Malware Detection using Ensemble Learning

In this round, we implemented malware detection using ensemble learning.

Installation

1. (Optional) Set up a virtual environment On linux

```
python3 -m venv env
source env/bin/activate
```

2. Install the necessary libraries (sklearn, numpy and ujson)

```
python3 -m pip install -r requirements.txt
```

Usage

Full specification

```
usage: MalwareDetection.py [-h] [--train] [--validate] [--predict]
                           [--split [SPLIT]]
                           [--model [{all,string,structure,dynamic}]]
                           [--output [OUTPUT]] [--choose [n]]
                           [inputs [inputs ...]]

Detect malware using machine learning

positional arguments:
  inputs                Directories to search for input files

optional arguments:
  -h, --help            show this help message and exit
  --train               Use the input data to train the model
  --validate            Use the input data to validate the model
  --predict             Predict output on the input data
  --split [SPLIT]       (only for train) train-total data ratio to use
  --model [{all,string,structure,dynamic}]
                        which model to use (default is all)
  --output [OUTPUT]     (only for predict) save the output to a file, use
                        stdout to print
  --choose [n]          Use only n randomly sampled files from the input
```

Examples

1. Use the ensemble of all 3 models to predict on a data set.

```
python3 MalwareDetection.py data_set/
```

2. Display the output on stdout instead of csv.

```
python3 MalwareDetection.py --output stdout data_set/
```

3. Read input files from many directories at once.

```
python3 MalwareDetection.py data1/ data2/ data3/ data4/
```

4. Train all the models on a given data set (labels deduced automatically).

```
python3 MalwareDetection.py --train data_set/
```

5. Choose the train-test split to be 70:30 during training.

```
python3 MalwareDetection.py --train --split 0.7 data_set/
```

6. Train only on 1000 examples randomly sampled.

```
python3 MalwareDetection.py --train --choose 1000 data_set/
```

7. Use only the string model for operations.

```
python3 MalwareDetection.py --model string ...
```

8. Validate the accuracy metrics of a model again.

```
python3 MalwareDetection.py --validate validation_set/
```

Note that training a model will overwrite the previous trained model. In any case if you want to restore to default pre-trained model run the following commands.

```
 cp dynamic_analysis/model/model.sav.back dynamic_analysis/model/model.sav
cp string_analysis/model/model.sav.back string_analysis/model/model.sav
cp structure_analysis/model/model.sav.back structure_analysis/model/model.sav
```

IMPORTANT: When you ensemble (that is no --model) make sure to have atleast one file of each type (String, Structure and JSON) otherwise an empty feature matrix is passed to models which may cause the program to crash. Note again that if you are doing this you most likely want to run the program using the `--model` flag. See example 7 for usage.

Model Description

We had access to three kinds of data for a binary:

1. String dump from the file which contains printable characters in `Strings.txt`
2. Structure information from the PE file in `Structure_Info.txt`
3. Dynamic malware analysis data from the Cuckoo Sandbox in `<file_hash>.json`

We trained three separate models, one for each type of data. Whenever all three types of data are available, we use a consensus based classification using predictions from each of the three models. Otherwise, we revert to using a single model (decided on the basis of confidence in accuracy, from among models corresponding to available data).

Each of these models was trained as a multiclass classifier (classes being 'benign', 'backdoor', 'trojan', 'trojandownloader', 'trojandropper', 'virus', 'worm').

A brief summary of all the models is produced below, refer to the observations document for a more detailed insight.

String analysis

We recognized that the `Strings.txt` file contains the dump of readable character strings from the binary. We constructed a frequency table of filtered strings in the file. Then we used a FeatureHasher to vectorize it to a lower-dimension feature vector. A suitable dimension was chosen for giving the best results without compromising on efficiency. Finally, a RandomForestClassifier was trained using the feature matrix obtained from the training examples.

Structure analysis

We recognized that the `Structure_Info.txt` files contain a human readable dump of executables in PE(portable executable) format, with the actual code summarized into entropy and hashes. Our strategy for feature extraction was:

1. We take all the 'key: value' pairs with numerical values in each section (eg. `[IMAGE_FILE_HEADER]`). We prefix the section name (eg. `CODE`, `.data`, `.rsrc`, etc) to the 'key' to get the feature name and 'value' is used as the feature value.
2. For feature names which appear multiple times, eg. for entropy in many sections of the same type, we compute the repetition count, mean, min and max as features.
3. We include the section indents in the feature names to capture nesting, especially in resource related sections.
4. We use presence of all flags in the PE as binary valued features.
5. We use the presence of .dll files as features whose values are the counts of the symbols imported from .dll files.

For feature selection, we used a VarianceThreshold transform which removes features showing little variation across training examples.

Finally, we used a DictVectorizer to vectorize the feature dicts and trained a RandomForestClassifier on the resulting feature matrix for the training examples.

Dynamic analysis

For analysis of the `<file_hash>.json` file, we chose the following features:

1. Duration.
2. Maximum of severity from the present signatures.

3. Frequency of the following types of network requests: udp, http, irc, tcp, smtp, dns, icmp.
4. The number of domains and hosts contacted.
5. Frequency of the following types of API categories: noti, certi, crypto, exception, file, iexplore, misc, netapi, network, ole, process, registry, resource, services, syn, system, ui, other.
6. Frequencies of any kinds of API calls.

An insight for choosing these features in particular is highlighted in the observations document. We used FeatureHasher to vectorize the API calls (because of the unknown and large number of possibilities) and explicitly vectorized the other hand-picked features. We used the concatenation of these two feature matrices to get the final feature matrix and trained a RandomForestClassifier on it for the training examples.

Ensemble Model

To ensemble all the 3 models described above - StringModel, StructureModel and DynamicModel we use the following algorithm.

1. If for a file hash, all of 3 models yield a prediction (that is, all 3 String.txt, Structure_Info.txt and Json are available) then the classical majority voting technique is used for consensus. That is prediction is taken to be that agreed upon by the most number of models.
2. If majority voting is not possible (<3 models yielded prediction) then the prediction from most precise model is taken to be the truth. In our case we found that the order of precision is dynamic > string > structure.

Primarily the ensemble algorithm increases robustness of the classifier. Because even if one of the model errs (which may be due to parsing errors or missing data or an outlier), the prediction obtained from other two is likely to be sufficient to reach consensus. Further this naturally also leads to increased accuracy which is reflected in our results obtained. With this final model we were able to reach accuracy as high as > 99.95% sometimes even 100% as tested by first training on 75% of the data and then testing on remaining 25%.

Summary

String Model

Feature Extraction: using a custom build parser, extracted the frequency table of *useful* keywords from String.txt file.

Feature Reduction: Using the feature hashing trick (sklearn's FeatureHasher) reduce the dimensionality to a fixed (and adjustable) sized feature vector (length chosen `7000`).

Model: Random Forest Classifier

Accuracy: 98.1%

F1 Score: 98.1%

Structure Model

Feature Extraction: prominent key-value pairs + hand-picked information (entropy ...)

Feature Reduction: VarianceThreshold and DictVectorizer

Model: Random Forest Classifier

Accuracy: 98%

F1 Score: 98%

Dynamic Model

Feature Extraction: hand picked (duration, severity, network requests, ...) + frequency table of API calls from all processes.

Feature Reduction: Using feature hashing trick on frequency table of API calls.

Model: Random Forest Classifier

Accuracy: 99.88% - 100% on 75-25 train-test split.

F1 Score: 99.88% - 100%

Results from training and testing on given data

From the output of (headings added for clarity) :

```
python3 MalwareDetection.py --train .
```

```
 train_test split: 75% : 25%

--- model for Strings.txt ---
files in training set: 7466
files in test set: 2489
feature extraction time for training set: 92.8 s
training time: 11.3 s
feature extraction time for test set: 27.3 s
testing time: 1.2 s
accuracy: 0.9807
f1 score (micro): 0.9807
precision score (micro): 0.9807
recall score (micro): 0.9807
f1 score (macro): 0.9807
precision score (macro): 0.9805
recall score (macro): 0.9811

--- model for Structure_Info.txt ---
files in training set: 7464
files in test set: 2488
feature extraction time for training set: 69.4 s
training time: 16.3 s
feature extraction time for test set: 22.3 s
testing time: 1.7 s
accuracy: 0.9803
f1 score (micro): 0.9803
precision score (micro): 0.9803
recall score (micro): 0.9803
f1 score (macro): 0.9802
precision score (macro): 0.9801
recall score (macro): 0.9805

--- model for <file_hash>.json ---
files in training set: 7464
files in test set: 2489
feature extraction time for training set: 326.3 s
training time: 5.4 s
feature extraction time for test set: 109.2 s
testing time: 0.4 s
accuracy: 0.9987
f1 score (micro): 0.9987
precision score (micro): 0.9987
recall score (micro): 0.9987
f1 score (macro): 0.9987
precision score (macro): 0.9987
recall score (macro): 0.9988

--- testing on ensemble ---
total extraction time for test set: 102.6 s
testing time: 1.2 s
accuracy: 0.9991
f1 score (micro): 0.9991
precision score (micro): 0.9991
recall score (micro): 0.9991
f1 score (macro): 0.9991
precision score (macro): 0.9991
recall score (macro): 0.9992
```