N=4(known as four grams), then the n grams would be :

89 50 4E 47

50 4E 47 0D

4E 47 0D 0A

47 0D 0A 1A

Text Directory Loader

Loads all text files in a directory and uses the subdirectory names as class labels. The content of the text files will be stored in a String attribute; the filename can be stored as well.

Example directory layout:

```
...
 |
 +- text_example
    |
    +- class1
    |   |
    |   + file1.txt
    |   |
    |   + file2.txt
    |   |
    |   ...
```

```
   |
   +- class2
   |   |
   |   + another_file1.txt
   |   |
   |   + another_file2.txt
   |   |
   |   ...
```

The above directory structure can be turned into an ARFF file like this:

weka.core.converters.TextDirectoryLoader:

Loads all text files in a directory and uses the subdirectory names as class labels. The content of the text files will be stored in a String attribute, the filename can be stored as well.

weka.filters.unsupervised.attribute.StringToWordVector:

Converts String attributes into a set of attributes representing word occurrence (depending on the tokenizer) information from the text contained in the strings. The set of words (attributes) is determined by the first batch filtered (typically training data). Hex dump generation using command in Weka.

Pre-processing steps:

1. All the malwares' hex dumps are placed in a folder named malware. Similarly all non-malwares' hex dumps are placed in a folder named non malware.
2. Both the above folders are placed in a parent folder.
3. The parent folder is loaded into Weka using simple CLI. As simple CLI is a bit faster than GUI.
4. The arff file generated is loaded into Weka. A filter in unsupervised attribute String to Word Vector is used. Again in this a tokeniser called n gram tokeniser is used.
5. Once the above filter is applied, all the hex dumps are converted into n grams and the data set converts to a binary data set.

Training the model:

1. Once the data set being pre-processed, the classification algorithms are applied.

Testing the model:

1. The data set is tested using both percentage split and cross validation.

Measurement metric:

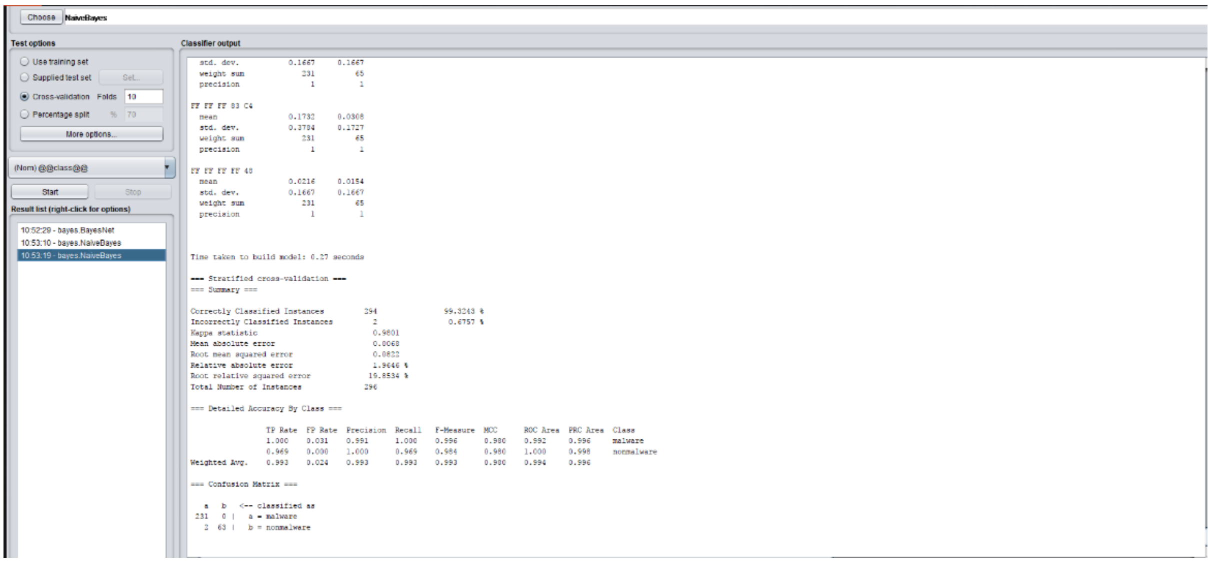
1. The metric used for measuring the accuracy is the confusion matrix for each algorithm.

Evaluation:

5-Grams:

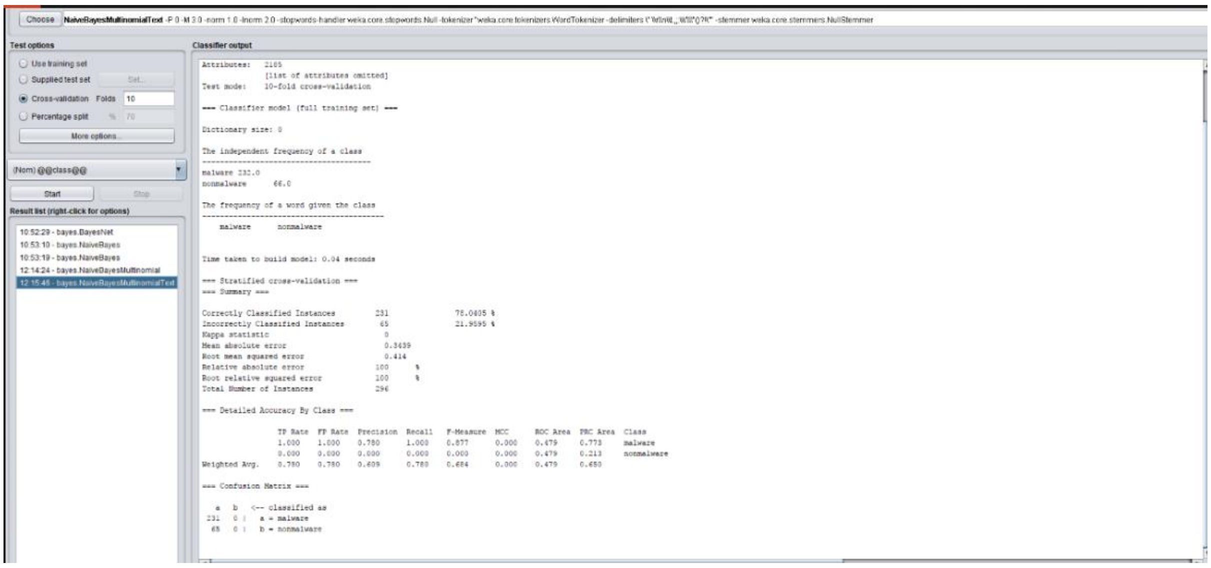
CROSS-VALIDATION: 10 Folds

Naïve-Bayes: 99.3243%



```
  std. dev.           0.1667     0.1667
  weight sum             231         65
  precision                1          1

FF FF FF 83 C4
  mean                0.1732     0.0308
  std. dev.           0.3784     0.1727
  weight sum             231         65
  precision                1          1

FF FF FF FF 48
  mean                0.0216     0.0154
  std. dev.           0.1667     0.1667
  weight sum             231         65
  precision                1          1



Time taken to build model: 0.27 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         294               99.3243 %
Incorrectly Classified Instances         2                0.6757 %
Kappa statistic                          0.9801
Mean absolute error                      0.0068
Root mean squared error                  0.0822
Relative absolute error                  1.9646 %
Root relative squared error             19.8534 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.031    0.991      1.000    0.996      0.980    0.992     0.996     malware
                 0.969    0.000    1.000      0.969    0.984      0.980    1.000     0.998     nonmalware
Weighted Avg.    0.993    0.024    0.993      0.993    0.993      0.980    0.994     0.996

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
   2  63 |   b = nonmalware
```

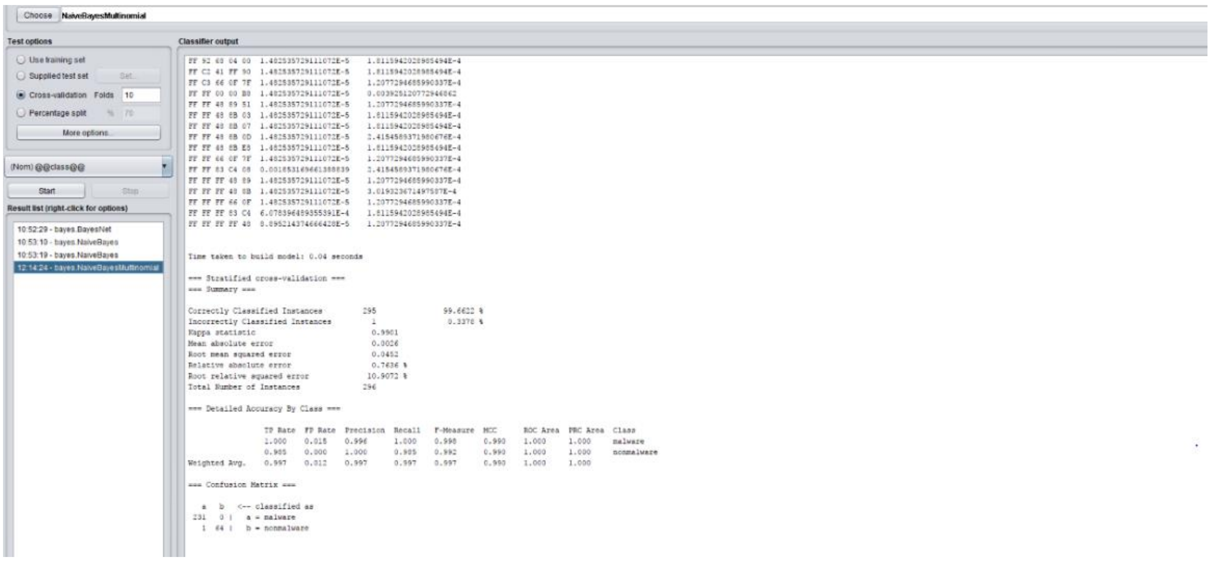
NaiveBayesMultinomialText:78.0405%


Choose
Test options
Use training set
Supplied test set
Set...
Cross-validation Folds 10
Percentage split % 70
More options...
(Nom) @@class@@
Start
Stop
Result list (right-click for options)
10:52:29 - bayes.BayesNet
10:53:10 - bayes.NaiveBayes
10:53:19 - bayes.NaiveBayes
12:14:24 - bayes.NaiveBayesMultinomial
12:15:46 - bayes.NaiveBayesMultinomialText
Classifier output

NaiveBayesMultinomialText -P 0 -M 3.0 -norm 1.0 -lnorm 2.0 -stopwords-handler weka.core.stopwords.Null -tokenizer "weka.core.tokenizers.WordTokenizer -delimiters \" \\r\\n\\t.,;:\\'\\"()?!\"" -stemmer weka.core.stemmers.NullStemmer

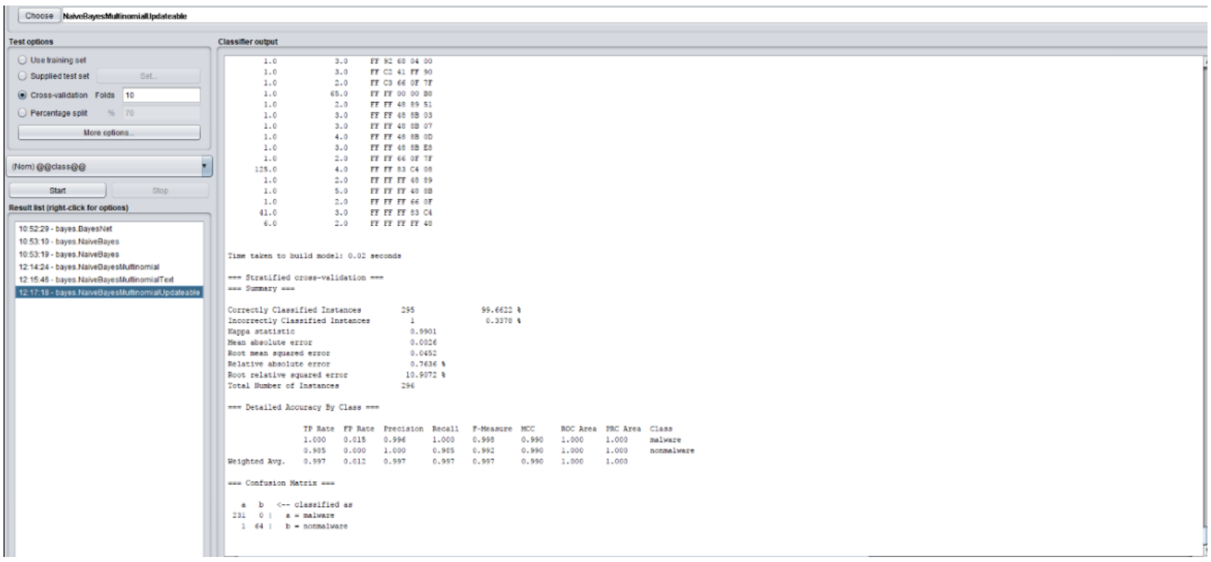
```
Attributes:   2185
              [list of attributes omitted]
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

Dictionary size: 0

The independent frequency of a class
--------------------------------------
malware 232.0
nonmalware      66.0

The frequency of a word given the class
-----------------------------------------
    malware      nonmalware


Time taken to build model: 0.04 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         231               78.0405 %
Incorrectly Classified Instances        65               21.9595 %
Kappa statistic                          0
Mean absolute error                      0.3439
Root mean squared error                  0.414
Relative absolute error                100      %
Root relative squared error            100      %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    1.000    0.780      1.000    0.877      0.000    0.479     0.773     malware
                 0.000    0.000    0.000      0.000    0.000      0.000    0.479     0.213     nonmalware
Weighted Avg.    0.780    0.780    0.609      0.780    0.684      0.000    0.479     0.650

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
  65   0 |   b = nonmalware
```

NaiveBayesMultinomial: 99.632%


Choose
NaiveBayesMultinomial
Test options
Use training set
Supplied test set
Set...
Cross-validation Folds 10
Percentage split % 70
More options...
(Nom) @@class@@
Start
Stop
Result list (right-click for options)
10:52:29 - bayes.BayesNet
10:53:10 - bayes.NaiveBayes
10:53:19 - bayes.NaiveBayes
12:14:24 - bayes.NaiveBayesMultinomial
Classifier output

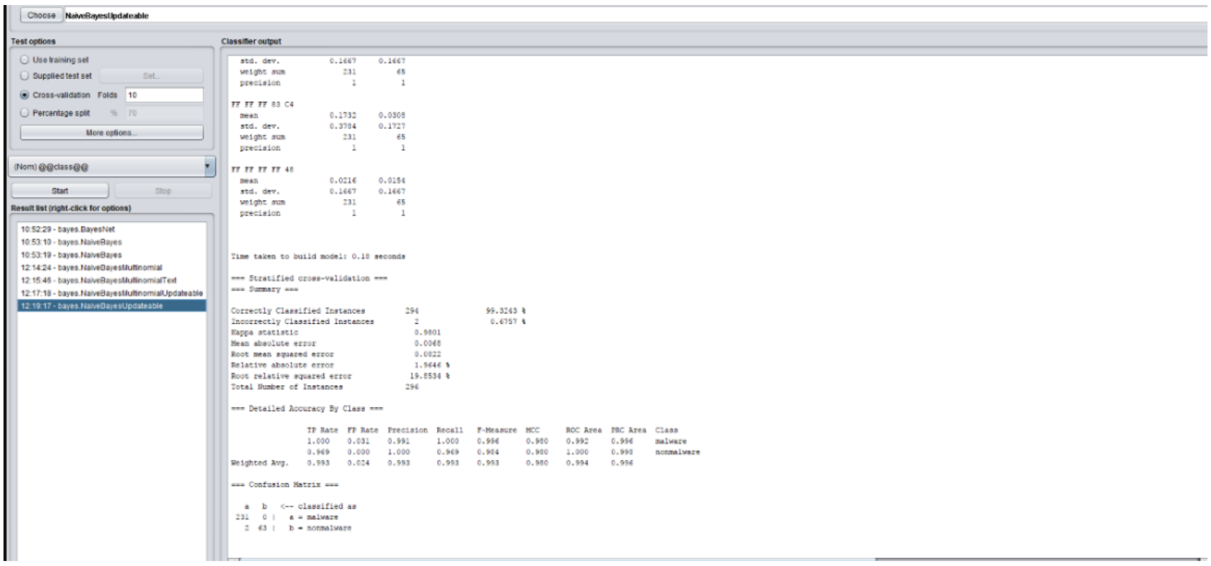
```
FF 92 68 04 00  1.482535729111072E-5    1.8115942028985494E-4
FF C2 41 FF 90  1.482535729111072E-5    1.8115942028985494E-4
FF C3 66 0F 7F  1.482535729111072E-5    1.2077294685990337E-4
FF FF 00 00 B8  1.482535729111072E-5    0.003925120772946862
FF FF 48 89 51  1.482535729111072E-5    1.2077294685990337E-4
FF FF 48 8B 03  1.482535729111072E-5    1.8115942028985494E-4
FF FF 48 8B 07  1.482535729111072E-5    1.8115942028985494E-4
FF FF 48 8B 0D  1.482535729111072E-5    2.4154589371980676E-4
FF FF 48 8B E8  1.482535729111072E-5    1.8115942028985494E-4
FF FF 66 0F 7F  1.482535729111072E-5    1.2077294685990337E-4
FF FF 83 C4 08  0.0018531696613888839   2.4154589371980676E-4
FF FF FF 48 89  1.482535729111072E-5    1.2077294685990337E-4
FF FF FF 48 8B  1.482535729111072E-5    3.0193236714975597E-4
FF FF FF 66 0F  1.482535729111072E-5    1.2077294685990337E-4
FF FF FF 83 C4  6.078396489355391E-4    1.8115942028985494E-4
FF FF FF FF 48  8.895214374666428E-5    1.2077294685990337E-4


Time taken to build model: 0.04 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         295               99.6622 %
Incorrectly Classified Instances         1                0.3378 %
Kappa statistic                          0.9901
Mean absolute error                      0.0026
Root mean squared error                  0.0452
Relative absolute error                  0.7636 %
Root relative squared error             10.9072 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.015    0.996      1.000    0.998      0.990    1.000     1.000     malware
                 0.985    0.000    1.000      0.985    0.992      0.990    1.000     1.000     nonmalware
Weighted Avg.    0.997    0.012    0.997      0.997    0.997      0.990    1.000     1.000

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
   1  64 |   b = nonmalware
```

NaiveBayesMultinomialUpdatable: 99.632%


Choose
NaiveBayesMultinomialUpdateable
Test options
Use training set
Supplied test set
Set...
Cross-validation Folds 10
Percentage split % 70
More options...
(Nom) @@class@@
Start
Stop
Result list (right-click for options)
10:52:29 - bayes.BayesNet
10:53:10 - bayes.NaiveBayes
10:53:19 - bayes.NaiveBayes
12:14:24 - bayes.NaiveBayesMultinomial
12:15:46 - bayes.NaiveBayesMultinomialText
12:17:18 - bayes.NaiveBayesMultinomialUpdateable
Classifier output

```
         1.0            3.0      FF 92 68 04 00
         1.0            3.0      FF C2 41 FF 90
         1.0            2.0      FF C3 66 0F 7F
         1.0           65.0      FF FF 00 00 B8
         1.0            2.0      FF FF 48 89 51
         1.0            3.0      FF FF 48 8B 03
         1.0            3.0      FF FF 48 8B 07
         1.0            4.0      FF FF 48 8B 0D
         1.0            3.0      FF FF 48 8B E8
         1.0            2.0      FF FF 66 0F 7F
       125.0            4.0      FF FF 83 C4 08
         1.0            2.0      FF FF FF 48 89
         1.0            5.0      FF FF FF 48 8B
         1.0            2.0      FF FF FF 66 0F
        41.0            3.0      FF FF FF 83 C4
         6.0            2.0      FF FF FF FF 48


Time taken to build model: 0.02 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         295               99.6622 %
Incorrectly Classified Instances         1                0.3378 %
Kappa statistic                          0.9901
Mean absolute error                      0.0026
Root mean squared error                  0.0452
Relative absolute error                  0.7636 %
Root relative squared error             10.9072 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.015    0.996      1.000    0.998      0.990    1.000     1.000     malware
                 0.985    0.000    1.000      0.985    0.992      0.990    1.000     1.000     nonmalware
Weighted Avg.    0.997    0.012    0.997      0.997    0.997      0.990    1.000     1.000

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
   1  64 |   b = nonmalware
```

NaiveBayesUpdatable:99.3243



NaiveBayesUpdateable

Test options: Cross-validation, Folds 10

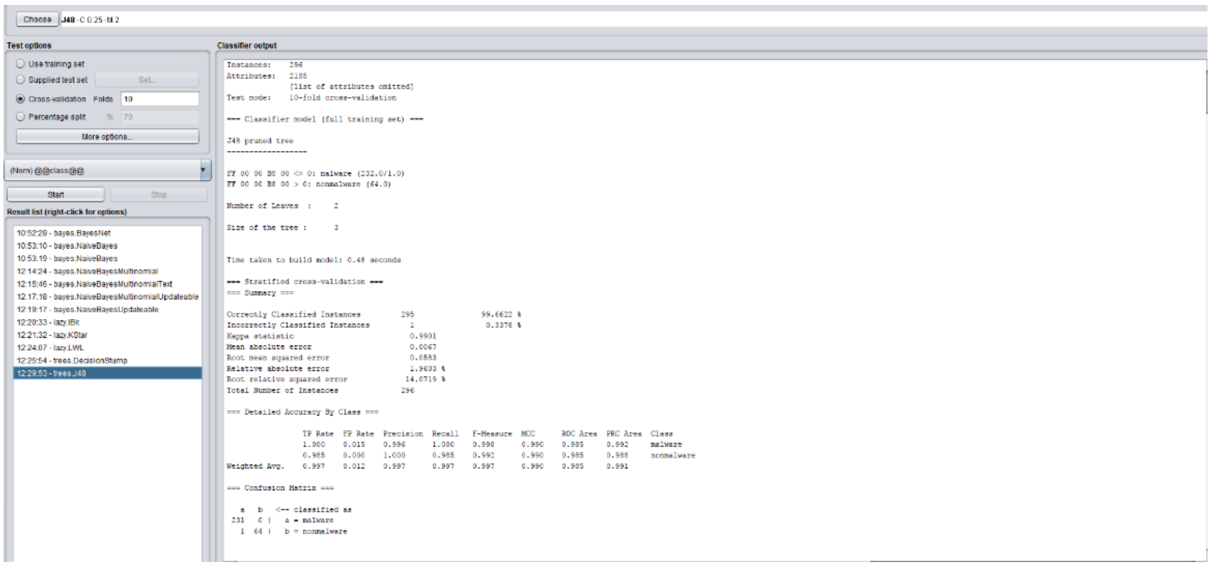
(Nom) @@class@@

Result list (right-click for options)
10:52:29 - bayes.BayesNet
10:53:10 - bayes.NaiveBayes
10:53:19 - bayes.NaiveBayes
12:14:24 - bayes.NaiveBayesMultinomial
12:15:46 - bayes.NaiveBayesMultinomialText
12:17:18 - bayes.NaiveBayesMultinomialUpdateable
12:19:17 - bayes.NaiveBayesUpdateable

Classifier output

```
  std. dev.          0.1667     0.1667
  weight sum            231         65
  precision               1          1

FF FF FF 83 C4
  mean               0.1732     0.0308
  std. dev.          0.3784     0.1727
  weight sum            231         65
  precision               1          1

FF FF FF FF 48
  mean               0.0216     0.0154
  std. dev.          0.1667     0.1667
  weight sum            231         65
  precision               1          1



Time taken to build model: 0.18 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         294               99.3243 %
Incorrectly Classified Instances         2                0.6757 %
Kappa statistic                          0.9801
Mean absolute error                      0.0068
Root mean squared error                  0.0822
Relative absolute error                  1.9646 %
Root relative squared error             19.8534 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.031    0.991      1.000    0.996      0.980    0.992     0.996     malware
                 0.969    0.000    1.000      0.969    0.984      0.980    1.000     0.998     nonmalware
Weighted Avg.    0.993    0.024    0.993      0.993    0.993      0.980    0.994     0.996

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
   2  63 |   b = nonmalware
```

J48: 99.6632%



J48 -C 0.25 -M 2

Test options: Cross-validation, Folds 10

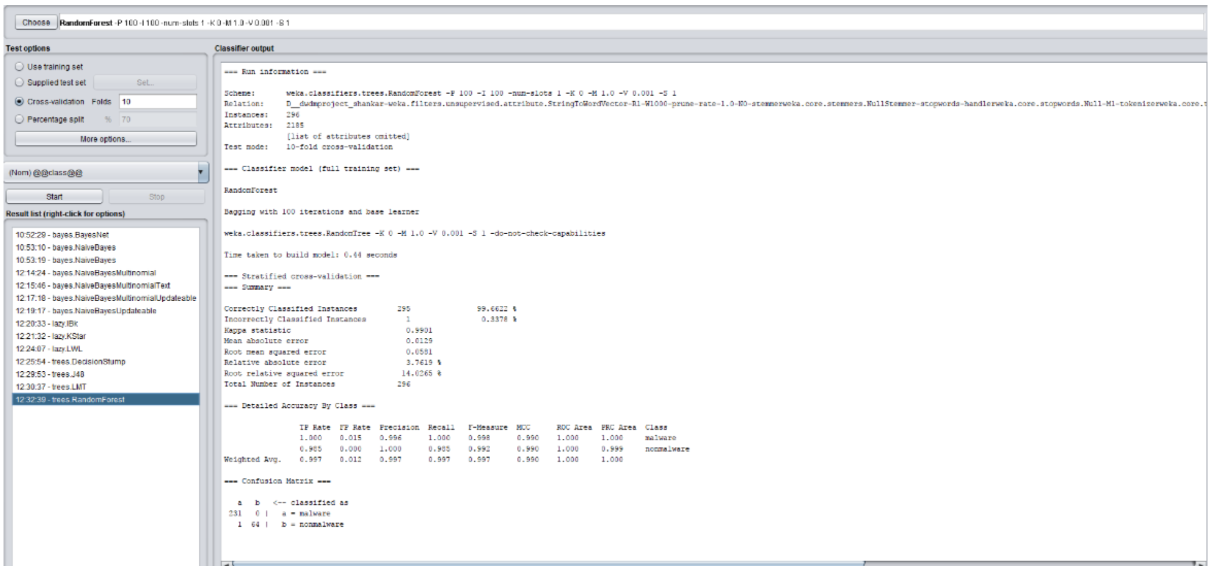
(Nom) @@class@@

Result list (right-click for options)
10:52:29 - bayes.BayesNet
10:53:10 - bayes.NaiveBayes
10:53:19 - bayes.NaiveBayes
12:14:24 - bayes.NaiveBayesMultinomial
12:15:46 - bayes.NaiveBayesMultinomialText
12:17:18 - bayes.NaiveBayesMultinomialUpdateable
12:19:17 - bayes.NaiveBayesUpdateable
12:20:33 - lazy.IBk
12:21:32 - lazy.KStar
12:24:07 - lazy.LWL
12:25:54 - trees.DecisionStump
12:29:53 - trees.J48

Classifier output

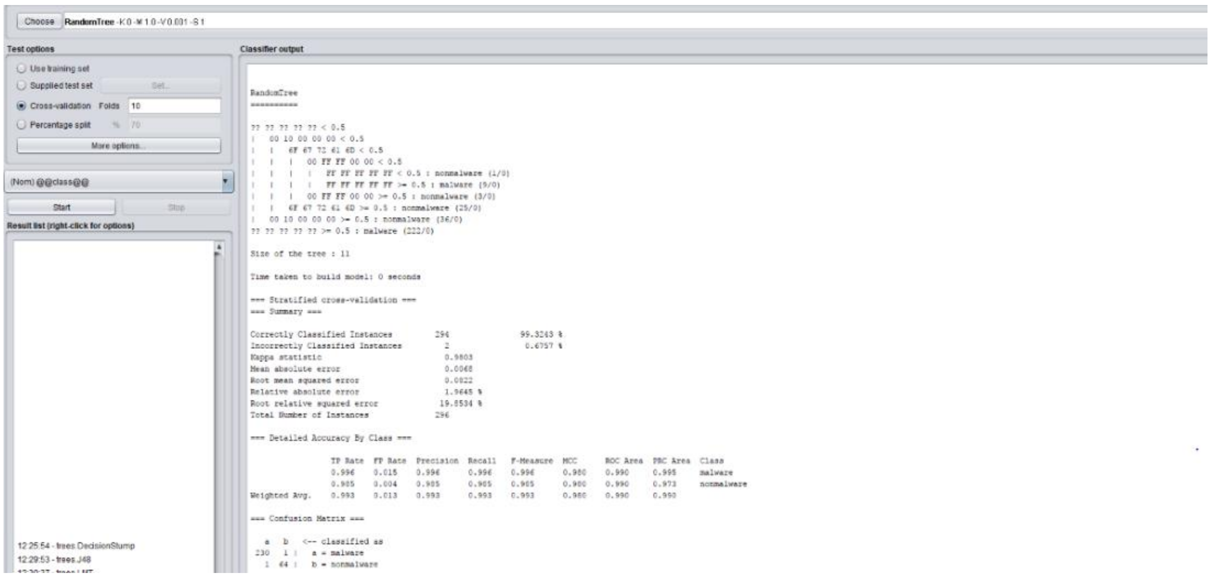
```
Instances:    296
Attributes:   2185
              [list of attributes omitted]
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
------------------

FF 00 00 B8 00 <= 0: malware (232.0/1.0)
FF 00 00 B8 00 > 0: nonmalware (64.0)

Number of Leaves  :     2

Size of the tree :      3


Time taken to build model: 0.48 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         295               99.6622 %
Incorrectly Classified Instances         1                0.3378 %
Kappa statistic                          0.9901
Mean absolute error                      0.0067
Root mean squared error                  0.0583
Relative absolute error                  1.9603 %
Root relative squared error             14.0719 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.015    0.996      1.000    0.998      0.990    0.985     0.992     malware
                 0.985    0.000    1.000      0.985    0.992      0.990    0.985     0.988     nonmalware
Weighted Avg.    0.997    0.012    0.997      0.997    0.997      0.990    0.985     0.991

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
   1  64 |   b = nonmalware
```

Random Forest: 99.662%



```
=== Run information ===

Scheme:       weka.classifiers.trees.RandomForest -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1
Relation:     D__dwdmproject_shankar-weka.filters.unsupervised.attribute.StringToWordVector-R1-W1000-prune-rate-1.0-N0-stemmerweka.core.stemmers.NullStemmer-stopwords-handlerweka.core.stopwords.Null-M1-tokenizerweka.core.t
Instances:    296
Attributes:   2185
              [list of attributes omitted]
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.44 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         295               99.6622 %
Incorrectly Classified Instances         1                0.3378 %
Kappa statistic                          0.9901
Mean absolute error                      0.0129
Root mean squared error                  0.0581
Relative absolute error                  3.7619 %
Root relative squared error             14.0265 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.015    0.996      1.000    0.998      0.990    1.000     1.000     malware
                 0.985    0.000    1.000      0.985    0.992      0.990    1.000     0.999     nonmalware
Weighted Avg.    0.997    0.012    0.997      0.997    0.997      0.990    1.000     1.000

=== Confusion Matrix ===

   a   b   <-- classified as
 231   0 |   a = malware
   1  64 |   b = nonmalware
```

RandomTree: 99.3243



```
RandomTree
==========

?? ?? ?? ?? ?? < 0.5
|   00 10 00 00 00 < 0.5
|   |   6F 67 72 61 6D < 0.5
|   |   |   00 FF FF 00 00 < 0.5
|   |   |   |   FF FF FF FF FF < 0.5 : nonmalware (1/0)
|   |   |   |   FF FF FF FF FF >= 0.5 : malware (9/0)
|   |   |   00 FF FF 00 00 >= 0.5 : nonmalware (3/0)
|   |   6F 67 72 61 6D >= 0.5 : nonmalware (25/0)
|   00 10 00 00 00 >= 0.5 : nonmalware (36/0)
?? ?? ?? ?? ?? >= 0.5 : malware (222/0)

Size of the tree : 11

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         294               99.3243 %
Incorrectly Classified Instances         2                0.6757 %
Kappa statistic                          0.9803
Mean absolute error                      0.0068
Root mean squared error                  0.0822
Relative absolute error                  1.9645 %
Root relative squared error             19.8534 %
Total Number of Instances              296

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.996    0.015    0.996      0.996    0.996      0.980    0.990     0.995     malware
                 0.985    0.004    0.985      0.985    0.985      0.980    0.990     0.973     nonmalware
Weighted Avg.    0.993    0.013    0.993      0.993    0.993      0.980    0.990     0.990

=== Confusion Matrix ===

   a   b   <-- classified as
 230   1 |   a = malware
   1  64 |   b = nonmalware
```

MODULE 2: (STATIC ANALYSIS TO FIND THE TYPE OF VIRUS)

Data set: The Data set which was considered above contains all malware files along with the type of virus that the file has got injected with it. In the data set there are totally nine classes of viruses, namely:

1. Ramnit
2. Lollipop
3. Kelihos_ver3

4. Vundo
5. Simda
6. Tracur
7. Kelihos_ver1
8. Obfuscator.ACY
9. Gatak

Pre-processing steps:

1. The pre-processing steps are the same as for the above module, i.e generating n grams using String to word vector.

Training the model:

1. Once the data set being pre-processed, the classification algorithms are applied.

Evaluating:

1. Once the model being trained, the hex dump which was found to contain virus in module 1 is applied to this trained model.
2. The model, then assigns a class to this hex dump i.e. the type of virus that has injected to it.

MODULE 3: (DYNAMIC ANALYSIS)

Dynamic Analysis: The features are extracted from executable during execution.

Data set: http://www.cs.unm.edu/~immsec/data/live-sendmail.html

Features used: System call sequences.

SYSTEM CALL SEQUENCES:

An application programming interface (API) is a source code interface that an operating system or library provides to support requests for services to be made of it by computer programs. For operating system requests, API is interchangeably termed as system calls. An API call sequences captures the activity of a program and, hence, is an excellent candidate for mining of any malicious behaviour.

Data set description: The data set consist of both malware and non-malware

Process' system call sequences generated using Linux commands like send mail, login, etc., The system calls are represented as numbers and hence the sequences as string of numbers.

Pre-processing:

1. All the non-malwares process' system call sequences are stored in a file named malware.
2. Generate nonmalware.arff using texttodirectory loader.
3. Similarly generate another malware. Arff for malware process' system call sequences.

Training the model:

1. Load the arff into Weka and apply cluster analysis.

Testing the model:

1. Supply the malware. Arff as the test set and run the model.
2. Detect the outliers as the malwares' system call sequences will be different from the non-malwares' system call sequences.

Measurement metric:

1. More the ratio of number of outliers and total number of malware files, more is the accuracy of the trained model.

MODULE 4: (DYNAMIC ANALYSIS)

Assumption: The malware sequence is assumed to be of fixed length. **Algorithm:**

1. Taken the malware system call sequences and converted to spmf format.

2. Applied the TKS Algorithm and obtained the top malware sequences.

3. Now generated the same length sequences as that of the topk sequences.

4. Each such sequence is compared with all the top k sequences. If match is found then the count of the process is incremented.

5. More the count, more the probability of being it a malware process.

MODULE 5: (DYNAMIC ANALYSIS)

Assumption: The Malware system call sequences are complete and the attacker does intrusion without wasting time.

Algorithm:

1. Taken a process' system call sequence 's'.
2. For each Malware system call sequence s_i , check whether all calls are present or not in 's'
3. If yes, check whether the same sequence of calls is present or not.

4. Else, check with other Malware system call sequence until no malware sequence available.
5. If yes, count the number of calls in between the malwares' calls.
6. Else,'s' is non-malware.
7. More the count, less the probability of it being a malware.

MODULE 6: (DYNAMIC ANALYSIS)

Assumption: The Malware sequences are complete and each genuine activity's sequences are complete.

Algorithm:

1. Taken a process' system call sequence's'.
2. For each Malware system call sequence s_i , check whether all calls are present or not in 's'
3. If yes, check whether the same sequence of calls is present or not.
4. Else, check with other Malware system call sequence until no malware sequence available.
5. If yes, append all the calls (say new sequence 'u') in between the malwares' calls.
6. Else,'s' is non-malware.
7. For each activity 'a_i', generate n grams of 'u' where 'n' is the length of each activity.
8. Check whether each gram is equal to the 'a_i'.
9. If yes increase the count.
10. Check with other activities until there are no activities left.
11. More the count, less the probability.

THEORETICAL MODULES:

MODULE 1: (INTEGRATION OF STATIC AND DYNAMIC ANALYSIS)

Explanation: Both Static and Dynamic Analysis techniques are complementary to each other and.Static Analysis does the complete code coverage of a program whereas Dynamic Analysis does less code coverage. But Static Analysis won't be able to provide sufficient information compared to Dynamic Analysis. They have their own advantages and disadvantages. This led to the belief that integrating both the techniques would lead us to better accuracy when compared to implementing them individually.

For an executable, if it contains malicious code in it, then it would be waste of time to perform Dynamic Analysis (Executing the program in Virtual Machine, extracting the features and then comparing to the model trained). Instead Static Analysis can be applied and it can be detected as malicious or not.

So, first Static Analysis can be applied and then the Dynamic Analysis.

Steps:

1. Given a sample executable to test.
2. First Static Analysis is applied.
3. If found to be virus, then done.
4. Else, apply Dynamic Analysis.
5. Extract features and test with trained model.
6. If found to be malicious, done.
7. If not, the executable is free from viruses.

MODULE 2: (USING INTEL'S PIN TOOL TO DETECT BASIC ROP Attacks)

PINTOOL: It is API to extract features from an executable during its execution.

Programs are written on top of this API to extract the desirable features**.**

How ROP Attack came into picture:

Attack 1: Generally, the formal way of attacking is to inject malicious code(,i.e Payload) into any program's address space and perform malicious activity using that injected code. (This is achieved through Buffer Overflow Attack).

Response: To overcome this, many operating systems provide the NX bit enabled, i.e, the programs' address space is either executable or writable but not both. So, even if the buffer is overwritten with payload, it will not be executed with this NX bit enabled.

Attack 2: With this attackers, had only one way to perform malicious activity, i.e. using the instructions present in the DLL (Dynamic Linked Library).But using those instructions, the extent of intrusion that can be done was limited.

Response: ASLR (Address Space Layout Randomisation). The addresses of the libraries were randomised. And the attackers had to apply brute force approach to find the instructions of the library.

Attack 3: Now comes ROP, which just uses small instructions sets already available in a program. Those small instructions are named gadgets for ROP. The Gadgets contain mostly 3-5 instructions and would end with a return instruction and hence the name Return Oriented

Programming (ROP). Now all those Gadgets are arranged in such a way that upon execution of sequence of Gadgets would lead to a malicious activity. In this way the attackers can perform any malicious activity as desired.

Our Approach to detect ROP:

Reports and how ROP attacks programs show that the number of return instructions executed in an ROP attack and normal program execution vary greatly.

Steps:

1. Perform normal program execution by giving normal inputs to the program and find out the number of return instructions using a program written on top of PINTOOL API.
2. Now perform execution of the program giving ROP Payload and find out the number of return instructions.
3. If the count is much greater than that of the normal executions. Count the number of instructions in between the return instructions considering a sliding window using another program written on top of PINTOOL.
4. If there is a sudden decrease in the number of instructions and the decrease continuous for more than a threshold, then it is more probable that the given input is a ROP payload and performs a malicious activity.
5. Else, it is less probable to be ROP payload.

MODULE 3 (DYNAMIC ANALYSIS USING CUCKOO SANDBOX)

Different run time features can be extracted from the reports generated by Cuckoo Sandbox after a sample is given to the sandbox. The sandbox executes the program in a virtual Machine and submits reports to the host machine in a JSON format.

Each Analysis report contains the execution trace of a program which includes process tree, file and memory changes, memory dump and mutexes, etc.,

From the JSON format, different kinds of run time features can be extracted and can be used for training our models.

NOVELTY:

❖ If module 1 or 2’s implementation of the Theoretical models explained above give accuracy, then it leads to Novelty.

CONCLUSION

Virus detection has been done using two different features with two different data sets. The accuracies of the different techniques applied are compared for the two features. Explored different techniques how an attack is done and different tools for Dynamic Analysis**.**

FUTURE WORK

- All the Theoretical Models explained above will be implemented.
- We will be building a model to detect ROP with better accuracy.
- Also after detecting the virus and virus' type, we will try to delete that virus and ensure normal execution to the user.
- If everything above are done, will build a anti-virus software for windows OS.

TIMELINE

NO OF DAYS SPENT	TASK
2	**Finding, downloading and extracting the data set for hex dump**
7	**Pre-processing**
7	**Training and Testing (module 1 completed)**
7	**PINTOOL AND ROP**
1	**Finding and Downloading the dataset for system call sequences**
2	**Module 2 completed**
4	**Module 3 completed**
3	**Module 4 completed**
2	**Module 5 completed**
3	**Module 6 completed**
3	**Cuckoo Sandbox installation and setting up environment**
1	**For Novelty**

Total Number of days : 42 (Approx)

REFERENCES:

1. **Base paper 1:** **http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.7848&rep=rep1&type=pdf**
2. **Base paper 2:** **http://stars.library.ucf.edu/cgi/viewcontent.cgi?article=4709&context=etd**
3. **ROP : http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html**
4. **PIN TOOL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool**
5. **CUCKOO SANDBOX: https://www.cuckoosandbox.org/**

CODES:

MODULE 5:

Java Code :

```
import java.io.*;

import java.util.*;

public class Test

{

public static void main(String args[])

{

String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();

String content2 = new Scanner(new File("filename")).useDelimiter("\\Z").next();

String[] words = content.split("\\s+");

String[] words2= content2.split("\\s+");

int found;

for(int i=0;i<words.length;i++)

{

found =0;

   for(int j=0;j<words2.length;j++)

   {

   if(words[i].equals(words2[j]))

    {

    found=1;
```

```
     }
  }
if(found ==0)
break;


}
if(found ==0)
{
System.out.println("all the system call sequences are not present");}
else
{
prev = -1
cur = 0;
for(int i=0;i<words.length;i++)
{
  for(int j=0;j<words2.length;j++)
    {
      if(words[i].equals(words2[j]))
       { cur = j;break;}}
if(cur<prev)
{found =0;break;
}
prev=cur;}
if(found==0)
{
System.out.println("all the calls are presentn but the sequence is missing");
}
```

```
else
{
m=0;
count =0;
for(int i=0;i<words.length;i++)
{
while(!(words2[m].equals(words[i])))
{
++count;

++m;
}
++m;
}
}






}}
```

MODULE 6:

Java Code :

```
import java.io.*;
import java.util.*;
public class Test2
{
```

```
public static void main(String args[])
{
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
String content2 = new Scanner(new File("filename")).useDelimiter("\\Z").next();
String[] words = content.split("\\s+");
String[] words2= content2.split("\\s+");
int found;
for(int i=0;i<words.length;i++)
{
found =0;
   for(int j=0;j<words2.length;j++)
   {
   if(words[i].equals(words2[j]))
    {
    found=1;
     }
  }
if(found ==0)
break;
}
if(found ==0)
{
System.out.println("all the system call sequences are not present");
}
else
{
prev = -1
```

```
cur = 0;

for(int i=0;i<words.length;i++)

{

  for(int j=0;j<words2.length;j++)

    {

      if(words[i].equals(words2[j]))

       {

       cur = j;break;  }}

if(cur<prev)

{found =0;break;

}

prev=cur;}

if(found==0)

{

System.out.println("all the calls are presentn but the sequence is missing");

}

else

{

m=0;

count =0;

for(int i=0;i<words.length;i++)

{

StringBuffer sb = "";

while(!(words2[m].equals(words[i])))

{


sb.append(words[m]);
++m;

}

if(sb.length()!=0)

{

System.out.println(sb);

}

++m;

}

}}}
```