Endpoint Telemetry for Malware Detection

Siddhartha Jagannath
Department of Computer Science, Stanford University
kachapah@stanford.edu

Abstract

Traditionally, malware detection is performed using a set of heuristics that are predetermined as indicators of malware in a file. Thus the anti-virus software evaluates input files based solely on those heuristics. However, as malware becomes more advanced, current methods of detection that use heuristics are constantly outpaced by the malware they are fighting. This is a problem that the IT security industry is still trying to tackle to this day. In this project, I will present a data-driven approach to solving the malware detection problem for portable executable files.

1 Introduction

The goal of this project is to apply state-of-the-art machine intelligence techniques to understand assembly opcodes executed by binary files and make predictions about whether those sequences of opcodes plus additional features about the files make up a malicious file. Some of these additional features include things like checksum values (a common feature of files used as an indicator of non-malware). PE files also have multiple section headers containing different information about the file, when it was created, what libraries it imports, what data it exports, what exactly its visible contents are and much much more. We can extract copious amounts of data from these sections - things like different section entropies, imported symbols, checksum values, etc. All of these features are also powerful in determining whether a file is malicious or not.

My contributions in this paper are two-fold. First, I applied machine learning techniques to strictly numerical features that I extracted from the PE Files in my dataset. The goal here was to see how well I could predict whether a file was malicious or not only using such features. I then applied machine learning and deep learning frameworks on opcode sequences to see if that performed better for the malware classification/detection task.

Finally, my results are as follows. For the numeric features, I set a baseline on the test set of 18.09%. My neural network classifier on numeric features acheived a test set accuracy of 85.58%. For the opcode sequence data, my LSTM classifier set a baseline performance of 78.44 % accuracy on the test set. My Gradient Boosting Machine model acheived a test set accuracy of 89.78 %.

2 Related Work

There have been several key papers written on this particular topic of malware detection:

1. **Malware Detection Using Machine Learning** paper makes use of Random Forest and acheives a state-of-the-art 99.7 % accuracy on the test set, with a dataset of over 100,000 files. This paper is the basis for my use of decision trees-based method as these seem to produces the best results for this task. [7]

2. **Android malware detection based on system call sequences and LSTM** is a novel detection method based on deep learning is proposed to distinguish malware from trusted applications. Treats one system call sequence as a sentence in the language and construct a classifier based on the LSTM language model. I took inspiration from this model's use of LSTMs and their treatment of opcode sequences as words in a traditional language model [3].s
3. **Opcode-Sequence-Based Malware Detection** is a model that proposes to predict malware based on opcode frequency. While the model does make use of opcode sequences, it only looks at frequency and does not account for order of opcodes.[2]
4. **A System-call Behavior Language System for Malware Detection Using A Sensitivity-based LSTM Model** presents a slightly different approach to opcode sequence-based by assigning a sensitivity score to different opcodes - however, while powerful, this approach was not possible given the nature of my dataset. [1]
5. **Malware Detection with LSTM using Opcode Language**. proposes a novel, state-of-the-art model for malware prediction. The paper proposes a two-stage LSTM model for malware detection, which use two LSTM layers and one mean-pooling layer to obtain the feature representations of opcode sequences of malwares. Despite receiving an average AUC score of 0.987, the data set used was relatively small («1000 files) with there being a heavy bias towards malware files (a 90-10 split between malware and benign files, respectively).

3 Methods & Dataset/Features

My approach to this problem is multi-pronged. Malware detection is an ever-evolving field and the application of AI to it in recent years has yielded mixed results. As mentioned, I plan on implementing several approaches to detecting malware, using techniques learned in the CS229 course. At a higher level, there are two ways to approach the malware detection problem. We can use numerical data extracted from the files, or use language-based modelling techniques to analyze things like opcode sequences generated, etc. We first look at using numerical features to perform classification with several techniques.

Data: . For this project, I will be using a data set that consists of about 4151 portable executable files, 2195 of which are classified/labelled as "good" or free of malware and the other 1956 are labelled as "bad" or containing malware. These files contain a very broad spectrum of different executable code for various different task. Thus, what ever model is developed with this data should ideally be quite generalizable.

Numeric Approach: As mentioned, in this section I examine the performance of classification using only numerical features of the files. These features were selected because of their use in heuristics-based malware detection. These include section entropies (size of a section in the file), imported symbols (what the file is importing), checksum values (a integer value calculated by the compiler after it builds the executable), imphash values (a hash over the imported functions by PE file), etc. Here is an example of what some of these features look like.

```
checksum  :  1950167
imphash  :  884310b1928934402ea6fec1dbd3cf5e
num_sections  :  8
section_entropy_ODE  :  6.631765876950794
section_entropy_tls  :  0.0
IMAGE_DIRECTORY_ENTRY_IMPORT_size  :  2384
IMAGE_DIRECTORY_ENTRY_IMPORT_rva  :  53248
```

In order to analyze this data I deployed two separate approaches:

1. The first, rather rudimentary approach, was to create a logistic regression classifier to set a baseline for performance.

2. After I have set the baseline, I needed to decide on a set of numerical features and train a neural network to perform a binary classification task on those features. The details on my experimentation is provided in the **Experiments** section

Language Model Approach: The second, more detailed approach is to build a language model based on actual opcode sequences generated by the PE files. Often the exact opcodes and the order in which they are called can be indicative of whether or not a file is malicious. In other words, if a file is doing something that it's not supposed to, it is often quite easy to tell by dissecting what opcodes are being called, in what order. In addition to building a classifier, I wanted to verify this hypothesis that order if opcodes matter. To establish this, I ran two separate models:

1. First, I set a baseline using a Long short-term memory (LSTM) network. For the LSTM, instead of using n-grams, the data for each file included a list of opcodes that occur in the PE file. Thus the ordering of the opcodes was not captured. The idea here was to set a baseline without ordering, to gauge how much capturing the ordering does affect accuracy.
2. Once the baseline is set using the LSTM, I used XGBoost, a gradient boosting framework that uses decision trees. For the Gradient Boosting, I formatted the opcode sequences as n-grams to capture the inter-sequence relationships that these opcodes have. Opcodes are after all not independent, they represent a series of sequential actions.

For the sake of illustration, below are some examples of opcode sequence n-grams that are fed into the XGBoost model.

```
opcode 5-grams  :
[['or', 'jae', 'jb', 'outsb', 'add'],
['jae', 'jb', 'outsb', 'add', 'add'],
 ['test', 'dec', 'mov', 'dec', 'mov'],
 ['jne', 'mov', 'xor', 'test', 'jle'],
 ['mov', 'xor', 'test', 'jle', 'sub']]
```

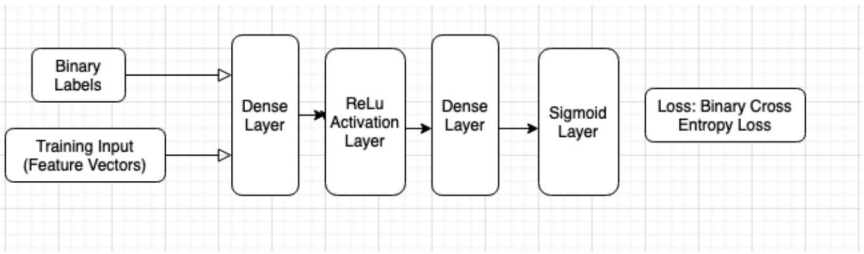
4 Experiments

4.1 Tools

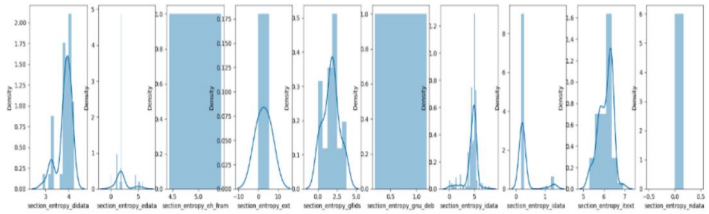
There exist numerous technologies to capture endpoint telemetry statistics (e.g. procmon and sysmon). I am using the `pefile` library to help me extract features from the files in my dataset, as it is python compatible. The pefile library allows me to extract all the numerical features as well as n-grams. I am using a combination of the PyTorch library, Tensorflow and XGBoost to create my data preprocessing and modelling pipelines.

4.2 Experimental details

Numerical Feature-based Models: When it came to implementing a numeric-based approach to this problem, there is a fundamental issue with the data I had. Because each of my files contain binaries for an array of different functionalities, their PEFile structures are very different. The result is that when I extract the features from them, I end up with vastly sparse feature vectors. Without filtering, a dataset of PE files can have anywhere between 6000 and 10,000 combined features, given that I am using a one-hot/indicator-style encoding to represent many of the features in the feature vector. This is because every PE file is different and they often have very unique attributes. This means that the diversity of features is very high, resulting in very sparse feature vectors.This is the classic "p » n" problem in Machine Learning. Most ML classifiers and models assume that "p « n", so this was not ideal. And it reflected in my model's peformance. I was getting test-set accuracy for my logistic regression model of around 0.18, which is not optimal.



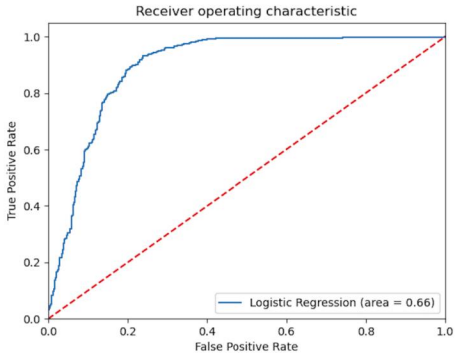
Thus, I had to experiment with methods for feature selection. I started by plotting feature distributions across the data set for the most common features as a way of manually selecting them(pictured below).



However, this proved rather inefficient and I decided to instead experiment with sklearn's SelectKBest feature selection library along with Chi Squared statistics to select the best 100 features from the data set. This drastically improved my model performance, as we can see in the table below. For logistic regression baseline, I implemented a rudimentary model using sklearn's LogisticRegression() library. For the neural network classifier, I used several dense layers coupled with a ReLU activation layer and evaluate the loss using binary cross entropy. Here we use the F1 score as a performance measure, where:

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

Test Set Results		
Model Iteration	Accuracy	F1
Logistic Regression w/out SelectKBest	18.89	20.09
Logistic Regression w/ SelectKBest	58.72 (+39.83)	53.51 (+33.42)
Neural Network Classifier	85.56 (+26.84)	82.77 (+29.26)



AUC-ROC curve for Final Neural Network Classifier

Language-Based Models Approach: As mentioned in earlier sections, my appraoch with this project was to test both numeric-based features as well as word-based features. In the **Approach**

section we covered how the language data is generated. To recap, we disassemble the PE file, extracting the opcode sequences that the file executed during runtime. Opcode sequences are an established heuristic for determining whether a file is malicious. First I ran the baseline LSTM with the set of opcodes present for each PE file. For example, if an opcode sequence was ['or', 'jae', 'jb', 'outsb', 'add', 'jae', 'jb', 'outsb', 'add', 'add'], the opcode set passed into the LSTM would be ['or', 'jae', 'jb', 'outsb', 'add']. The reason for using an LSTM instead of an RNN is that these files contain 100s of different opcodes and thus LSTMs are more appropriate given that they capture long term dependencies. I then ran a model built on XGBoost, gradient boosting machine framework. Gradient Boosting is a technique that utilizes decision trees for supervised learning tasks. The reason I chose this model is that decision-tree-based models have proven rather effective for the task of opcode n-gram analysis.

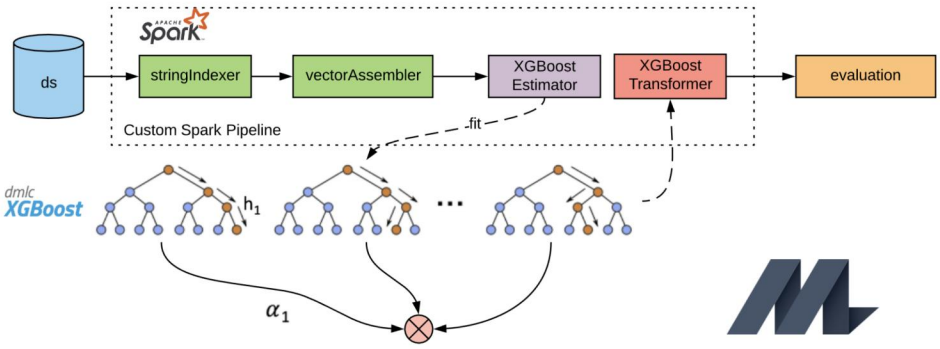


Figure: XGBoost architecture [6]

Test Set Results		
Model Iteration	Accuracy	Loss
Long short-term Memory	78.44	0.3661
Gradient Boosting Machine	89.64	0.234

Figure: Language Model Test-set results [6]

5 Analysis/Conclusion

In this paper, I introduce a two-part malware detection pipeline that takes two distinct approaches to the task. For the first approach, using numeric features, we were able to set a logistic regression baseline and then beat that baseline with a neural network classifier by about 30% in predictive accuracy. For the second language mode-based approach using opcode sequences, my baseline performed quite well with a 78.44 % accuracy on the test set. We were still able to outperform the baseline using order-aware representations of the opcode sequences, getting the highest accuracy across all the experimentof 89.64 %. This result seems to demonstrate that opcode sequences are more powerful in detecting malware when compared to numeric features. Future work would involve indepth ablation studies, as well as focusing on developing further opcode sequence-based learning frameworks, with more sophisticated language models involving concepts such as attention.

6 References

[1] Xie, Wenqi & Xu, Shengwei & Zou, Shihong & Xi, Jinwen. (2020). A System-call Behavior Language System for Malware Detection Using A Sensitivity-based LSTM Model. 112-118. 10.1145/3403746.3403914.

[2] Xiao, X., Zhang, S., Mercaldo, F. et al. Android malware detection based on system call sequences and LSTM. Multimed Tools Appl 78, 3979–3999 (2019). https://doi-org.stanford.idm.oclc.org/10.1007/s11042-017-5104-0

[3] Mathew J., Ajay Kumara M.A. (2020) API Call Based Malware Detection Approach Using Recurrent Neural Network—LSTM. In: Abraham A., Cherukuri A., Melin P., Gandhi N. (eds) Intelligent Systems Design and Applications. ISDA 2018 2018. Advances in Intelligent Systems and Computing, vol 940. Springer, Cham. https://doi.org/10.1007/978-3-030-16657-1

[4] Elkhawas, Amr & Abdelbaki, Nashwa. (2018). Malware Detection using Opcode Trigram Sequence with SVM. 1-6. 10.23919/SOFTCOM.2018.8555738.

[5] Gill & Mandil, Malware Benign File Classification using Machine Learning Deep Learning

[6] Wang, Holler, et al Productionizing Distributed XGBoost to Train Deep Tree Models with Large Data Sets at Uber (https://eng.uber.com/productionizing-distributed-xgboost/)

[7] Kumar A. et al. (2020) Malware Detection Using Machine Learning. In: Villazón-Terrazas B., Ortiz-Rodríguez F., Tiwari S.M., Shandilya S.K. (eds) Knowledge Graphs and Semantic Web. KGSWC 2020. Communications in Computer and Information Science, vol 1232. Springer, Cham. https://doi-org.stanford.idm.oclc.org/10.1007/978-3-030-65384-2$_5$