Malware Detection using API Call Graphs

Sneha Mandan (ID:201401422)* and Aalisha Dalal (ID:2201401433)†
BTech Mini Project (PC403)
Project Guide: Prof. Anish Mathuria

The project involves implementing a tool which will efficiently detect malware programs. It has been observed that the graph theoretic properties of API call graphs for benign softwares differ from malware [1]. We will be using this property and machine learning to classify the programs as either benign or malware. This project broadly consists of three phases. In the first phase, we generate API call sequence and construct an API call graph for each of the malware and benign programs. In the second phase, feature extraction from the graphs is done. These features are used as input for the classification model. In the third phase, we choose an optimal classification model. The performance of tool is measured by testing it for unknown programs. This approach will be efficient with respect to both space and time as compared to the signature-based approach where programs were examined against malware signatures stored in database.

INTRODUCTION

Malware

Malware, a malicious software, is a software designed to infiltrate a computer system without the owner's knowledge or consent. There are various categories of malware such as Virus- a self replicating program that hooks to a Windows process and spreads to other executables, Worm -a self replicating program that infects systems on the network, Trojan - a dangerous program in disguised form, Drive-by-download - Infects the system on visiting a bad website by exploiting the weakness of browser, etc. A program which is not a malware is a benign software.

In the Windows O/S, user applications rely on the interface provided within a set of libraries, such as *kernel32.dll*, *ntdll.dll* and *user32.dll* in order to access system resources including files, processes, network information and the registry. This interface is known as the Win32 API[2]. In a malware program, API calls are related to functionalities such as Search files, Copy/Delete files, Get file information, Move Files, Read /Write files and Change file attributes are exploited. The API calls that create or modify files or even get information from the files to change some value and information about the DLLs are more likely to be present in malware programs.

Past work

.

- Static Analysis Approach The static analysis approach can be either signature-based or behavioural approach[3].

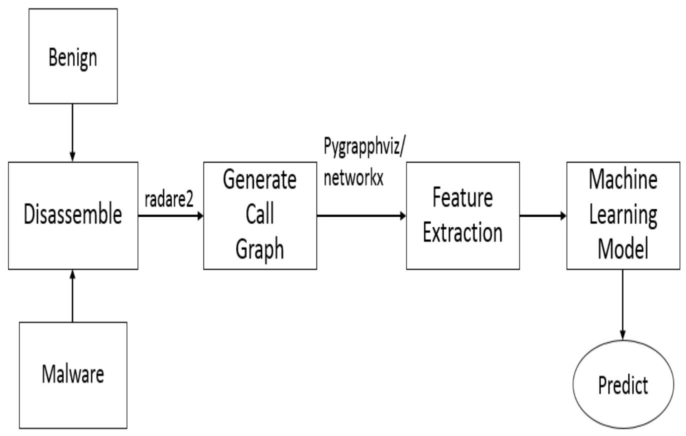
  1. Signature-Based Approach
     In the static signature-based approach, the byte code in the malicious binaries is hashed and stored in the database. The number of signatures in the database grows at a much higher rate in this approach. The signature of the program would be compared with a large database of signatures and due to which it is computationally very expensive. Along with that, it does not help detect *zero-day malware*. Zero day refers to the day the vulnerability is discovered. If the bug is not fixed, the attackers can exploit the security vulnerability and execute zero day malware attack.
  2. Static Behavioural Approach
     The behaviour of the code can be analyzed using control flow graphs.In our project, we have followed a static behavioural approach for malware analysis. This approach is not only efficient both in terms of storage as well as time, but it is immune to malware obfuscations. This was the motivation to follow this approach.

- Dynamic Based Approach

  The dynamic analysis of a malware is done by running it for few times in a virtual environment. Based on the change in behaviour on the computer system at run-time, malicious behaviour and its effects can be determined.

  But this approach has certain limitations.

  1. The dynamic analysis of malware has to be carried out in a virtual environment to protect the computer system from its malicious behaviour. In current times, the malware programs have become more neat. Having realized a virtual environment, the malware program will refrain from carrying out malicious activity. Due to this, detection of malware using dynamic-based approach becomes cumbersome.



2. Secondly, the dynamic-based approach for malware analysis requires execution of the malware program and hence, detection takes a much longer duration.

Motivation

Function call graph is chosen as signature of malware as it reprsents the functionality and objective of the program semantically. Even after obfuscation of the code, the functionality of the program would remain the same. Hence, function call graph would be more resilient to code obfuscation than string signatures[4].

Dataset

In this work, we are primarily focusing on the windows executable files. The malware samples have been collected from the git-hub repository *zoo malware dataset* [5]. The data-set consisted of about 32 malware files and 90 benign files.

MODEL

Generating Call Graphs

API calls represent the abstract functionality of the programs in Windows executable. Windows API calls are also used to obtain system-level information. Malicious code make use of these calls to execute malicious behaviour. The sequence of the *jump* and *call* instructions in malicious programs can be useful to differentiate benign and malware programs. Anti-Virus software developed using a signature-based approach are vulnerable to malware obfuscation. Along with that, the signature-based approach has a much higher time and space complexity. In our approach, we are developing a classification model which captures the behaviour of the benign and malware programs through the graph theoretic features obtained from the API call graphs.

The function call graphs can be generated from the program executables using softwares that allow reverse engineering. The tool used in this project is an open-source tool, *radare2*. *radare2* is a reverse engineering framework which supports executable files for Windows, Linux as well as Mac OS X operating system. It also supports malformed binaries which made it an appropriate tool for our research. The tool was used to disassemble the executable and produce an analysis through function call graphs. The same can be achieved via proprietary software IDA Pro as well. But IDA Pro limits its functionalities to Windows program executables.

In the first step, we generate the API call graph of benign and malware programs from their executables using radare2. After disassembling the entire binary file and running a statistical analysis in radare2, the generated call graphs were stored in .dot file format. The dis-assembly view represent the call graph in radare2 as basic blocks of assembly code separated by conditional jump instruction. The generated graph for all functions is stored in graph's .dot file format.

For the graph G(V,E), the nodes V represent the functions and the system calls in the program. And, the edges represents caller-callee relationship. It can be visualized using any graph visualization tool. Here, we have used python packages ”pygraphviz” and ”networkx”.

Feature Selection

From the .dot files generated in the previous step, graph theoretic properties are determined which will be used as features to train the learning algorithm. *pygraphviz* and *networkx* packages of Python are used to obtain the features. Let G denote the graph, V the set of vertices and E denote the set of edges in graph G. The features that we focus on in this study are as follows

Indegree: For a vertex, indegree is the number of incident edges on that vertex. Here, it is the number of function calls made to that subroutine. *average indegree* for the graph is the mean value of indegree of all the vertices. *max indegree* is the maximum of the indegree of all nodes. *var indegree* is the variance of the indegree values of all nodes.

Outdegree: For a vertex, outdegree is the number of incident edges from that vertex on to other vertices. Here, it is the number of function calls made by that subroutine. *max outdegree* is the maximum of the outdegree of all nodes. *var outdegree* is the variance of the outdegree values of all nodes.

Average shortest path: The average shortest path length is

$$a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)} \tag{1}$$

where $d(s,t)$ the shortest path from node s to node t, and n is the number of nodes in G.

Diameter: The diameter is the maximum eccentricity. The eccentricity of a node v is the maximum distance from v to all other nodes in G.

Average eigenvalue: It is the average of eigenvalues of the adjacency matrix of G.

Density: It describes the fraction of the *potential connections* in a network that are actual connections. A *potential connection* is a connection that could potentially exist between two nodes regardless of whether or not it actually does. The density of graph is

$$d = \frac{m}{n(n-1)},$$

where n is the number of nodes and m is the number of edges in G.

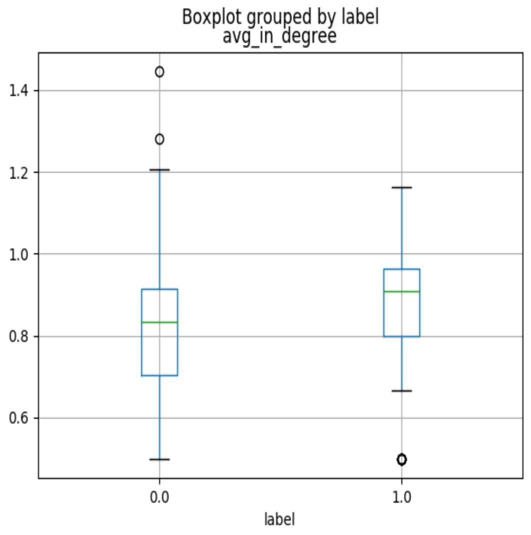
Graph Entropy: The graph entropy represent the uncertainty of a node in the macroscopic description given the uncertainty of the node in microscopic one. The graph with same microscopic and macroscopic uncertainty is considered to have a zero-entropy. For example, a complete graph will have very high entropy since all nodes in the graph have a similar degree structure.
Graph entropy is computed from the normalized degree centrality of each node as

$$ent = \sum_{i}^{N} \frac{c_i}{log_2 c_i} \tag{2}$$

where c_i is the normalized degree centrality of *ith* node.

Feature Selection

Correlation of above features with respect to the output label was observed and only the relevant and independent features were considered for training the model. In addition, the difference in the feature values for benign and malware programs was statistically significant or not was determined from the box-plot.



Logistic Regression Model

Logistic regression is the simplest model which is used for classification. We used this model as a benchmark for malware classification.
The model maps the input to the output by computing the optimal values of parameter θ in the below equation. Here, θ vector represent the weights assigned to each input feature.

$$Y_{pred} = g(\theta' * X + b) = h_\theta(X) \tag{3}$$

Here *g(x)* is the sigmoidal function. $g(x) = \frac{1}{1+exp(-x)}$

The cost of the model J(θ) for all samples can be given by,

$$J(\theta) = \frac{-1}{m} \sum_{j=1}^{N} (y_i * log(h_\theta(x_i)) + (1-y_i) * log(1-h_\theta(x_i))). \tag{4}$$

The optimal value of θ is obtained by iteratively applying gradient descent algorithm given as below.

$$\theta_{i+1} = \theta_i - \alpha * \nabla(J(\theta)) \tag{5}$$

MLP Neural Network Model

Inspired from the functioning of human brain, this deep learning model has proved to perform well even when the input and output has a non-linear relationship.
In this project, we have implemented a three-layer Multi-layer Perceptron neural network model. The three layers are the input, hidden and output layer. The number of neurons in the input layer represent

the number of features in the input while the number of neurons in the output layer represent the number of classes. The weighted matrix W_i maps the input layer to the hidden layer. The dimension of the matrix W_i is #(hidden neurons) x #(input neurons). While, the weighted matrix W_h maps the hidden layer to the output layer. The dimension of the matrix is #(output neurons) x #(hidden neurons).

The neural network model can be formulated as below,

$$Y = N(W_i, W_h, X) = g((W_h * g(W_i * X))) \tag{6}$$

Here, g is a bi-sigmoidal function which can be given as below,

$$g(X) = \frac{-1 + exp(X)}{1 + exp(X)} \tag{7}$$

The error of the model is computed by comparing the predicted output with the actual output.

Error function:
For each input sample, the error is computed using modified least square error (MLSE)

$$\mathbf{err_i} = \begin{cases} 0 & \text{if } Y_{pred} * Y_{actual} > 1 \\ (Y_{pred} - Y_{actual}) & \text{else} \end{cases} \tag{8}$$

The total modified least square error for N samples and n classes is as below,

$$\mathbf{MLSE} = \sum_{j=1}^{N} (\sum_{i=1}^{n} (err_i^2)) \tag{9}$$

The objective is to minimize this error function which is done by updating the weights of the matrix using the backpropagation algorithm similar to logistic regression.

RESULTS AND ANALYSIS

The data-set consisted of about 32 malware files and 90 benign files. Of this, 70% data was used for training the model while the remaining 30% was used for testing purpose. The proportion of benign and malware samples in training/test data was taken similar to that of original dataset using stratified split.

Evaluation Metrics

Confusion Matrix consists of True positives (TP) correctly classified malware instances, True Negative (TN) correctly identified benign instances, False Positive (FP) benign instances misclassified as malware and False Negative (FN) malicious samples misclassified as benign. True Positive rate (TPR) and True Negative Rate (TNR) are known as sensitivity and specificity respectively.[6]. Following metrics were evaluated for each model:

- Recall or TPR = TP/(TP + FN)
- FPR = FP/(FP + TN)
- TNR = TN/(TN + FP)
- FNR = FN/(FN + TP)
- ACC = (TP + TN)/(TP + TN + FP +FN)
- Precision = TP/(TP+FP)

The training data consisted of 63 benign samples and 23 malware samples whereas test dataset consisted of 27 benign and 10 malware samples. To take care of the varying number of samples of each class, weight-age of each sample was taken as inversely proportional to the total number of samples of that class.

Logistic Model

Confusion Matrix for train data:

$$\begin{array}{ccc} & Pred_0 & Pred_1 \\ Actual_0 & 50 & 13 \\ Actual_1 & 8 & 15 \end{array}$$

Confusion Matrix for test data:

$$\begin{array}{ccc} & Pred_0 & Pred_1 \\ Actual_0 & 24 & 3 \\ Actual_1 & 6 & 4 \end{array}$$

The weighted accuracy for training data was 75.58% and for test data was 75.67%. Here, we would like to report that rather than accuracy, focus should be on improving the recall rate. It is acceptable if a benign program, gets misclassified as malware but a malware getting misclassified as benign is matter of concern.

MLP Neural Network Model

Confusion Matrix for train data:

$$\begin{array}{ccc} & Pred_0 & Pred_1 \\ Actual_0 & 59 & 4 \\ Actual_1 & 14 & 9 \end{array}$$

Confusion Matrix for test data:

	$Pred_0$	$Pred_1$
$Actual_0$	27	0
$Actual_1$	4	6

The weighted accuracy for training data was 79.06% and for test data was 81.08%. Thus, MLP model behaves slightly better than Logistic model.

CONCLUSION

- We developed a zero-day malware detection tool using API call graphs of benign and malware program executables.
- Graph theoretic properties of the API call graphs which differentiated benign and malware program were selected as features for the model.
- MLP-based neural network model was chosen as binary classification model.

LIMITATIONS AND FUTURE WORK

- To improve the performance of the model, more features can be explored. A more hybrid data-set constituting equal proportion of malware and benign programs and malwares of different categories can lead to better training of the model. In future, classification of malware programs into its respective family can be done.
- An alternative approach based on graph embedding can be used for classification. In this approach, the vectorized representations of the graphs are generated using a method similar to the Natural language Processing based word embedding models.

* 201401422@daiict.ac.in
† 201401433@daiict.ac.in

[1] Zubair Shafiq and Alex Liu. A graph theoretic approach to fast and accurate malware detection.

[2] Mamoun Alazab, Sitalakshmi Venkataraman, and Paul Watters. Towards understanding malware behaviour by the extraction of api calls. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, pages 52–59. IEEE, 2010.

[3] DEEPTI VIDYARTHI G. HAMSA. Study and analysis of various approaches for malware detection and identification.

[4] Shanhu Shang, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang. Detecting malware variants via function-call graph similarity. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 113–120. IEEE, 2010.

[5] ytisf. the zoo malware database.

[6] Parvez Faruki, Vijay Laxmi, Manoj Singh Gaur, and P Vinod. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks*, pages 130–137. ACM, 2012.