Machine Learning Based Malware Detection Tool

Final Project Report, Introduction to Computer Security

Hsin-Chen Hu
hhu@brandeis.edu

May 2025

Overview

The main goal of this term project is to develop a general sense [8][9] of how malware works and attack the computers, as well as build a machine-learning-based malware detection tool and to evaluate the performance of different machine learning models for malware detection.

This project develops an malware analysis using machine learning to detect and classify malicious software. It gathers datasets of malware samples, extracts features such as API calls, opcodes, file metadata, and behavioral patterns, and trains classification models.

Understanding the Malware

Malware, or malicious software, encompasses a wide range of programs designed to infiltrate, damage, or gain unauthorized access to systems. Types of malware include viruses, worms, trojans, ransomware, spyware, and rootkits, each with its own methods of infection and goals.

To develop an effective detection tool, it's essential to gain a foundational understanding of different malware types, how they operate, and the ways in which they attack systems. This knowledge informs the selection of features to be extracted and used in training machine learning models.

For instance, ransomware is a type of malware that encrypts a victim's files or locks access to their system, rendering data and applications inaccessible. Once the ransomware has taken hold, it typically presents a ransom note demanding payment in exchange for the decryption key or the promise to restore access. Ransomware can spread through various vectors, including phishing emails, malicious attachments, drive-by downloads, or exploiting software vulnerabilities.

Modern variants of ransomeware could also exfiltrate data before encryption, threatening to leak it publicly if the ransom is not paid. Ransomware behavior is often characterized by rapid file access and modification, encryption routines, changes to system settings, and suspicious network activity. These behavioral patterns make it possible to extract relevant features. For example, sudden spikes in CPU usage, unusual file system access, or encrypted file extensions. These features can be used to detect malware in machine leaning models, which is one of the main goals in this term project.

Data Collection

A well-labeled, diverse dataset is important and beneficial for building reliable malware detection models. During the research phrase term project, I explored several open-source resources to find appropriate data.

EMBER Dataset [1]
The Elastic Malware Benchmark for Empowering Researchers Dataset is a collection of features from PE files that serves as a benchmark dataset for researchers. One of the main advantages of EMBER is that it abstracts away the need for direct binary parsing and reverse engineering by providing vectorized and preprocessed features in JSON and CSV formats.

For this term project, I had attempted to use the EMBER dataset for training the malware detection models. However, due to environment setup issues [6] and compatibility challenges, I turned to an alternative dataset. Despite this, EMBER remains one of the most robust and popular datasets for malware research and could be used for future work.

VirusShare Dataset [3]
VirusShare is also a very popular dataset for research. Unlike like EMBER, VirusShare primarily provides raw binary files rather than pre-extracted or labeled features. This means that while it is extremely valuable for collecting diverse malware specimens, it requires substantial preprocessing.

Kaggle [4]
Kaggle is a good platform for the data science community. It offers a variety of datasets and resources. However, the quality and suitability of the data can vary depending on the needs of projects. For example, as I was looking for a dataset for this term project, I came across one dataset that seemed promising at first, but the entries in that dataset were unfortunately all labeled as "malware," making it unusable for training the models.
On the other hand, one positive aspect of Kaggle is its active community. Users and researchers can share their thoughts, review datasets, and ask questions, which greatly enhances the collaborative learning experience.

The Dataset in This Project[5]
To train and evaluate multiple machine learning models, selecting appropriate datasets and features to optimize their performance is an important step. The dataset we are using in this project is a smaller scaled, labeled dataset found on GitHub by Burak Ergenc.

This dataset was chosen because it contains system-level metrics, extracted from malware process executions over time. The features have already been vectorized, and each entry is labeled as either “malware” or “benign” in the classification column. This structure makes it suitable for training a supervised machine learning model.

The features in this dataset includes:

- hash
  The hash column is the unique identifier of the malware binary, it groups observations belonging to the same sample.
- millisecond
  This column is the timestamp. It allows temporal tracking of behavior over time.
- classification
  This is the label column for supervised learning. It labels the entries to “malware” of “benign”. It’s the target variable for classification.
- state
  This column is the a process state flag (e.g. running, sleeping). Some states may be more common in malicious processes, which would help classification.
- usage_counter
  The count of how many times a resource was used. Unusual usage patterns may signal malicious behavior.
- prio, static_prio, normal_prio
  Process scheduling priorities. Malware might adjust priorities for persistence or performance.
- policy
  Scheduling policy (e.g., FIFO, RR, etc.). Malware might favor real-time policies for fast execution.
- vm_pgoff
  Page offset in memory. Low-level memory usage feature, can hint at evasion.
- vm_truncate_count
  Times VM memory was truncated. Rarely used in benign software, may indicate manipulation.
- task_size
  Virtual memory size of the process. Malware may allocate unusually large memory.
- cached_hole_size
  Size of cached memory holes. May correlate with memory manipulation.
- free_area_cache
  Info on free memory areas for allocation. Unusual patterns may indicate heap spraying.
- mm_users
  Number of users of memory mapping. High sharing may hint at code injection or DLL hooking.
- map_count
  Number of memory mappings. Malware may have abnormal memory mapping behavior.
- hiwater_rss
  Peak resident set size. It could reveal memory peaks during execution.
- total_vm, shared_vm, exce_vm, reserved_vm
  Various views of memory usage. Disproportionate values may point to malicious intent.
- nr_ptes
  Number of Page Table Entries. It may reflect memory complexity of the process.
- end_data

End address of initialized data section. Low-level executable format analysis.

- last_interval
  Possibly time since last recorded behavior. This column can help identify timing patterns in behavior.
- nvcsw, nivcsw
  Voluntary / involuntary context switches. Malware may behave differently under CPU pressure.
- min_flt, maj_flt
  Minor/Major page faults. Excessive page faults could be a red flag and be potential malware possibility.
- fs_excl_counter
  Exclusive file system access count. Malware may frequently access or lock files to waste the resources and therefore harm the system and block the access.
- lock
  Use of locks or mutexes. It could indicate inter-process manipulation or stealth.
- utime, stime, gtime, cgtime
  User/system/group/child time of CPU usage. Resource consumption patterns often differ between benign and malicious code.
- signal_nvcsw
  Signals associated with context switches. This could reveal abnormal process control behavior.

Building the Models

Now we are ready to build and evaluate our machine learning models using the labeled dataset. The complete implementation and the dataset in CSV format and the Jupyter Notebook used for training the models is available in the GitHub repository linked here.

Since the dataset we are using includes labeled entries (classified as either "malware" or "benign"), the models would be supervised classification models. The goal is to learn patterns from labeled data that can help predict the class of unseen samples. The models chosen in the s project are K-Nearest Neighbors (kNN), Decision Trees, Random Forest, and Linear Support Vector Machine (SVM). They are all widely-used algorithms in machine learning, particularly for binary classification tasks like malware detection.

The dataset we are using has been pretty much well-preprocessed and vectorized. However, we still need to take care of the non-numeric columns and impute the missing cells of the data so that the models can be trained properly.

The Python library *Scikit-learn* provides with many useful data preprocessing tools and classifiers to help train the models like *ColumnTransformer, SimpleImputer, KNeightborsClassifiers,* etc. The implementation can be found in the GitHub repository. The prediction and the actual labels are printed in the notebook for reference.

```
knn predicted labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
actual labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
------------------------------------------------------------
decision tree predicted labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
actual labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
------------------------------------------------------------
random forest predicted labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
actual labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
------------------------------------------------------------
svm predicted labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
actual labels:
['malware' 'malware' 'malware' ... 'malware' 'malware' 'malware']
```

Performance Evaluation

In this project, we implemented 4 different models: kNN, decision trees, random forest, and linear support vector machine. To evaluate the performances of these 4 models. We will calculate their f1 scores, confusion matrices, and finally plot their ROC curves to decide which of the four works the best as our classification model.

f1 Score [7]

The f1 score is a very objective and an efficient way to evaluate the performances of the models. Generally, the higher the f1 score is, the better the performance of the model is. F1 scores considers both precision and recall:

$$Precision = True\ Positive / (True\ Positive + False\ Positive)$$
$$Recall = True\ Positive / (True\ Positive + False\ Negative)$$
$$F1 = 2 * (Precision * Recall) / (Precision + Recall)$$

The Python library *Scikit-learn* provides with many useful functions to help evaluate the models. Before calculating with any of these evaluation tools, we used *cross_val_predict* (a scikit-learn function) to prevent bias by the models, giving out-of-fold predictions.

The *f1_score* function comes in handy when we want to calculate the f1 score. In this project, we'd like to compare the 4 models, their f1 scores are (round to the fourth decimal place):
kNN: 0.7934
Decision tree: 0.8818
Random Forest: 0.8956
Linear SVM: 0.8431

Confusion Matrix [7]

Confusion Matrix is a table representation to define the performance of models. In a binary classification models (like the ones we have in the project, classifying the entries as either "malware" or "benign"), the confusion matrix would look like this:

True Positive	False Positive
False Negative	True Negative

The *confusion_matrix* function shows the confusion matrices of models in 2D arrays. We want the True Positive and True Negative as high as possible, while the False Positive and the False Negative as low as possible. In this project, the confusion matrices of the 4 models are:

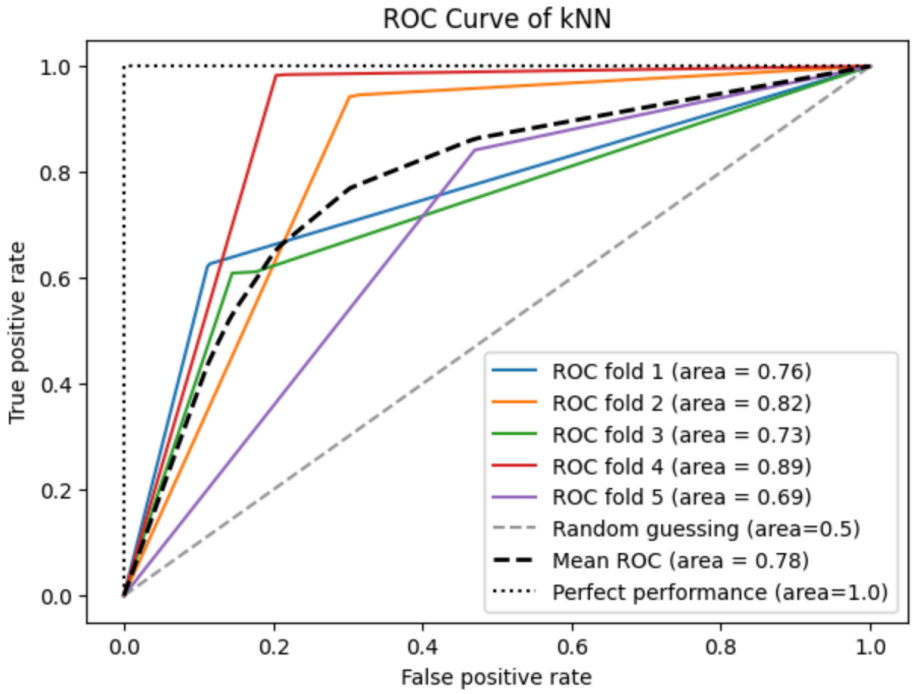
```
[[37819 12181]  [[41654  8346]  [[46628  3372]  [[40880  9120]
 [ 9113 40887]]  [ 3989 46011]]  [ 6719 43281]]  [ 6920 43080]]
```

kNN | Decision Trees | Random Forest | Linear SVM

ROC Curve [7]

The Receiver Operating Characteristic curves can be plotted to visualize the trade-off between true positive rate and false positive rate across different thresholds.

We split the dats into 5 thresholds and use the python library *matplotlib* to visualize the ROC curve. The plot of each model can be found under each cell of the Jupyter Notebook. Here is the ROC curve if the kNN model:



Generally, we want the ROC curves of each threshold to be as close as possible to the upper-left corner, which represents perfect classification performance. In the chart, it is the dark grey dotted line. The closer a threshold's curve is to this line, the better its performance.

The light grey diagonal line represents the baseline of random guessing. If a model's ROC curve falls below this line, it means that the model performs worse than randomly guessing, which means it is making poor or misleading predictions.

The steeper the curve, especially near the beginning (top left), the better the model is at distinguishing between classes. In this project, malware and benign entries.

Comparison

Given the f1 scores, confusion matrices, and the ROC curves, random forest seems to have the best performance (high accuracy and high f1 score, really steep ROC curves), while the kNN has the worst performance(low accuracy and low f1 score, not as steep ROC curves).

Presentation

Documenting the full implementation of this project from initial idea development to evaluation has been a rewarding process. In addition to helping track decisions and methodology, by writing a report documentation, it serves as a valuable reference for future improvements and discussions.

The code for this term project, including data preprocessing scripts, model training, evaluation routines, and visualizations, has been made available through a GitHub repository. Sharing the project on GitHub could ensure easier access and enhance collaboration, and opens up potential contributions from other students, researchers or developers who are interested in malware detection, encourages feedback, reuse, and collaborative communication.

Conclusion & Reflection

- What is achieved

This project helped me gain a broad understanding of different types of malware and their attack mechanisms. It also introduced me to various open-source datasets available online that can be used to train machine learning models for malware detection. Through hands-on

implementation, I developed a better understanding of system-level features and how they contribute to identifying malicious behavior. Additionally, documenting the project process and using GitHub to share progress improved my workflow and organization as well as prepared me for collaborative and open-source development practices.

- What's more to be done

Due to time constraints, some goals of this project could not be fully achieved. However, there are several directions for future work:

Hands-On Exploration on EMBER Dataset

I plan to explore and utilize the EMBER dataset. The environment setup for EMBER was not quite successful given the limited time and I had to turn to another alternative. By utilizing EMBER, it could potentially improve the generalization and robustness of the detection system. And prevent overfitting since now the dataset might be too homogenous and produces bias.

Sophisticated Analysis

A deeper and more sophisticated analysis of malware behavior is essential for understanding evolving attack patterns. Future work could involve a detailed study of malware signatures, behavioral traces, and execution flows. This would allow the detection models to capture more complex patterns and adapt to new or obfuscated threats.

In addition, the outcome of the random forest seems to perform the best of the 4 models, we can try to improve and develop even more complex models and features based on this random forest model and fine-tune it to get a even better outcome.

Optimizing the Interface

While the main focus of this project was on building accurate and efficient detection models, an intuitive and user-friendly interface is equally important. A well-designed interface would allow cybersecurity professionals and general users to easily interpret model outputs, visualize threat levels, and take timely actions. Future iterations of this project could involve integrating visualization tools and user dashboards to enhance usability.

Reference

1. Ember Dataset GitHub: https://github.com/elastic/ember
2. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models, Hyrum S. Anderson, Phil Roth, https://arxiv.org/abs/1804.04637
3. VirusShare website: https://virusshare.com/
4. Kaggle website: https://www.kaggle.com/
5. Burak Ergenc's GitHub Repository: https://github.com/mburakergenc/Malware-Detection-using-Machine-Learning
6. Ember Dataset Tutorial: https://medium.com/@clintonvisaya/getting-started-with-ember-a-step-by-step-process-on-how-to-install-and-run-ember-on-a-windows-10-2f42b350917e

7. Machine Learning with Pytorch and Scikit-Learn, Vahid Mirjalili, Sebastian Raschka, Liu Yuxi, https://www.amazon.com/Machine-Learning-PyTorch-Scikit-Learn-learning-ebook/dp/B09NW48MR1 (chapter 6)
8. A Comprehensive Review on Malware Detection Approaches, Ömer Aslan Aslan, Refik Samet, https://ieeexplore.ieee.org/abstract/document/8949524
9. Intro to malware, IBM, https://www.ibm.com/think/topics/malware (for general understanding)