A Graph Theoretic Approach to Malware Detection

By

Syeda Momina Tabish

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science

2012

ABSTRACT

A Graph Theoretic Approach to Malware Detection

By

Syeda Momina Tabish

Current malware detection approaches (i.e. anti-virus software) deployed at end hosts utilize features of a specific malware instance. These approaches suffer from poor accuracy because such features are easily evaded by trivial obfuscation such as garbage insertion and re-ordering of instruction or call sequences. In this paper, we introduce a novel graph theoretic approach to detect malware from instruction or call sequences. In our approach, we map the instruction or call sequence of an executable program to a graph. We then extract features from the constructed graphs at three levels: (1) vertex level, (2) sub-graph level, and (3) graph level. These features act as footprints of the behavior of an executable program and are leveraged to differentiate between benign and malware programs. The results of our experiments show that our graph-theoretic approach differentiates between benign and malware programs with $\approx 100\%$ accuracy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background and Motivation

Malware analysis and detection has recently been a prime research area due to the widespread increase in the number of new and previously unknown malware. The huge monetary gains involved and ease of writing and distributing a malicious piece of code have made it a cat and mouse chase, where the malware writers are ahead and the security experts are closely following behind. Although a number of *behavior* based malware detection techniques have been proposed recently, *signature* based malware detection is still the most widely deployed defence mechanism by commercial anti-virus software. This is mainly because behavior based malware detection techniques suffer from false alarms; whereas, signature based malware detection techniques have zero false alarm rate. Furthermore, behavior based malware detection techniques suffer from large processing overheads; on the other hand, signature based malware detection techniques have acceptable processing overheads.

However, behavior based malware detection techniques are being actively investigated because of their ability to detect zero-day (or previously unknown) malware as they do not rely on any a-priori knowledge about new malware. Some of the behavior based malware detection approaches have been integrated into the commercial anti-virus software with limited success. The state-of-the-art commercial anti-virus software still rely heavily on exact signature matching for malware detection and hence they typically fail whenever there is a new malware outbreak. At the moment the only line of defense for the naive end user does not provide sufficient protection and therefore an efficient and effective non-signature based line of defense is needed that can protect end hosts from previously unseen or zero-day malware. There is still room for non-signature based malware detection techniques that have high detection accuracy, are computationally inexpensive, and are easily scalable with the increasing number of new malware.

1.2 Limitations of Prior Art

Existing commercial anti-virus software generally rely on separate signature or behavioral pattern extraction for every malware sample. This approach requires signature extraction for every malware sample and the size of signature database has a tendency to explode and hence is unscalable due the increasing number of new malware. According to a survey by TrendMicro, almost $5,000$ new malware samples are submitted daily to the anti-virus software vendors for analysis [18].

Some behavior based malware detection techniques have also been proposed in prior literature. A majority of these techniques rely on extracting or mining unique patterns or signatures from the instruction or call graphs, as discussed later in Section 2. The detection is typically done by graph matching algorithms or subgraph based features, which limits the granularity of such approaches to a particular malware instance only.

1.3 Proposed Approach

In this paper, we propose a graph theoretic approach to malware detection at end hosts using instruction sequences or API call sequences. The instruction or API call sequences are mapped to graphs and structural properties of graphs are leveraged to extract relevant features. The distinguishing feature sets, forming the basis of our detection methodology, are extracted at three levels: (1) vertex level, (2) sub-graph level, and (3) graph level. Our proposed vertex level features can be divided into degree, path, and connectivity categories. Examples of these three categories respectively are degree, diameter, and clustering coefficient. The aforementioned and other features also give us interesting insights about general graph properties. We extract a similar set of graph level features. At sub-graph level, we identify and extract features (program instructions or call sequences) based on Markov chains. We also use the concept of *typicality* of Markov chain states to identify program instruction or call sequences of varying lengths. The concept of typicality is important in the context of efficiently identifying a small subset of program instruction or call sequences from a very large sample space. All of the aforementioned features, extracted at various graph levels, are then used along with machine learning algorithms for automatically learning classification models.

1.4 Key Contributions

The key contributions of our proposed two-step approach are described below.

1. The first step of our approach is feature extraction. To our best knowledge, this work is the first attempt to extract and capture graph properties at various levels starting from vertex level, to sub-graph level, and to graph level. The canvas of feature extraction in our approach is wide enough to portray sufficient information for accurate classification. In the second step, this comprehensive feature set is further used in conjunction with machine learning algorithms, so as to make full use of their sophistication and classification capabilities.

2. We evaluate the effectiveness and accuracy of our proposed approach using a comprehensive data set of benign and malware files, which is collected keeping in mind that this data set should be a good representation of a real-world end host. We made sure to include majority of the normal daily use benign application with wide range of functionality some of which also match closely with the malware behavior and vice versa.

The rest of this paper is organized as follows. We provide an overview of the related work in Section 2. The details of our proposed approach are explained in Section 3. In Section 4, we evaluate the accuracy of our proposed approach. We finally conclude this paper in Section 5.

Chapter 2

Related Work

Malware detection has been one of the most pressing security problem in recent years. A lot of work has been done in this field, which can be divided into two broad categories; static analysis and dynamic analysis techniques. Static analysis is based on the concept of detecting or classifying an executable before it is run. On the other hand, dynamic analysis schemes base their detection process while the executable is running. Both analysis schemes have their pros and cons, for instance, in static analysis it is not guaranteed that the code analyzed is the same as the code executed, many polymorphic worms change the execution pattern on the fly. Likewise, in most of the dynamic analysis techniques, the malicious program is first run in a virtual environment or sand box to determine detection check points. We separately discuss both categories of work in the following text.

2.1 Static Detection Techniques

We now discuss and summarize the most relevant static analysis techniques.

In [9], Christodorescu *et al.* presented an approach to detect malicious patterns in executables using static analysis. They also claim that the proposed approach is resilient to common obfuscation transformations. They first generalize the malicious code into an automaton and then an internal representation of malicious code is generated using abstraction patterns. The third process in the proposed framework is that using IDA pro and Code surfer, control flow graphs (CFGs) are created which are further passed as an input to the annotator. The annotator creates an annotated CFG which contains the information such as where an abstraction pattern is found in the code. The last step is the detector which uses the annotated CFG's and the malicious code automaton and decides using code unification if the malware pattern matches with a pattern in the annotated CFG. The evaluation is done on 4 viruses and their obfuscated versions and has good accuracy in terms of false positive and false negative rates. The limitation of the proposed approach is that it is computationally expensive and can handle only very simple obfuscation techniques.

In [10], the authors presented a malware detection scheme which uses semantics of the executables instead of exact string or signature matching. The executable is parsed through IDA pro and CFG's are constructed and an intermediate representation, which is a library of x86 instructions also called a template, is created. The detection algorithm keeps finding template nodes which match the program template and returns with the instructions if a match is found. The limitations of their approach are in the intermediate representation of the instructions, where it is computationally hard to generate a complete set of every possible instruction representation and take significant time for processing as well.

2.2 Dynamic Detection Techniques

Below we describe some dynamic analysis techniques that are most relevant to our approach.

In [4] and [3], Bayer *et al.* proposed an analysis tool for malware security experts so that they do not have to manually analyze the suspicious executable and which can give them enough information about it as to render it malware or benign. The tool is based on a PC emulator Qemu, which can handle self modifying code. The proposed tool works by analyzing Windows API and native API calls made by the malicious process, most importantly it keeps track of file created or opened using function arguments. The analysis is compiled in the form of a comprehensive report which contains, general information (file size, command line arguments, etc), file activity, registry activity, service activity, process activity, and network activity. The evaluation is done on ten malware samples and the analysis report matched closely with the descriptions of majority of them.

In [20] a taint graph based malware detection technique was proposed by Yin and Song. The malicious program is run in an emulator containing a test engine on which the test scripts are run which first introduce important taint information (like password input, TCP/UDP/ICMP traffic etc) and then monitor the activities of the sample program and the overall system under observation. The behavior of the program in a system wide context is further represented in the form of graphs which help to identify if the program tries to read or misuse the taint information. The evaluation is done on 42 real-world malware samples containing root kits, password thieves, key loggers etc and the proposed approach detected all the malware, however, falsely detected 3 benign programs, which closely matched with the malware behavior.

In [6], Bose *et al.* proposed a behavioral detection framework for mobile malware. In this framework, system calls are logged and correlated to create dependency graphs that are further pruned and aggregated for behavioral signature generation. The classification of partial or incomplete malware behavior, when the malware has not yet finished execution is done using machine learning algorithm support vector machine for binary classification. The evaluation of the proposed approach is done on custom made programs which emulate real world Symbian worms and other well known malware. The results show that the proposed framework achieved good detection accuracy with decreasing number of false negatives as the training data size is increased.

In [16], Kolbitsch *et al.* proposed a dynamic malware detection technique which using system calls information automatically creates fine-grained behavior models and match any given unknown program against these models to classify it as benign or malware. The behavior models are represented in the form of graphs in which nodes denote interesting system calls and edges denote the data dependency between two system calls when the input of one is dependent on the output of the other. The first part of the approach is malware analysis that is carried out in a custom made dynamic malware analysis tool Anubis, which records all the disassembled instructions, system calls, data dependencies using taint analysis, and memory logs. All of the above information is used to create behavior graphs. The second part of the approach is detection which comprises of a scanner that monitors system call invocations and checks if the monitored program matches any of the behavior graphs. Upon matching, the malicious program is terminated. The evaluation of the proposed approach is done on 6 malware families and it achieves reasonable detection accuracy.

In [1], Ahmed *et al.* proposed a runtime malware analysis and detection tool that detects malware using spatio-temporal information in API call logs. The spatial information (from arguments and return values of calls), and temporal information (from sequence of calls) is further used to build formal models to distinguish malware from benign programs on the basis of API calls. The most discriminative spatial and temporal features of the API calls are selected on the basis of information gain, and are then passed on to the standard machine learning and data mining classifiers using 10-fold cross validation. The evaluation is done on 416 malware and 100 benign samples. The proposed approach achieves up to approximately 98% detection accuracy.

In [8] a graph based malware detection approach is presented, which focuses on mining patterns in a dependence graph after reducing its data space. The dependence graph is constructed using system calls and is often quite large in practice. The proposed approach first drastically reduces the size of dependence graph by randomly and repeatedly summarizing it, and then it uses subgraph mining algorithm to traverse the pattern space. The evaluation of the proposed approach is done on 6 malware families and it is also compared to gSpan, a state-of-the-art graph miner. The results show that the proposed approach outperforms gSpan by showing stable running time and efficiency and reveals interesting malware fingerprints.

In [15], Islam *et al.* proposed a behavior based malware detection technique, which extracts strings (system call names and function arguments) from malware and benign trace logs, which are then used for classification. Each file of the dataset is run in a virtual environment and using a trace tool HookMe, the system call logs are obtained. The trace files are used in feature extraction, which generates a list of strings present in the binary and stores them in a hash table. The feature vectors are created on the basis of the absence or presence of a string in a particular file. These features are then passed onto data mining and machine learning base classifiers, which are decision tree, rule learner, support vector machine, instance based classifier, and meta classifier to perform k-fold cross validation. The experimental evaluation is done on about one thousand malware and about five hundred benign samples. The proposed approach achieves 97.3% detection accuracy.

In [14], Fredrikson *et al.* proposed an approach which focuses on automatically extracting optimal specifications for behavior based malware detection. They use graph mining (leap mining) and concept analysis algorithms to analyze a set of malicious and benign programs, that extracts significant malicious behaviors, and creates an optimally discriminative specification. The experimental evaluation is done on 912 malware samples, collected using an internet honeypot and 49 benign samples. The proposed approach achieves 86% detection accuracy for previously unknown malware samples with 0% false positive rate. The comparison of their technique with commercial anti-virus software detectors shows that their proposed approach outperforms commercial software.

Chapter 3

Proposed Approach

In this section, we present the details of our proposed graph theoretic approach towards malware detection. We first provide an overview of its architecture, which consists of three main modules: (1) graph construction, (2) feature extraction, and (3) detection.

3.1 Overview

We now introduce our graph-theoretic approach to detect malware from instruction or call sequences. Figure 3.1 shows the architecture of our proposed approach, which consists of three main modules. In the first module, we map the instruction or call sequence of an executable program to a graph. The next module extracts features from the constructed graphs at three levels: (1) vertex level, (2) sub-graph level, and (3) graph level. These features act as footprints of the behavior of an executable program and are leveraged in the detection module to differentiate between benign and malware programs.

3.2 Graph Construction

The prior solutions for behavioral malware detection can be broadly classified into two categories: static analysis (*e.g.* [9,10]) and dynamic analysis (*e.g.* [3,6,17,20]). Depending on the category of solution, static or dynamic, the input is a sequence of

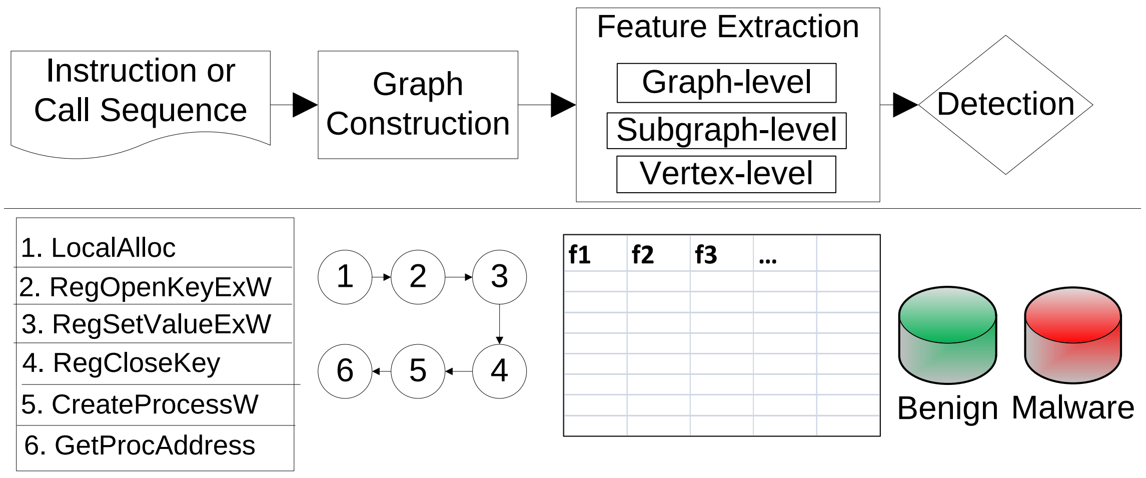


Figure 3.1: Architecture diagram of our proposed approach. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this thesis.

disassembled code instructions or system calls. A sequence of disassembled code instructions or system calls can be mapped to a multi-digraph where vertices represent elements of the sequence and directed edges appear between consecutive sequence elements. Let $G = (V, E)$ denote the graph, where V is the set of vertices and E is the set of directed edges. The graph corresponding to a sequence of length l will have the edge set, such that $|E| = l - 1$. Each edge in the edge set has an associated time index, which is defined by the function $T(e_{ij})$, where e_{ij} is an edge incident on vertex v_j from vertex v_i. In the rest of this paper, we will refer to the constructed graphs as *behavior graphs.*

3.3 Feature Extraction

We characterize the properties of behavior graphs in terms of feature sets that capture their structure at different levels of granularity. More specifically, we extract features quantifying their behavior at three levels: (1) vertex level, (2) sub-graph level, and (3) graph level. The details of these feature sets are separately provided below.

Vertex-level Features

The vertex-level features can be classified into the following categories: degree, path, and connectivity. Important degree based features that we analyze in our proposed approach are: in-degree, out-degree, degree, and reciprocity. Important path based features that we analyze in our proposed approach are: betweenness centrality and closeness centrality. Important connectivity based features that we analyze in our proposed approach are: number of triangles, clustering coefficient, and eigenvector centrality. These features are separately computed for all unique vertices (representing instructions or API calls) in the behavior graphs. Below we formally define these properties.

- **Degree:** The degree of a vertex is defined as the number of edges incident on it. The degree δ_i of a vertex v_i is defined as:

$$\delta_i = \left| \bigcup_{\forall j=i \vee k=i} e_{jk} \right| \tag{3.1}$$

  where e_{jk} denotes an inwards or outwards edge between vertices v_j and v_k.

- **In-Degree:** The degree of a vertex is defined as the number of inwards edges incident on it. The in-degree $\delta_{\downarrow_i}$ of a vertex v_i is defined as follows.

$$\delta_{\downarrow_i} = \left| \bigcup_{\forall k=i} e_{jk} \right| \tag{3.2}$$

- **Out-Degree:** The degree of a vertex is defined as the number of outwards edges incident on it. The out-degree $\delta_{\uparrow_i}$ of a vertex v_i is defined as follows.

$$\delta_{\uparrow_i} = \left| \bigcup_{\forall j=i} e_{jk} \right| \tag{3.3}$$

- **Reciprocity:** The reciprocity of a vertex is defined as the ratio of its out-degree to its in-degree. The reciprocity R_i of a vertex v_i is defined as:

$$R_i = \delta_{\uparrow_i} / \delta_{\downarrow_i} = \left| \bigcup_{\forall j=i} e_{jk} \right| / \left| \bigcup_{\forall k=i} e_{jk} \right| \tag{3.4}$$

where e_{jk} denotes an inwards or outwards edge between vertices v_j and v_k.

- **Number of Triangles:** A triangle is defined as the subset of any three vertices in a graph that are completely connected. The triangle count $\mathit{\Delta}_i$ of a vertex v_i is defined as the number of triangles that contain the given vertex as one of their vertices. Let $\mathit{\Gamma}_i$ denote the set of vertices that a vertex v_i is connected to then its triangle count $\mathit{\Delta}_i$ is defined as:

$$\mathit{\Delta}_i = \left| \bigcup_{v_j, v_k \in \mathit{\Gamma}_i} e_{jk} \right| \tag{3.5}$$

- **Clustering Coefficient:** The clustering coefficient of a vertex is defined as the ratio of number of triangles it is a part of to the total number of possible triangles. If δ_i denotes the degree of a vertex and T_i denotes the number of triangles it is a part of then its clustering coefficient C_i is defined as:

$$C_i = \frac{\mathit{\Delta}_i}{\binom{\delta_i}{2}} = \frac{2\mathit{\Delta}_i}{\delta_i(\delta_i - 1)} \tag{3.6}$$

- **Eigenvector Centrality:** The eigenvector centrality of a vertex is a measure of its importance in a network. To define eigenvector centrality e_i of vertex v_i, let $\mathbf{W}$ denote the adjacency matrix of the graph G where $w_{i,j}$ is 1 if an edge exists between vertices v_i and v_j, and 0 otherwise.

$$e_i = \frac{1}{\lambda} \sum_{j=1}^{N} w_{i,j} x_j \tag{3.7}$$

where λ is the principal eigenvalue of matrix $\mathbf{W}$.

- **Betweenness Centrality:** The betweenness centrality of a vertex is defined as the fraction of all pair shortest paths, except those originating or terminating at it, that pass through it. Let P_{jk} denote the shortest path from vertex v_j to vertex v_k, where $P_{jk} = (v_j, v_l, v_m, v_n, ..., v_k)$. The betweenness centrality b_i of a vertex v_i is defined as:

$$b_i = \frac{2I(P_{jk}, i)}{|V|(|V| - 1)} \tag{3.8}$$

where $I(P_{jk}, i)$ is an indicator function such that $I(P_{jk}, i) = 1$ when $v_i \in P_{jk}$ and $I(P_{jk}, i) = 0$ when $v_i \notin P_{jk}$.

- **Closeness Centrality:** The closeness centrality of a vertex is defined as the average length of shortest paths to all vertices reachable from it. Let $|P_{ij}|$ denote the shortest path length from vertex v_i to vertex v_j. The closeness centrality c_i of a vertex v_i is defined as follows.

$$c_i = \frac{\sum_{j=1}^{N} |P_{ij}|}{|V|} \tag{3.9}$$

Sub-graph level Features

Let $P(x_1, x_2, ..., x_n)$ denote the probability of finding a particular n-gram $x_1, x_2, ..., x_n$ in a given sequence and k denote the sample space size of each element in the sequence. The size of sample space for a sequence containing n elements is k^n. Equivalently, it can be represented as the joint probability $P(x_1 \cap x_2 ... \cap x_n)$. Using Bayes theorem, this joint probability can be defined by a combination of the marginal and conditional probabilities. More specifically, $P(x_1 \cap x_2 ... \cap x_n) = P(x_1 | x_2 ... \cap x_n) P(x_2 ... \cap x_n)$. In fact, the conditional probability contains more precise information due to its reduced sample space compared to the joint probability. These conditional n-gram probabilities can be conveniently represented using a discrete time Markov chain of order $n-1$ containing k^{n-1} states. Note that each conditional n-gram sequence corresponds to a unique state in the multi-order Markov chain. Now let us assume that the presence of a state in Markov chain is represented by a binary indicator random variable $I_i, i = 1, 2, ..., k^{n-1}$, where k^{n-1} is the total number of states of the Markov chain of order $n-1$. Hence, $P(I_i = 1)$ represents the probability for the presence of state X_i.

It is important to select an appropriate order for a Markov chain model. It is also important to note that we are typically interested in employing a single Markov chain to model a set of multiple sequences $\mathbb{S}$. Here let $|\mathbb{S}|$ denote the size of the set of sequences we want to model. For each sequence, autocorrelation is a well-known heuristic for selecting appropriate order for its Markov chain model [7]. For a given lag t, the autocorrelation function of a sequence, S_m (where m is the index), is defined as:

$$\rho[t] = \frac{E\{S_0 S_t\} - E\{S_0\}E\{S_t\}}{\sigma_{S_0}\sigma_{S_t}}, \tag{3.10}$$

where $E(S_i)$ and σ_{S_i} respectively represent the expectation and standard deviation of S at lag i. The value of the autocorrelation function lies in the range $[-1, 1]$, where $|\rho[t]| = 1$ indicates perfect correlation at lag t and $\rho[t] = 0$ means no correlation at lag t. The minimum value of lag t_{min} for which $\rho[t_{min}]$ falls inside the 95% confidence interval band is selected to be the appropriate order for a Markov chain. For a set of multiple sequences, let $\mathbb{T}$ denote the set of selected orders as per the aforementioned criterion. We select the maximum value in $\mathbb{T}$, denoted by $\mathbb{T}_{max}$, as the order of a single Markov chain model that we want to employ.

As mentioned earlier, the number of states in a Markov chain increase exponentially for higher orders and so does the complexity of the underlying model. Furthermore, higher order Markov chains require a large amount of training data to identify a subset of states that actually appear in the training data. In other words, a Markov chain model trained with limited data is typically sparse. To overcome these challenges, we can combine multiple states in a higher order Markov chain to reduce its number of states. We are essentially using states from lower order Markov chains as we combine different states in a multi-order Markov chain. We also need to establish a criterion to combine states in a multi-order Markov chain.

Towards this end, we use the concept of "typicality" of Markov chain states [7]. The basic idea of typicality is that we can identify a "typical" subset of Markov chain states by generating its realizations. Before delving into further details, we first state the well-known "typicality theorem" below:

For any stationary and irreducible Markov process X and a constant c, the sequence $x_1, x_2, ..., x_m$ is almost surely (n, ϵ)-typical for every $n \leq c \log m$ as $m \to \infty$. A sequence $x_1, x_2, ..., x_m$ is called (n, ϵ)-typical for a Markov process X if $\hat{P}(x_1, x_2, ..., x_n) = 0$, whenever $P(x_1, x_2, ..., x_n) = 0$, and

$$\left|\frac{\hat{P}(x_1, x_2, ..., x_n)}{P(x_1, x_2, ..., x_n)} - 1\right| < \epsilon, \text{ when } P(x_1, x_2, ..., x_n) > 0.$$

Here $\hat{P}(x_1, x_2, ..., x_n)$ and $P(x_1, x_2, ..., x_n)$ are the empirical relative frequency and the actual probability of the sequence $x_1, x_2, ..., x_n$, respectively. In other words,

$$\hat{P}(x_1, x_2, ..., x_n) \approx P(x_1, x_2, ..., x_n).$$

A direct consequence of the aforementioned typicality theorem is that we can empirically identify "typical" sample paths of arbitrary length for a given Markov process. Towards this end, we can generate realizations (or sample paths) of arbitrary lengths from the transition matrix of the Markov process. By generating sufficiently large number of sample paths of a given length, we can accurately identify a relatively small subset of sample paths that are typical.

Using this criterion, we select a subset of top-10000 typical states $\mathbb{X}_{1000}$ as potential features, whose lengths vary in the range $[0, \mathbb{T}_{max}]$. To further cut down the number of sub-graph level features (up to 100), we use an information-theoretic measure called information gain to rank features [11]. Information gain is used to quantify the differentiation power of features (Markov chain states in our case). In this context, information gain is the mutual information between a given feature X_i and the class variable Y. For a given feature X_i and the class variable Y, the information gain of X_i with respect to Y is defined as:

$$IG(X_i; Y) = H(Y) - H(Y|X_i), \tag{3.11}$$

where $H(Y)$ denotes the marginal entropy of the class variable Y and $H(Y|X_i)$ represents the conditional entropy of Y given feature X_i. In other words, information gain quantifies the reduction in the uncertainty of the class variable Y given that we have complete knowledge of the feature X_i. Note that, in this paper, the class variable Y is $\{\text{Benign}, \text{Malware}\}$ for our application. Using information gain, we finally select a subset of top-100 typical states $\mathbb{X}_{100}$ as features.

Graph-level Features

We now separately define the graph-level features below.

- **Clique Number:** A clique is a sub-graph such that all vertices in it are directly connected to each other by an edge. The clique number ω of a graph $G = (V, E)$ is defined as the number of vertices in its largest clique.

- **Average Clustering Coefficient:** The average clustering coefficient $\bar{C}$ of a graph $G(V, E)$ is defined as:

$$\bar{C} = \frac{1}{|V|} \sum_{i=1}^{|V|} C_i \tag{3.12}$$

- **Diameter:** The diameter D of a graph $G(V, E)$ is defined as:

$$D(G) = \max_{\forall j,k}(|P_{jk}|), \tag{3.13}$$

  where P_{jk} is the shortest path length between vertices v_j and v_k.

- **Average Path Length:** The average path length l of a graph $G(V, E)$ is defined as:

$$l(G) = \frac{\sum_{\forall j,k}(|P_{jk}|)}{|V|(|V|-1)}, \tag{3.14}$$

  where P_{jk} is the shortest path length between vertices v_j and v_k.

3.4 Detection

We can now jointly use all features to differentiate between benign and malware programs. In particular, given separate feature sets characterizing vertex-level, sub-graph level, and graph-level properties, we first separately classify behavior graphs by deploying a machine learning classifier for each of them. Given the classification results for each of the feature sets, we then use a meta machine learning classifier for final classification. In this study, we use Naïve Bayes algorithm as the base- and the meta-classifier. Naïve Bayes is a popular probabilistic classifier that has been widely used for problems like text and malware classification and is known to outperform more complex techniques in terms of accuracy [19]. It trains using two sets of probabilities: the prior, which represents the marginal probability $P(Y)$ of the class variable Y; and the a-priori conditional probabilities $P(X_i|Y)$ of the features X_i given the class variable Y. As previously explained, these probabilities can be computed from the training set. Now, for a given test behavior graph with observed features X_i, $i = 1, 2, ..., n$, the *a-posteriori* probability $P(Y|X^{(n)})$ can be computed for both classes $Y \in \{\text{Benign}, \text{Malware}\}$, where $X^{(n)} = (X_1, X_2, ..., X_n)$ is the vector of observed features in the test cascade under consideration:

$$P(Y|X^{(n)}) = \frac{P(X^{(n)}, Y)}{P(X^{(n)})} = \frac{P(X^{(n)}|Y)P(Y)}{P(X^{(n)})}. \tag{3.15}$$

The Naïve Bayes classifier then combines the a-posteriori probabilities by assuming conditional independence (hence the "naïve" term) among the features.

$$P(X^{(n)}|Y) = \prod_{i=1}^{n} P(X_i|Y). \tag{3.16}$$

Although the independence assumption among features makes it feasible to evaluate the a-posteriori probabilities with much lower complexity, it is unlikely that this assumption truly holds all the time. For our study, we mitigate the effect of the independence assumption by pre-processing the features using the well-known Karhunen-Loeve Transform (KLT) to un-correlate them [12].

Chapter 4

Experimental Evaluation

In this section, we first provide details of the dataset used to evaluate our proposed approach. We then provide the classification results of individual vertex-level, sub-graph level, and graph-level feature sets along with those of the meta-classifier. We also provide the results of obfuscation analysis in this section.

4.1 Dataset

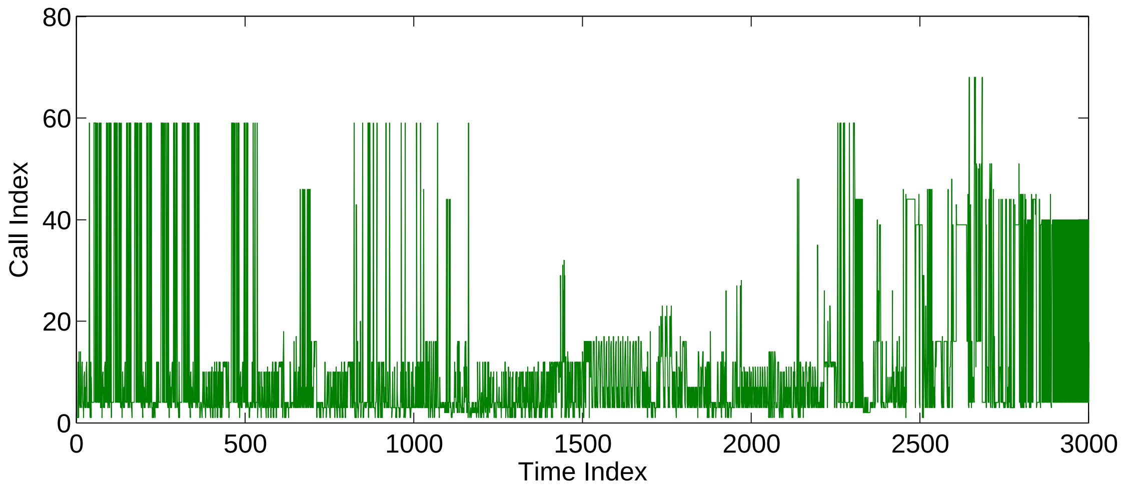
The behavior of a program can be characterized using either of the two well-known approaches: static and dynamic analysis. For static analysis, off-the-shelf executable program loaders can be used to construct a sequence of code instructions. IDA Pro [13] and CodeSurfer [2] are two well-known software to construct instruction call graphs. For dynamic analysis, an open source processor emulator, such as QEMU [5], can be used to log system calls in case of the Unix operating system and Application Programming Interface (API) calls in case of the Windows operating system. QEMU efficiently program calls through dynamic translation and caching [5]. Other analysis systems such as TTAnalyze [3] and Panorama [20] have been built on the top of QEMU.

To evaluate the effectiveness of our proposed approach, we have collected the API call sequences of benign and malware programs. A set of one hundred programs from freshly installed Windows are chosen to represent benign class. We have chosen a diverse subset of more than three hundred malware programs from the dataset provided by OffensiveComputing.net. These malware programs are further categorized into the categories of virus, trojans, and worm based on their functionality. Table 4.1 shows basic statistics of the API call data used in this study. In our data set, we log more than 51 million API calls belonging to 237 unique API functions. We note from Table 4.1 that benign programs tend to have larger number of calls to more unique API functions on average than malware programs. This shows that benign programs tend to have more complex functionality than malware programs in general. Within malware programs, we observe that trojans have the largest number of calls to most unique API functions on average. On the other hand, worms have the smallest number of calls to least unique API functions on average. This observation is linked to the functionality of program, where worms are concisely implemented to facilitate automated propagation and trojans covertly perform some backdoor activity with some benign upfront functionality. Finally, it is interesting to compare the average entropy of API call distributions for benign and malware programs. We note that benign programs have larger average entropy than malware programs, which points to more complex functionality of benign programs than malware programs.

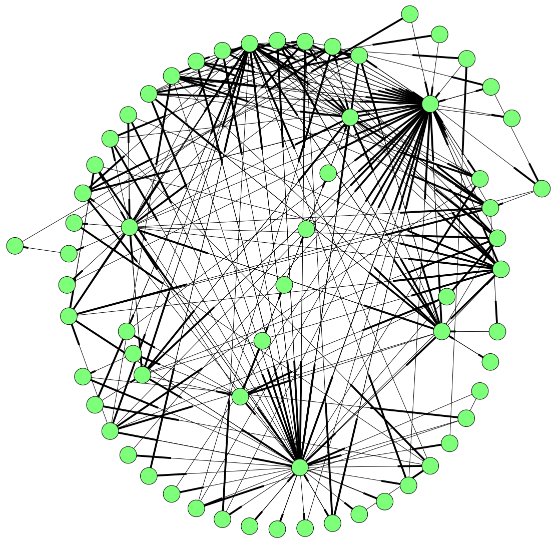
Table 4.1: Basic statistics of the API call dataset used in this study.

Program Category	**Average Number of Programs**	**Average Number of Calls**	**Average Number of Unique Calls**	**Average Entropy of Call Dist.**
Benign	100	$271,694$	135	0.96
Malware	312	$77,250$	43	0.57
Virus	104	$59,086$	43	0.57
Trojan	104	$152,527$	47	0.54
Worm	104	$33,232$	39	0.60

As mentioned in Section 3, instruction or call sequences can be mapped to behavior graph. Figure 4.1 shows the plots of timeseries and behavior graph of an example benign program. Likewise, Figure 4.2 shows the plots of timeseries and behavior



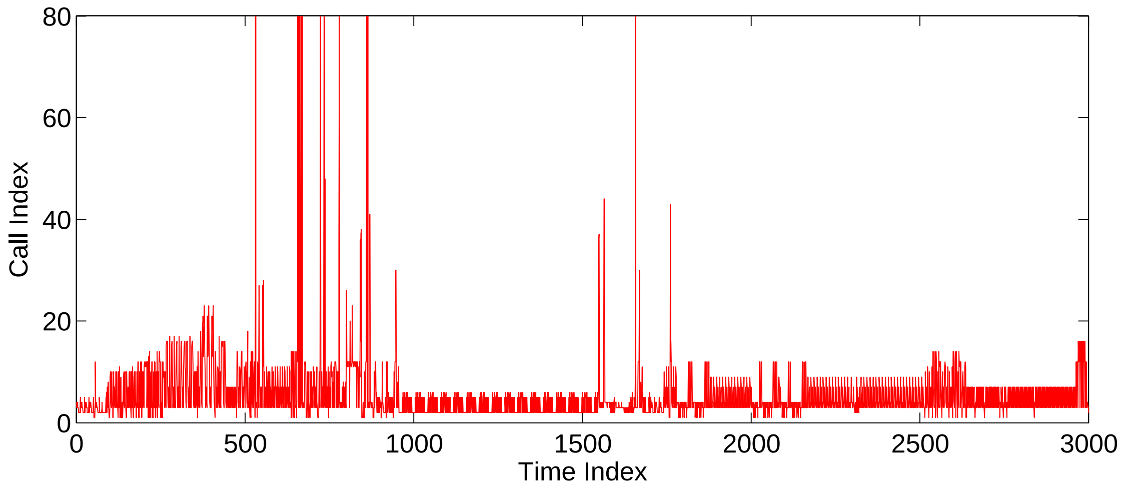
(a) Timeseries



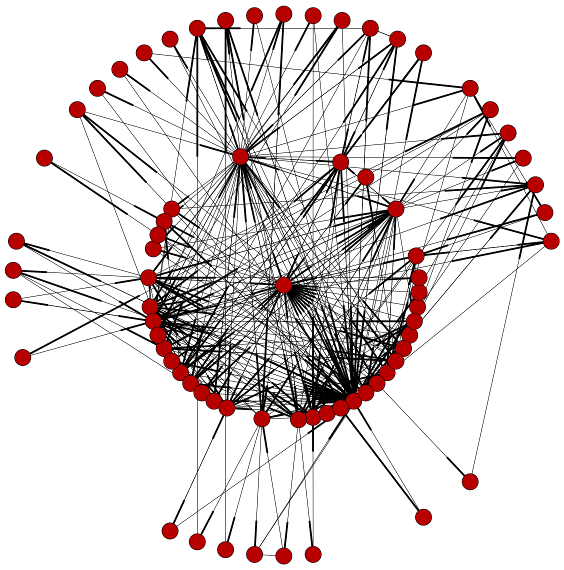
(b) Behavior Graph

Figure 4.1: Timeseries and behavior graph of an example benign program.

graph of an example malware program. The indices of calls are fixed in the alphabetical order in the timeseries plots. We visually observe interesting patterns in the timeseries and behavior graph of benign and malware programs. For instance, we observe repetitive call subsequence blocks in timeseries of both benign and malware programs. However, the sizes of blocks are significantly smaller for benign programs



(a) Timeseries



(b) Behavior Graph

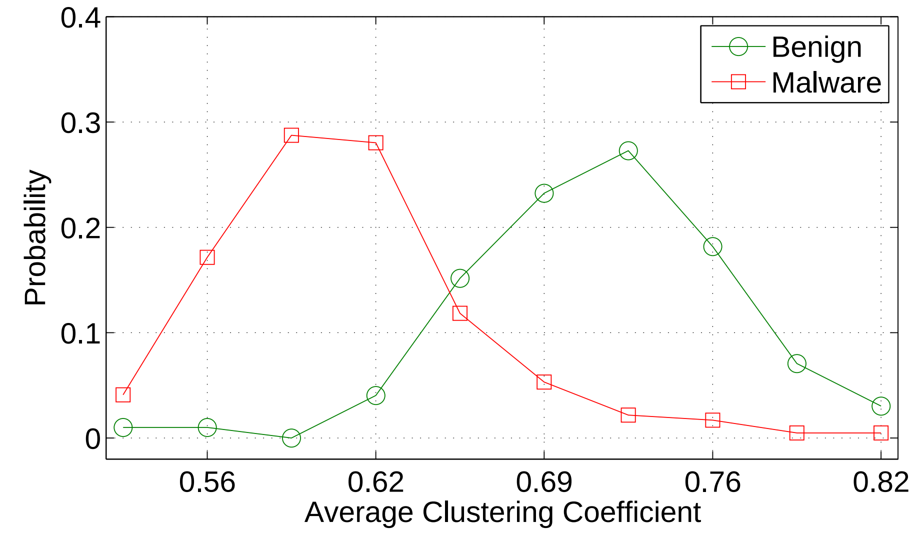
Figure 4.2: Timeseries and behavior graph of an example malware program.

as compared to malware programs. Note that the layout of behavior graphs in Figures 4.2 and 4.1 is radial in nature. In a radial layout, we randomly choose a call as a center vertex and the remaining vertices are put in concentric circles based on the distance from the center vertex. A visual comparison of benign and malware behavior graphs also highlights interesting differences. For instance, we note that the degree distribution of behavior graph of malware program is significantly more skewed compared to the behavior graph of benign program. In addition, we observe significantly deeper branches of vertices in the behavior graph of example benign program compared to that of example malware program. We aim to capture such differences in an automated fashion by leveraging the three feature sets encompassing properties of behavior graphs at various levels (graph, sub-graph, and vertex). Below we explore the distribution of these feature sets across benign and malware programs.

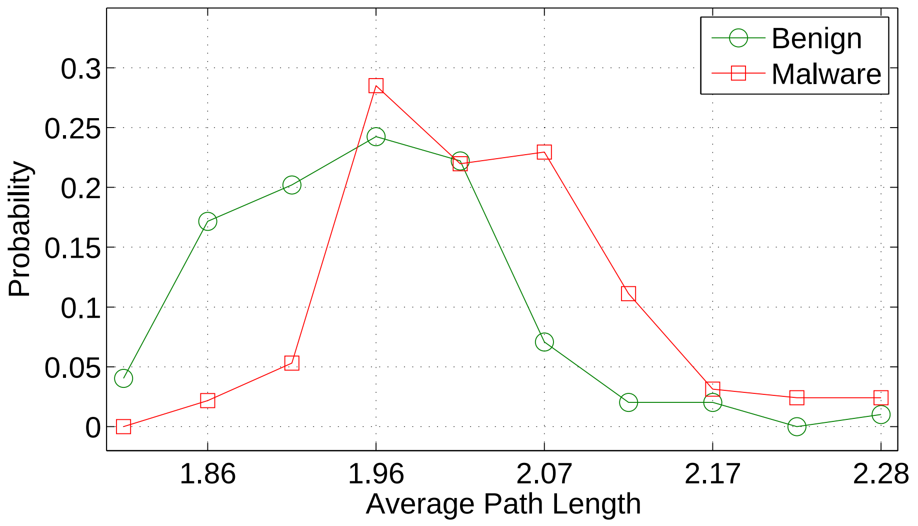
4.2 Feature Analysis

Figure 4.3 shows the distribution of two graph level features. In Figure 4.3(a), we observe that behavior graphs of benign programs tend to have larger values of average clustering coefficient compared malware programs. Recall from Section 3 that average clustering coefficient is a measure of connectivity of a graph, where larger values of average clustering coefficient indicate more connectivity and smaller values indicate less connectivity. The class-wise distribution of average clustering coefficient shows that benign behavior graphs are more tightly connected than malware behavior graphs. This observation also provides an insight into the peculiar functional characteristics of malware programs. Figure 4.3(b) shows the class-wise distribution of average path length of benign and malware programs. We would expect the average path length of benign behavior graphs to be smaller than that of malware behavior graphs because the former are more tightly connected (quantified by clustering coefficient) compared to the latter. However, we observe an opposite trend in Figure 4.3(b) where malware behavior graphs having larger average path length than benign behavior programs. This indicates that benign behavior graphs are tightly connected such that average path length are relatively larger. This observation is explained by the reasoning that benign behavior graphs have tightly connected clusters which have fewer connections across them.

We now analyze the properties of sub-graph level features of behavior graphs.



(a) Average Clustering Coefficient



(b) Average Path Length

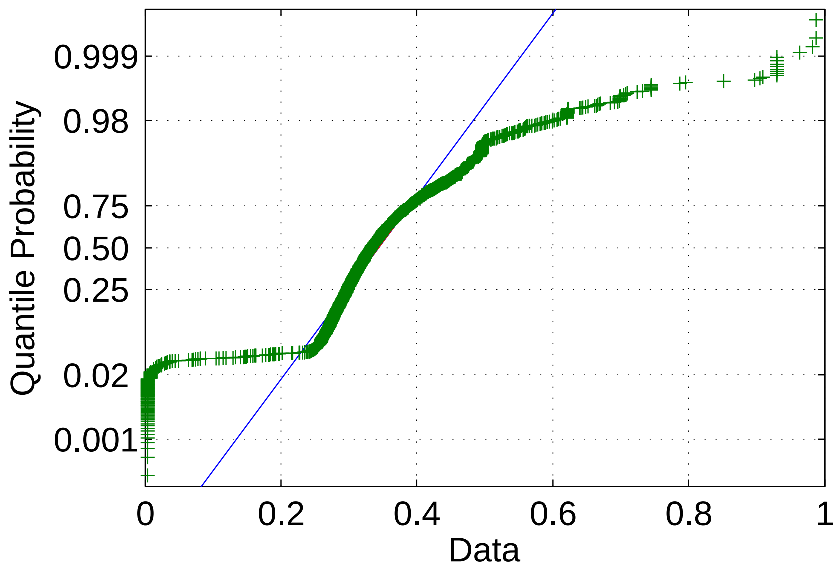
Figure 4.3: Distribution of some graph level features.

Recall from Section 3 that we can identify a typical sample paths of Markov chain model of behavior graphs. The presence or absence of these typical sample paths act as sub-graph level binary features. Here we are interested in studying the distribution of their empirically estimated probabilities. Ideally, we want the sample paths in typical set to have high probabilities. Figure 4.4 shows the Q-Q probability plot of sub-graph level features. We observe that most sample paths have moderate to high occurrence probabilities which is desirable. This shows that we can successfully capture variable length Markov chain sample paths have

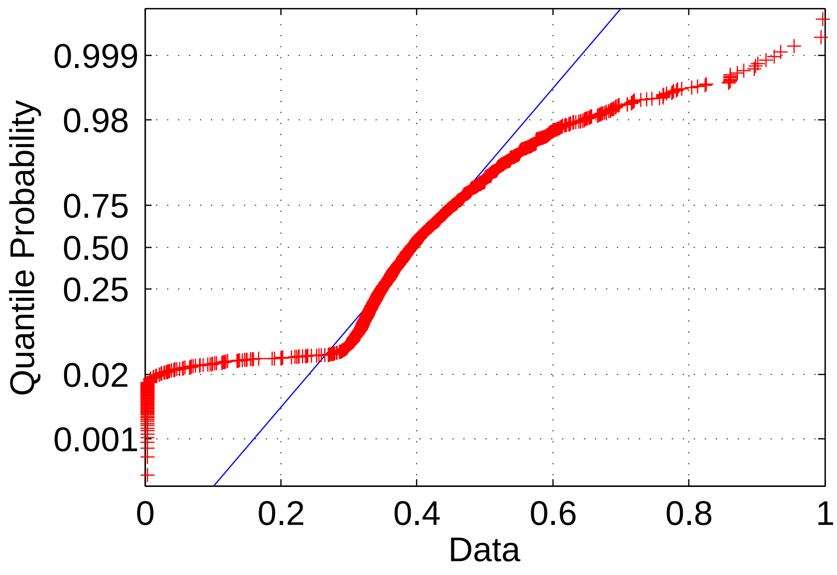
We now investigate the class-wise distribution of some vertex level features. Figures 4.5, 4.6 and 4.7 show the distribution of some vertex level features including out-degree, number of triangles, and closeness centrality for `LocalAlloc` API call. This particular API call is important for memory management and allocates the specified number of bytes from the heap. A common trend observed for all of these vertex level features is that we observe smaller values for malware programs compared to benign programs. Similar to the analysis of class-wise distribution of graph level features, these observations provide insight into the differences between benign and malware program behavior.

4.3 Classification Results

We now provide the classification results of different feature sets proposed in Section 3. To systematically evaluate the effectiveness of these feature sets in classifying benign and malware programs, we first evaluate standalone feature sets and then evaluate their all possible combinations. Figure 4.8(a) shows the Receiver Operating Characteristics (ROC) plots for the Naïve Bayes classifier with standalone feature sets. Figure 4.8(b) shows the ROC plots for the Naïve Bayes classifier with combinations of feature sets. ROC performance is characterized in terms of false positive rate (fraction of benign programs falsely detected as malware programs) and true positive rate (fraction of malware programs correctly detected as malware programs). In ROC plot, the x-axis represents the false positive rate and the y-axis represents the true positive rate. An ideal classifier will approach the operating point in the top-left corner with 1.0 true positive rate and 0.0 false positive rate. Both true positive rate and false positive rate are jointly incorporated in a single metric called Area Under ROC Curve (AUC). The AUC for an ideal classifier is 1.0. Another alternate measure of classification performance is called precision, which is defined as the fraction



(a) Benign transition matrix



(b) Malware transition matrix

Figure 4.4: Q-Q probability plot of sub-graph level features.

of correctly classified instances.

In Figure 4.8(a), we observe that ROC curve for sub-graph feature set is the

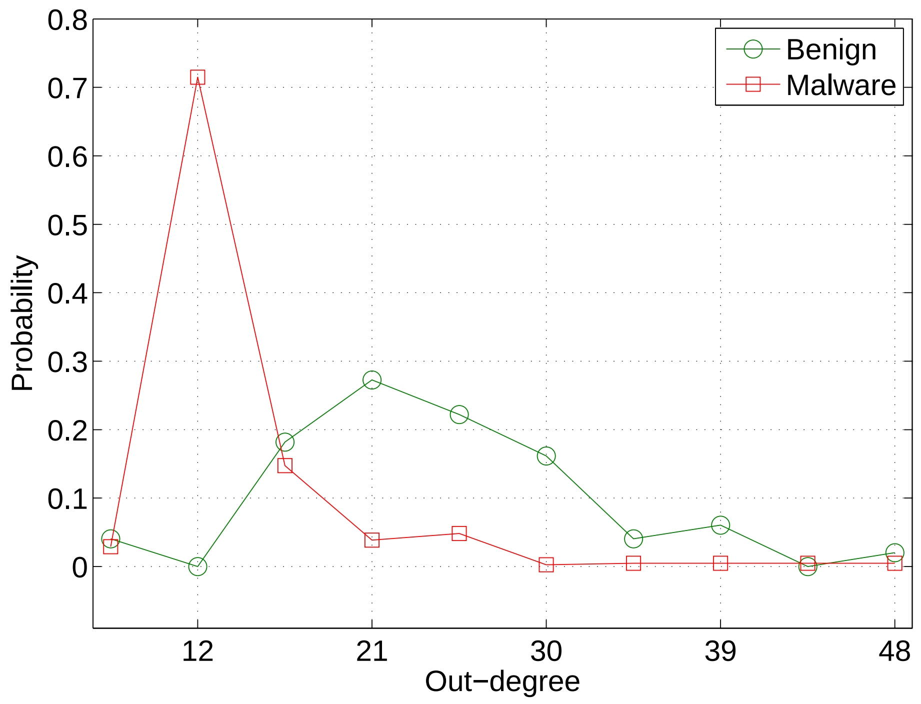


Figure 4.5: Out-degree: vertex level feature of `LocalAlloc` API call.

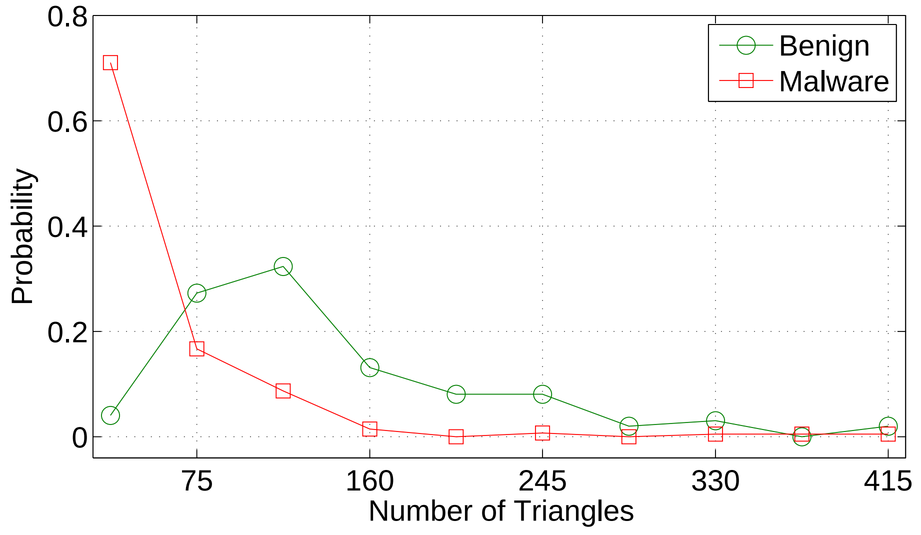


Figure 4.6: Number Of Triangles: vertex level feature of `LocalAlloc` API call.

closest to the top-left operating point. It is followed by the vertex feature set and the ROC curve of graph feature set is furthest from the top-left operating point. In Figure 4.8(b), we observe that combinations of feature sets does improve the classification performance. In fact, the best classification performance is achieved

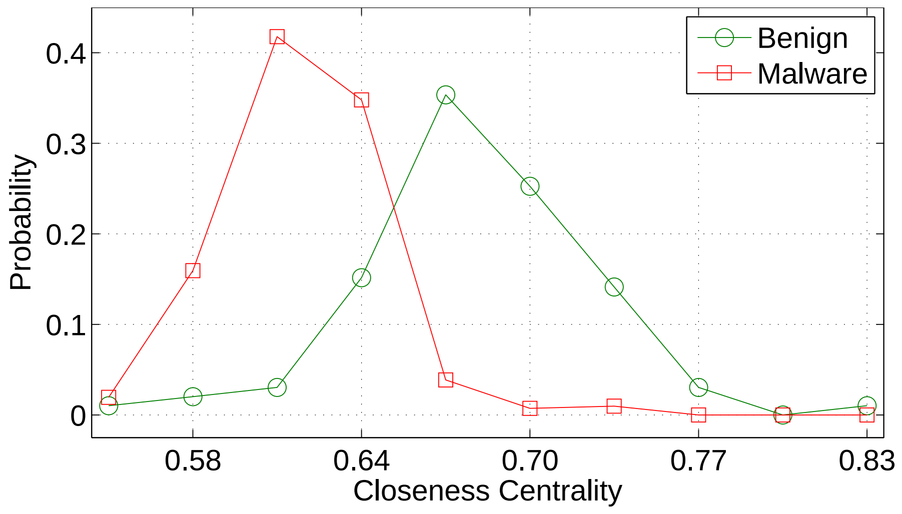


Figure 4.7: Closeness Centrality: vertex level feature of `LocalAlloc` API call.

when all feature sets are combined. Table 4.2 also provides the detailed classification accuracy results for different feature sets. It is interesting to note that combining sub-graph and graph feature sets actually degrades the classification performance compared to standalone feature sets. This is because redundant features can confuse the classifier resulting in degraded classification accuracy. However, we do note that combining all feature sets consistently provides the best classification performance in terms of all metrics. With combined feature set, our proposed approach achieves precision up to 99.5%.

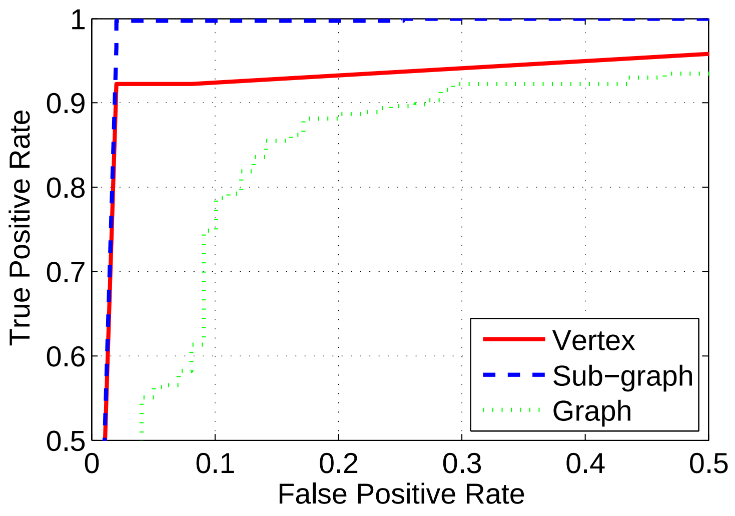
4.4 Obfuscation Analysis

We have also done obfuscation analysis on our scheme so that we can observe the effect of random removal/addition of API calls edges; that is in our case to the graphs. We specifically want to see whether a crafty malware writer would be successful in trying to break the sequence of API calls.

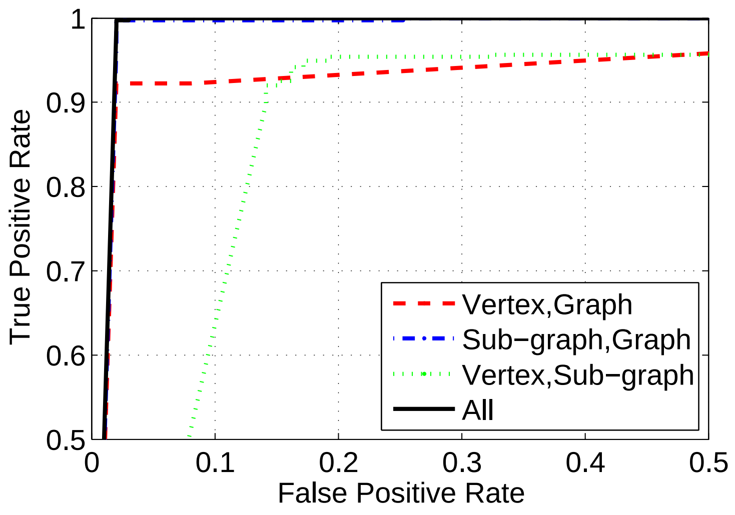
Figure 4.9 shows the AUC of the obfuscation analysis on our scheme. We obfuscated API calls ranging from 10 - 100. The accuracy gracefully degrades but no drastic effect even if 40 API calls are obfuscated.

Table 4.2: Classification accuracy results for different feature sets.

Feature Set	**AUC**	**True Positive Rate**	**False Positive Rate**	**Precision**
Graph	0.892	0.923	0.343	0.918
Vertex	0.949	0.923	0.02	0.995
Sub-graph	0.989	0.983	0.02	0.995
Vertex, Graph	0.949	0.923	0.02	0.995
Sub-graph, Graph	0.989	0.983	0.02	0.995
Vertex, Sub-graph	0.89	0.92	0.141	0.965
All	**0.993**	**0.998**	**0.02**	**0.995**

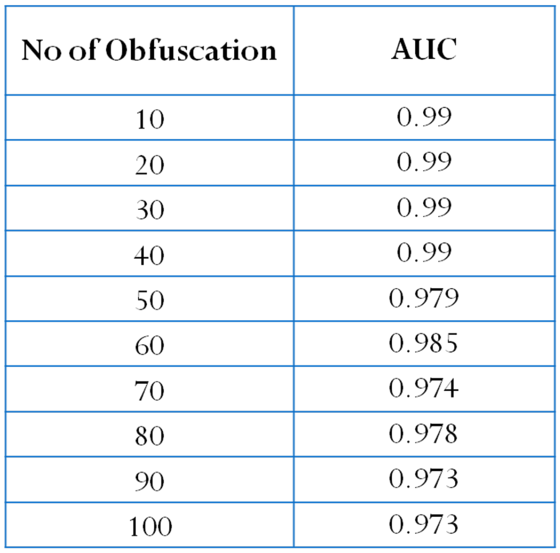


(a) Individual Feature Sets



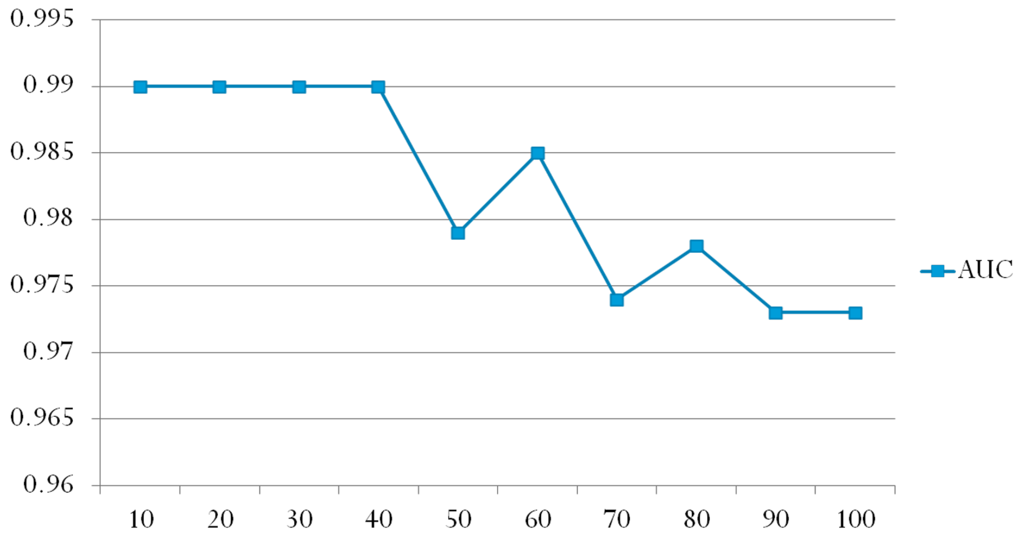
(b) Combinations of Feature Sets

Figure 4.8: ROC plots of classification accuracy.



No of Obfuscation	AUC
10	0.99
20	0.99
30	0.99
40	0.99
50	0.979
60	0.985
70	0.974
80	0.978
90	0.973
100	0.973

(a) Table of AUC for obfuscating API's ranging from 10 to 100



(b) AUC's plotted for different obfuscation values

Figure 4.9: AUC of Obfuscation Analysis

Chapter 5

Conclusion

In this paper, we have proposed a graph theoretic malware detection technique that operates on the program instruction sequences or the API call sequences. The instruction sequences are mapped to a graph from which distinguishing feature sets are extracted at three levels, namely vertex level, sub-graph level, and graph level. These feature sets are then given as an input to machine learning algorithms, both individually and as a whole, to generate training models. The experimental results show that the proposed approach achieves approximately 100% detection rate and 0% false positive rate when the three feature sets are jointly used.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Faraz Ahmed, Haider Hameed, M. Zubair Shafiq, and Muddassar Farooq:. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In *ACM Workshop on Security and Artificial Intelligence*, 2009.

[2] Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *Workshop on Inspection in Software Engineering*, 2001.

[3] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware. In *European Institute for Computer Antivirus Research Annual Conference*, 2006.

[4] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[6] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *ACM Conference on Mobile systems, Applications, and Services (MobiSys)*, 2008.

[7] Pierre Bremaud. *Markov Chains*. Springer, 2008.

[8] C. Chen, C. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining graph patterns efficiently via randomized summaries. In *ACM Conference on Very Large Data Bases*, 2009.

[9] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *USENIX Security Symposium*, 2003.

[10] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.

[11] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[12] R.D. Dony. *The Transform and Data Compression Handbook, Chapter 1*. CRC Press, 2001.

[13] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler.* No Starch Press, 2011.

[14] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symposium on Security and Privacy*, 2010.

[15] Rafiqul Islam, Ronghua Tian, Lynn M Batten, and Steve Versteeg. Differentiating malware from cleanware using behavioural analysis. In *IEEE International Conference on Malicious and Unwanted Software*, 2010.

[16] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, 2009.

[17] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: Using system-centric models for malware protection. In *ACM conference on Computer and Communications Security (CCS)*, 2010.

[18] D. Perry. Here comes the flood or end of the pattern file. *Virus Bulletin VB*, 2008.

[19] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2011.

[20] Heng Yin and Dawn Song. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM conference on Computer and Communications Security (CCS)*, 2007.