Static Malware Analysis

Research Supervisor

Dr. Rajshekar K

Research and Development Project Report Submitted by

Mohammad Sameer
Roll Number 190010024
and
Pranav Kanire
Roll Number 190010033

॥ सा विद्या या विमुक्तये ॥

भारतीय प्रौद्योगिकी संस्थान धारवाड
Indian Institute of Technology Dharwad

Indian Institute of Technology Dharwad
Computer Science and Engineering
April, 2022

Abstract

With modern evolution of technology, now malwares are also evolving everyday. But with Machine Learning approach we can detect malwares in easy and efficient manner. In this paper we explain how malware can be classified with the help of API calls and Byte n-grams. We have given different details about the methodologies required to detect malwares using these methods.

API calls are used by the majority of Windows-based apps to perform tasks. As a result, malware developers frequently exploit API calls to carry out destructive acts. Thus, we focus on finding all Windows API call features in binary static analysis to better understand malware behaviour.

We have also developed a model for static malware analysis using byte n-grams. From PE executable we generate n-grams. Using mutual information gain feature selection is performed. With 50 best features, a Decision Tree classification model is built.

Contents

List of Figures

List of Tables

Static Malware Analysis

1 Introduction

Malware is code running on a computerized system whose presence or behavior the system administrators are unaware of; were the system administrators aware of the code and its behavior, they would not permit it to run.

Malware compromises the confidentiality, integrity or the availability of the system by exploiting existing vulnerabilities in a system or by creating new ones.

The above definition of malware is cited from [1]. It is one of the best definitions of malware from a system administrators' point of view. But for technical heads like ourselves, it gives very little information to work with. So, in a more technical way malware stands for malicious software, which refers to any script or binary code that performs some malicious activity.

Modern techniques and software enable faster detection of previously known malware than ever before. However, this is insufficient. Every day, new previously unknown malware with malicious intentions enter the internet. As a result, being able to detect new malware is an important aspect of online security. While dynamic malware analysis focuses on detecting malware during the program's runtime, static malware analysis focuses on detecting malware using the program's static binary file. We have mainly focused on Portable Executable (PE) windows executable files. The following are some of the most important static malware methodologies (cited from [2]):

- Byte-n-gram analysis
- Opcode n-gram analysis
- API Calls analysis
- Header part analysis

All of the above-mentioned analysis methods are primarily concerned with obtaining static characteristics from PE files in order to use as features in various machine learning techniques to classify malware files. We concentrated on Byte-n-gram and API Calls analysis methods for this project. Section 2 contains an extension explanation of these methods. Then, in section 3, we describe our goal/objective that we intend to achieve after reading various research papers. Following that, in section 4, we go over the methodology we used to build a machine learning model to classify malware files. Section 5 summarises a few of our theoretical/experimental findings and future plan of work. Finally, in section 6, we conclude the report.

2 Background

In this section we present an extension explanation of API Calls analysis and byte-n-gram analysis methods.

2.1 API Calls

The core of the Windows Operating System consists of the application programming interface (API). Windows API enables the programs to exploit the power of Windows and hence malware authors make use of the API calls as a vehicle to perform malicious actions [3]. The Windows API function calls comprises of functional levels such as system services, user interfaces, network resources, windows shell and libraries, etc.

Majority of the Windows-based applications make use of API calls to execute tasks [4]. Hence, malware authors readily use API calls as a way to perform malicious actions. Therefore, in binary static analysis we focus on identifying all Windows API call features to understand the malware behaviour. This approach extracts and analyses these API call features including hooking of the system services that are responsible to manage files.

The API call sequences provided by static analysis need to be discovered by following multiple paths of the malware code, and a meaningful representation of the extracted information is rather difficult and time-consuming [5]. So the methodology should be very specific and concise. Droid API Miner ([6]) analysed that, most of Android apps contain one or more third-party packages. These packages often exhibit some suspicious behavior. For instance, many ads use encryption to hinder their removal. So we also need to ensure that there isn't any third party packages included. Some steps involved are as follows:

- Unpack the malware
- Disassemble the binary executable to retrieve the assembly program.
- Extract API calls and important machine-code features from the disassembly program.
- Train ML model over API calls as features.

A statistical analysis of the Windows API calling sequence reflects the behavior of a particular piece of code. Many articles mentioned some repetitive main categories of suspicious behavior of API call features. These being i) Search files to infect, ii) Copy/Delete files, iii) Get file information, iv) Move Files, v) Read /Write files and vi) Change file attributes. Among these, API calls for Read/Write files were predominantly used by malware as a vehicle to infect the program [3].
Few API calls along with their suspicious behaviour as a malware is shown below:

API function calls	**Suspicious behavior description**
GetWindowsDirectory, GetSystemDirectory	Obtain the system directory
FindFirstFile, FindNextFile	Search files to infect
CreateFile, OpenFile, WriteFile, CloseHandle	File write
GetFileAttributes, etFileAttributes	Modify file attributes
GetFileTime, SetFileTime	Modify time of file
GlobalAlloc, GlobalFree	Distribute global memory
VirtualAlloc, VirtualFree	Distribute virtual memory
RegCreateKey, RegSetValue, RegCloseKey	Load register

Table 2.1: Some API calls and respective behaviour

2.2 Byte-n-gram

There have been numerous research papers published on using Byte-n-gram technique as feature extraction for malware classification. One of such well cited papers is the paper titled "Learning to detect and classify malicious executable in the wild" [7]. This paper in many ways forms the basis of numerous other paper on this topic, it summarizes the overall methodology of building an ML model using byte-n-grams as features.

After going through the paper, it can be understood that building a malware classifier using byte-n-gram as features involves 3 important steps. Those are feature extraction, feature selection, building and testing the Machine Learning (ML) model. Each of these steps are explained below.

1. **Feature Extraction:** N-grams is technically defined as a contiguous sequence of n items from a given sample of items. Here items can be set of words or bytes of data etc. In their analysis they use digits of the hexadecimal dump of the PE file, this item is also referred as a word. In such a case, the word can be any integer number of digits that form an integral multiple of a byte. For example, if were to take the word as one byte then it would have 2 hex digits, since each hex digits is of 4 bits. This is exactly what the authors of the said paper ([7]) have taken as a word.

   The 'n' in n-gram refers to how many of such words we consider at once. In their analysis they found taking $n = 4$ gave the best results. So, to extract these byte-n-grams, they use `hexdump` Linux command to get the hexdump of the PE files and some high level programming language to get n-grams.

2. **Feature Selection:** With these features in hand, we cannot directly proceed with building ML model, as we can get billions of unique n-grams. For example, with $n = 4$, we could get $16^8 = 4,294,967,296$ unique n-grams at most. So, the next step is to select only best few of these n-grams. The authors of the paper choose to use Entropy and Information Gain to select top 500 n-grams.

3. **Building and testing the ML Model:** The next natural step is to build the ML model. Although, this step can be subdivided into number of steps and elaborated in greater detail, the primary focus of our analysis does not lie in this step. The primary focus is to test our method of analysis. The authors of the above paper, discuss briefly about various ML models that can be used to build a malware classifier. In their paper, they have used WEKA library to build ML models like Instance-based Learner, Naive Bayes, Support Vector Machines, Decision Trees and Boosted Classifiers.

The results of such classifiers were impressive with accuracy as high as 99% in the case of boosted decision trees. The authors later discuss why a few ML models did not preform as good as others. The model which we felt was interesting was the decision tree. In this model, we could graphically look at the tree and analyze based on presence or absence of what n-grams the files are classified to one class or the other. This is exactly what we have taken up to do in the implementation part of this project.

After reading this paper, we have glanced over the paper titled " An investigation of byte n-gram features for malware classification" ([8]). This paper talks about many important points which many other papers as ([7]) fail to mention. They primarily focus on

- Finding flaws in how previous corpora were created that leads to an overestimation of classification accuracy.
- Discovering that most of the information contained in n-grams stem from string features that could be obtained in simpler ways.
- Demonstrating that n-gram features promote overfitting, even with linear models and extreme regularization.

Most of their focus was on how to maintain the datasets, use ML models with care to avoid overfitting. They also mention that byte-n-gram models learn information from the text regions of the PE files, which is a plus for analyzing decision tree models, since n-grams can be seen in this model. Keeping these important points in mind, we set a few objectives before starting the implementation part of our project. In the later sections this is discussed in detail.

3 Objective

After going through the API call and Byte-n-gram analysis, firstly we set to find the source code of the papers. Unfortunately, the code was not available. And due to time constraints, we have decided to go ahead and code any one of these analysis methods completely from our understanding. We have chosen to code the byte-n-gram analysis.

Having chosen the byte-n-gram analysis, we have set ourselves a goal to build a decision tree ML model based on byte-n-grams as features. After which, we would analyze the tree to get insightful results about the model. In the following section we have elaborated the methodology followed to built this model.

4 Methodology

For the purpose of the analysis we have used a data set available online, which is a combination of files from various sources. The link to which is `https://github.com/iosifache/DikeDataset`. Out of these files we have taken 100 benign and 100 malware files. Our project source code files can be found at `https://github.com/sameermuhd/Malware-Analysis`.

4.1 Feature Extraction

In this step we have iterated over all the benign and malware files and found all unique byte-n-grams. Here, we have taken size of one word as 1 byte and $n = 4$. For this purpose we have written a bash script file named `hexdump.sh` which can be found in the github repository.

Having collected all of the unique n-grams, now our job is to find n-grams present in a particular file. To do this we have used `python` code and written the output to a `csv` file. This `csv` file contains file names as row index, label as one of the column, where 1 indicates a malware and 0 indicated a benign file. Every column part from the name and label, indicates an n-gram. Now, if an ith row and jth column is 1, it indicated that the file with name i, has an n-gram j, present in it, if the same is 0, it indicate absence of that n-gram.

In actual implementation due to limitations of our machine, we did not consider all the unique n-grams, which were approximately 23 million in number. Instead we have chosen $100,000$ n-grams randomly out of them, the reason for doing this is mentioned in section 5. So, at this point, we have a `csv` file with features extracted, on which we can perform feature selection.

4.2 Feature Selection

As the number of resulting features from feature extraction is very large, in this ([8]) paper, they used various methods of feature selection and reported the use of information gain based selection methods as the best in the malware classification. We decided to follow the same.

The mutual information gain is defined as

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C_i} P\left(v_j, C_i\right) \log \frac{P\left(v_j, C_i\right)}{P\left(v_j\right) P\left(C_i\right)}$$

where C_i is the i th class, v_j is the value of the j th attribute, $P\left(v_j, C_i\right)$ is the probability that the j th attribute in the class C_i has the value v_j , $P\left(v_j\right)$ is the probability that the j th n-gram takes the value v_j in the training data, and $P\left(C_i\right)$ is the probability of the training data belonging to the class C_i. This measure is also known as average mutual information.

Thus, Information gain is used for feature selection, by evaluating the gain of each variable in the context of the target variable. Information gain calculates the reduction in entropy or surprise from transforming a dataset in some way. We used *mutual_info_classif()* method from *sklearn.feature_selection* module to implement this.

Also it is commonly used in the construction of decision trees from a training dataset, by evaluating the information gain for each variable, and selecting the variable that maximizes the information gain. From this step we were able to get the top 50 byte-n-grams with high information gain, which were later used for building the decision tree model. Below graph shows the best features after performing feature selection based on mutual information gain, where the X-axis indicates various n-grams and Y-axis indicates the mutual information gain.

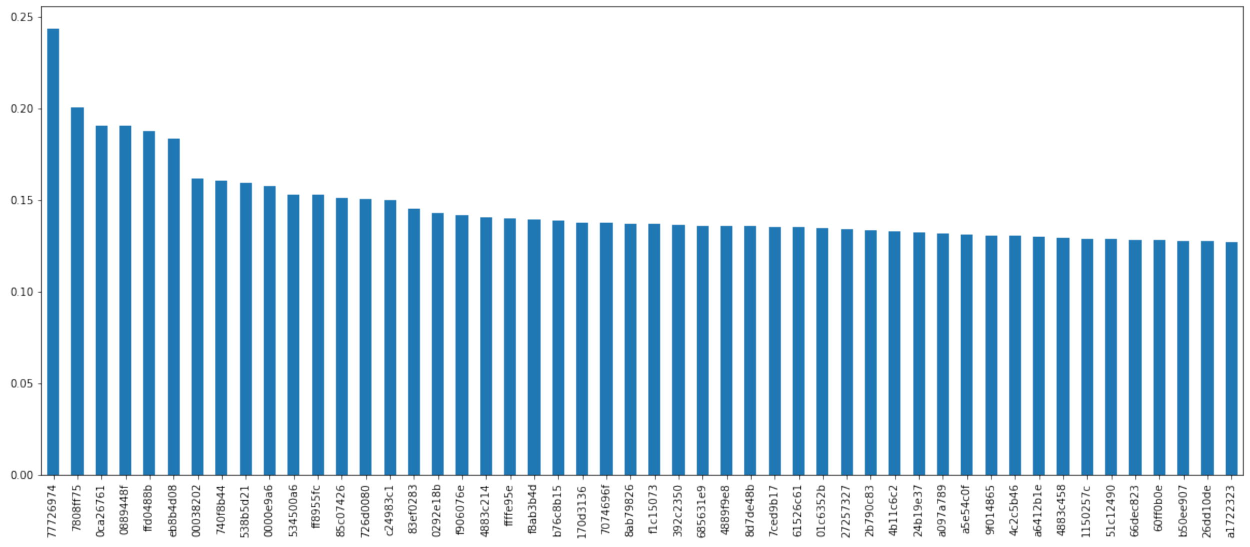

Figure 4.1: Mutual Information Gain

4.3 Classification Model

As discussed above we have Decision Tree as a classification model. Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. We specifically chose this method because it is simple to understand and to interpret. We can easily visualize trees and understand the role of different features.

Decision Trees are a type of algorithm that uses “if” and “else” conditions to make choices. Decisions are made on the task at hand based on these conditions. An algorithm based on the facts (data) at hand determines these conditions. The number of conditions, types of circumstances, and responses to those conditions are all data-driven and will vary each dataset.

Usually DTs performs well even if its assumptions are somewhat violated by the true model from which the data were generated. Since we can’t have all byte n-grams available to train model we decided to implement DTs here. We implemented this using *DecisionTreeClassifier()* method from *sklearn.tree* module.

After building the decision tree and testing it on test set with 50-best features, we got around 92% accuracy. Below we have attached the decision tree generated with 20 best features. With this tree we can easily understand which of the generated features has more impact on classifying a file as a malware. We can analyse these features and improve accuracy after understanding the reasoning for misclassification.

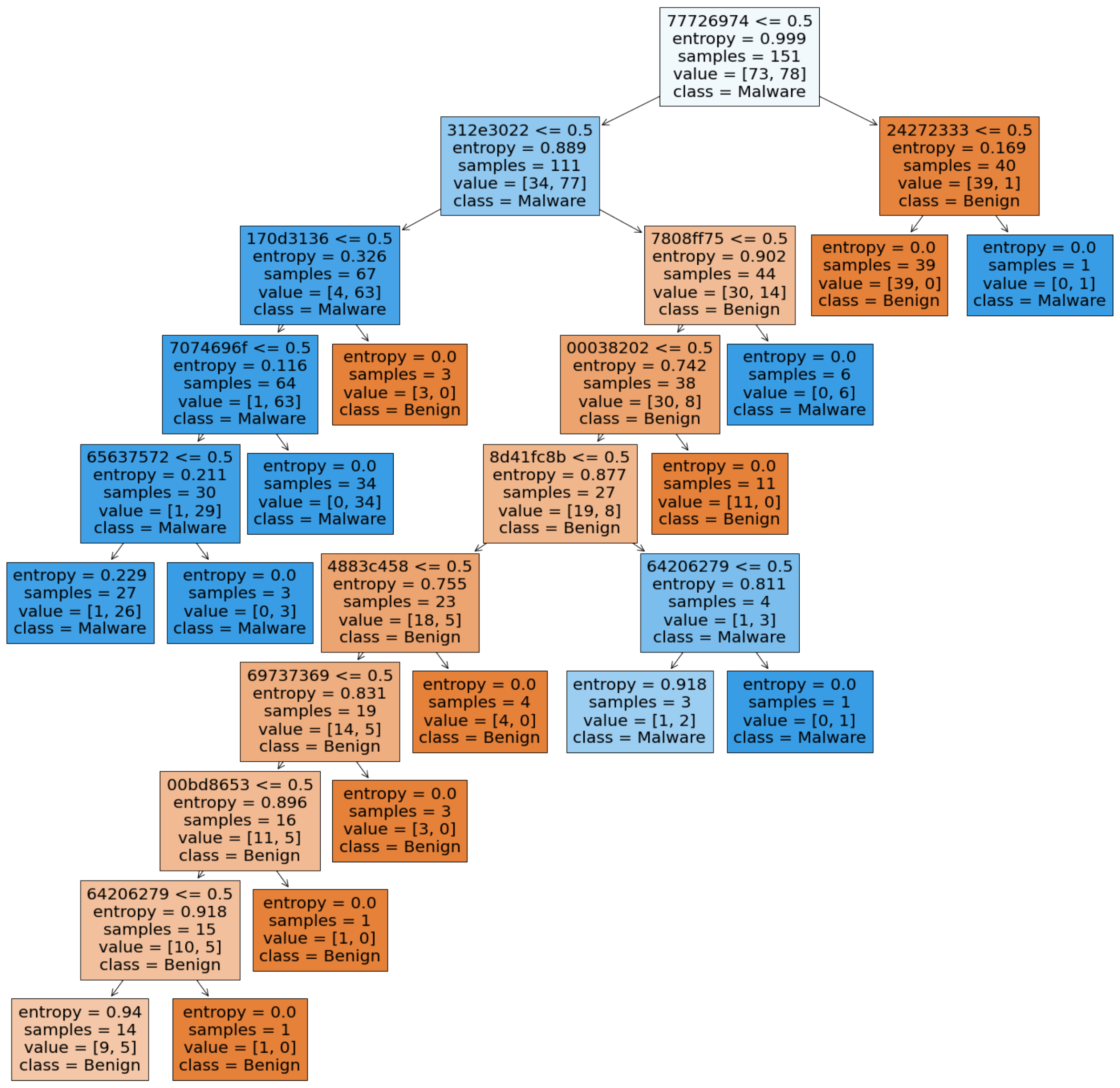

Figure 4.2: Decision Tree using just 20 files

5 Discussion and Future plan of work

The implementation of code completely by ourselves gave us a lot of freedom to work with tools and libraries of our choice. Some of our choices made our process smother but still challenges exists. The following are some of the most crucial theoretical/experimental findings:

- A major issue with malware classification based on byte-n-gram analysis is that it creates an extremely large explosion of features when n is increased. The number of unique n-gram we get is astronomical, we could get at most $16^8 = 4,294,967,296$ unique n-grams with just $n = 4$. So feature extraction and selection becomes very critical here. So we would like have a deeper look into it in future, so we don't miss any key features here.

- Being able to fit all of this data in a `csv` file and later load it into memory for feature selection is not easy on RAM. On our limited data set, using $n = 4$, we have completely utilized the complete 16 GB RAM of our PC but were still not able to load the complete `csv` file in python notebook. Hence we decided to choose random $100,000$ n-grams.

- The number $100,000$ is not any means calculated or scientific. It is just a number from the top of our head that we felt was reasonable. It remains to be seen on how to optimize the way we extract and store the data.
- We want our features to be large enough that they are not likely to occur by chance in a new file, but small enough that they are not specific to the training data as this would degenerate into a signature-based approach, which would have maximum specificity but no generalization. So we need to try our model on a bigger data set.
- At few nodes of our decision tree, we could see that the model is not able to classify the files into further classes, it could be due to lack of unique differentiating n-grams in those files. This could be due to the random selection we have done in feature extraction. We hope to fix this in our future work.
- Another point to note is the skew nature of the tree, which may or may not be too significant. But this too has to worked on and seen if it matters.
- An important point we would like to work on is the dimensionality. To be precise, checking how many top n-grams to choose to not cause over or under fitting.
- We intend to calculate false positives and negatives. As future work, we plan to reduce the false positives and negatives through analyzing the samples that were not correctly classified and finding out the reasons behind the misclassification.
- Looking at a few n-grams in the tree, we were able to see that a few ASCII strings seem to appear. This is in fact what authors of [8] have mentioned, that n-gram model seem to learn a lot of data from the text part of PE file. So, in the future we could try and run the analysis only on the text part of the PE files.
- This particular model has potential for further improvement. Given enough time, we believe we can improve it even further. Along with this we also could take up API call analysis, explore it and find ways to combine these two analysis if possible.
- Combining several analysis methods, such as byte-n-gram and API call analysis, or byte-n-gram and opcode-n-gram, is a major prospective analysis strategy that we intend to focus on in the future.

6 Conclusion

We focused on the byte-n-gram and API call static malware analysis method in our project. Later working on byte-n-gram analysis. While it remains an open issue whether an executable's static characteristics are predictive of its function, we have presented evidence in the form of ML model with 92% accuracy, suggesting it may be possible to achieve useful detection rates when predicting function. We believe this area of research has a lot of potential because it has a promising model and many interesting concepts to work on. All that is left now is to put in the effort.

References

[1] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, “Dynamic malware analysis in the modern era—a state of the art survey,” vol. 52, no. 5, sep 2019. [Online]. Available: https://doi.org/10.1145/3329786

[2] A. Shalaginov, S. Banin, A. Dehghantanha, and K. Franke, “Machine learning aided static malware analysis: A survey and tutorial,” in *Advances in Information Security*. Springer International Publishing, 2018, pp. 7–45. [Online]. Available: https://doi.org/10.1007%2F978-3-319-73951-9_2

[3] M. Alazab, S. Venkatraman, and P. Watters, “Towards understanding malware behaviour by the extraction of api calls,” 07 2010.

[4] G. G. Sundarkumar, V. Ravi, I. Nwogu, and V. Govindaraju, “Malware detection via api calls, topic models and machine learning,” in *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, 2015, pp. 1212–1217.

[5] A. Pektaş and T. Acarman, “Malware classification based on api calls and behaviour analysis,” *IET Information Security*, vol. 12, no. 2, pp. 107–117. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-ifs.2017.0430

[6] Y. Aafer, W. Du, and H. Yin, “Droidapiminer: Mining api-level features for robust malware detection in android,” in *Security and Privacy in Communication Networks*, T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds. Cham: Springer International Publishing, 2013, pp. 86–103.

[7] J. Z. Kolter and M. A. Maloof, “Learning to detect and classify malicious executables in the wild,” *J. Mach. Learn. Res.*, vol. 7, p. 2721–2744, dec 2006.

[8] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, “An investigation of byte n-gram features for malware classification,” *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 1–20, Feb 2018. [Online]. Available: https://doi.org/10.1007/s11416-016-0283-1