5-1-2014

Integrating Multiple Data Views for Improved Malware Analysis

Blake Anderson

Blake Anderson

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Jedidiah Crandall

, Chairperson

Terran Lane

Stephanie Forrest

Joshua Neil

Niall Adams

Integrating Multiple Data Views for Improved Malware Analysis

by

Blake Harrell Anderson

B.S., Rhodes College, 2006

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2014

Dedication

To caffeine (and my family, friends, and teachers)

Acknowledgments

While developing the material in this dissertation, I have had the pleasure of meeting a lot of very talented individuals. My first semester at UNM I started as a teaching assistant for Terran Lane, and later became one of his research assistants and advisees. He helped guide me through the art and science of machine learning, and was also there for me on a personal level when things seemed hopeless. I owe a lot of my development as a research scientist to him.

When I came to Los Alamos, I started working on the malware problem with the Advanced Computing Solutions group under my mentor Joshua Neil. His support throughout my time at Los Alamos has been invaluable. Daniel Quist, the resident malware expert when I arrived, also had an important role steering the early work of the dissertation. He provided the majority of the malware samples used, as well as the dynamic tracing infrastructure that helped to motivate a lot of this work. Curtis Hash wrote the CodeVision framework allowing for the classification portion of this dissertation to be operationalized. I would also like to thank Curtis Storlie who provided great feedback on all of the methods.

When Terran left UNM for Google, I had to find a new dissertation chair to help me finish the last stages of my PhD. Jedidiah Crandall was gracious enough to fill that role, and I am incredibly thankful for him taking the time to help me navigate the final stages of bureaucracy. I would also like to thank my talented committee. Jedidiah Crandall, Niall Adams, Stephanie Forrest, Terran Lane, and Joshua Neil have all provided inspiration and meaningful feedback allowing this dissertation to become a much stronger piece of work.

Integrating Multiple Data Views for Improved Malware Analysis

by

Blake Harrell Anderson

B.S., Rhodes College, 2006

Ph.D., Computer Science, University of New Mexico, 2014

Abstract

Malicious software (malware) has become a prominent fixture in computing. There have been many methods developed over the years to combat the spread of malware, but these methods have inevitably been met with countermeasures. For instance, signature-based malware detection gave rise to polymorphic viruses. This "arms race" will undoubtedly continue for the foreseeable future as the incentives to develop novel malware continue to outweigh the costs.

In this dissertation, I describe analysis frameworks for three important problems related to malware: classification, clustering, and phylogenetic reconstruction. The important component of my methods is that they all take into account multiple *views* of malware. Typically, analysis has been performed in either the static domain (e.g. the byte information of the executable) or the dynamic domain (e.g. system call traces). This dissertation develops frameworks that can easily incorporate well-studied views from both domains, as well as any new views that may become popular in the future. The only restriction that must be met is that a positive semidefinite similarity (kernel) matrix must be defined on the view, a restriction that is easily met in practice.

While the classification problem can be solved with well known multiple kernel learning techniques, the clustering and phylogenetic problems required the development of novel machine learning methods, which I present in this dissertation. It is important to note that although these methods were developed in the context of the malware problem, they are applicable to a wide variety of domains.

Contents

Contents

Contents

Contents

Contents

List of Figures

List of Tables

List of Tables

Chapter 1

Introduction

In the second quarter of 2013, McAfee labs catalogued more than 18.5 million new malware samples [76]. Despite the fact that the majority of this new malware is being created through polymorphism and code obfuscation techniques [67], and thus is similar to known malware, it is still not detected by signature-based antivirus programs [25,84]. To meet these challenges, machine learning techniques have been developed that can learn a generalized description of malware and apply this knowledge to classify/cluster new, unseen instances of malware [1,11,65,67].

This dissertation is concerned with representing malicious code in a way that allows classifying and clustering algorithms to be more robust against code obfuscation techniques. I approach this problem in two ways. The first is to find a new data representation which can better capture the information within a given data view, e.g. the dynamic instruction trace or byte sequences of the binary, allowing for a more accurate classification or clustering system. The second approach is to use multiple data views. Different data views provide complementary information about the true nature of a program, but no one data view contains all of this information. By looking at multiple views of a program, such as the dynamic trace and the binary, the program's true intentions become more clear. In this

dissertation, I show that the multiview paradigm does increase the performance on several important problems.

Given the data views and the data representation, I develop and apply machine learning algorithms to solve three important problems in the malware domain: classification, clustering, and phylogenetic reconstruction. To classify new instances as malicious or benign, I begin by defining a *kernel* [22], a positive semi-definite matrix where each entry in the matrix is a measure of similarity between a pair of instances in the dataset, one matrix for each data view. I then use multiple kernel learning [9, 101] to find the weights of each kernel, create a linear combination of the kernels, and finally use a support vector machine [22] to perform classification.

Many researchers have used k-means or hierarchical clustering algorithms to group malware together that exhibit similar functionality. I extend this research by introducing a novel multiple kernel learning algorithm based on a clustering objective function. With my method, I can combine multiple data views in a way that is relevant to the clustering problem.

I also approach clustering in a different way which allows for better attribution of the malware and allows reverse engineers to more quickly understand how a given piece of malware has evolved from known examples of malware. I develop a novel extension to *graphical lasso* [39] which uses multiple views to find a graph where the nodes are instances of malware, and the edges correspond to ancestor/descendant relationships between the malware instances. Graphical lasso starts with the covariance matrices of the data views, which I have found to be good discriminators, and finds a sparse precision matrix defining the conditional independencies in the phylogenetic graph of the given malware samples.

1.1   Overview of the Malware Problem

Malicious programs generally try to achieve one of two goals. The first goal is simply to earn money for the author or sponsoring organization. This can be done by gathering personally identifiable information (PII) from compromised computers or setting up a botnet to ransom corporations with the threat of a distributed denial of service attack (DDoS) [34]. A second goal is data exfiltration, which can be motivated either by economic or intelligence reasons. In either case, there are large incentives for malware authors to continue to develop new threats.

To detect malicious programs, most corporations and home users make use of some type of antivirus program. Many of the current antivirus programs rely on a signature-based approach to classify programs as being either malicious or benign. Signature-based approaches are popular due to their low false positive rates and low computational complexity on the end host, both of which are appealing for daily use. Unfortunately, these schemes can be easily defeated by simple code obfuscation techniques [25, 100]. As polymorphic viruses became more prevalent, non-signature based methods became more attractive [1, 67, 72]. Furthermore, traditional antivirus has little hope of detecting advanced 0-day malware, as this malware has not been previously seen in the wild and will therefore lack a signature.

While detection is the first component of malware analysis, it is more informative to ask several questions that go beyond the simple yes/no answer afforded by a good classification system:

1. What family does the malware belong to?

2. What functionality does the malware posses?

3. What group can the malware's creation be attributed to?

These questions become increasingly important in the context of responding to a malware

incident. A reverse engineer is someone who is assigned the task of understanding and responding to a malware infection, and it is crucial to leverage any previous information gained from reverse engineering samples from the same family or lineage. This decreases the reverse engineering time, and allows for a faster response.

1.2   Malware Domain Contributions

There has been a great deal of work applying machine learning techniques to the malware domain, attempting to solve problems such as classification, clustering, and phylogenetics [11, 45, 66]. The main drawback to many of the previous approaches is that they rely on a single view of the data. Common views of malware data include the disassembled instructions, the bytes in the binary, and the dynamic instructions gathered while the program is executed. Malware writers have great incentives to hide malware from regular users as well as reverse engineers, who are tasked with understanding the malware's functionality and responding with appropriate actions (e.g. creating a signature for antivirus software). The longer a new piece of malware can remain undetected, the more successful it will be.

Because malware can obfuscate itself in many views, in this dissertation I demonstrate a new multiview paradigm for malware analysis. The central thesis is that although malware can easily obfuscate itself in one or more views, obfuscating all views while maintaining malicious intent is significantly more difficult. I extend this multiview paradigm to three important malware problems: classification, clustering, and phylogenetic reconstruction.

In addition to the multiview paradigm for malware research, an important, pragmatic contribution of this dissertation is a new malware classifier that is not based on signatures, but rather fuses several data views using multiple kernel learning to arrive at a more robust classifier. The views are both static, that is, based on information that does not require the program to be executed, and dynamic, based on information collected while the program is running.

1.3   Machine Learning Contributions

The machine learning contributions of this dissertation center around extending the techniques of spectral clustering and graphical lasso (glasso) to incorporate multiple views of the data without the need for a priori information about the views. These extensions were motivated by the malware domain, where the importance of individual views is not known a priori and often changes when presented with a new dataset. However, the method can easily be applied to any domain where multiple views are available. For instance, multiview clustering could be used in the context of webpages where multiple views are derived from the webpage itself and the connections to other webpages [14].

Spectral clustering is a popular method for clustering [74], and I have extended this framework to naturally incorporate multiple views of data. Spectral clustering revolves around the following optimization problem:

$$\min_{U \in \mathbb{R}^{n \times k}} \quad \text{tr}(U^T L U) \tag{1.1}$$
$$\text{s.t.} \quad U^T U = I$$

where L is defined to be the Lapacian, $L = D - K$. K is the adjacency matrix of the data and D is a diagonal matrix of the degrees of each node in K. I begin by defining the multiview Laplacian, $L(\beta) = \sum_i^M \beta_i D_i - \sum_i^M \beta_i K_i$, and extend the optimization framework of spectral clustering to use the multiview Laplacian:

$$\min_{U \in \mathbb{R}^{n \times k}} \quad \text{tr}(U^T L(\beta) U) \tag{1.2}$$
$$\text{s.t.} \quad U^T U = I$$

I demonstrate this methodology on both the unnormalized and normalized Laplacians. A detailed description of this method is described in Chapter 5.

Graphical lasso is an algorithm for finding a sparse inverse covariance matrix [39]. This inverse covariance matrix, or precision matrix, is a Gaussian graphical model where the

features are nodes and edges represent conditional dependencies [123]. In the work presented in this dissertation, the nodes are malware samples, and the edges can be thought of as ancestor/descendant relationships between the samples. Typically, graphical lasso has solved the following single view problem:

$$\max_{\Theta}\{\log(\det(\Theta)) - \mathrm{tr}(K\Theta) - ||\Theta \circ P||_1\} \tag{1.3}$$

where K is the sample covariance matrix and P provides regularization to the entries in Θ under the l_1 norm. In Chapter 6, I show how to solve the follow multiview glasso problem:

$$\min_{\Theta,\beta}\left\{\sum_{i=1}^{M}\beta_i\mathrm{tr}(K_i\Theta) - \log(\det(\Theta)) + ||\Theta \circ P||_1\right\} \tag{1.4}$$

In this work, I use glasso to infer a phylogenetic graph of a given set of malware samples. In this context, a phylogenetic graph is a directed acyclic graph, where the node set is the malware samples and the edges represent evolutionary relationships such as "parent of".

1.4   Organization of this Dissertation

Chapter 2 reviews the background and related work. Chapter 3 explains the data representation used to model many of views used throughout this dissertation and gives a case study of this data representation with respect to the dynamic instruction trace view. In Chapter 4, I describe the deficiencies of using single views within the context of the malware problem, and I illustrate the need to incorporate multiple views for improved malware classification. Chapter 5 demonstrates the multiview framework for the spectral clustering problem and, Chapter 6 gives the details of the multiview graphical lasso problem. Finally, in Chapter 7, I conclude and give some directions for future research.

Chapter 2

Background and Related Work

This chapter highlights the background and related work that is relevant to this dissertation. As this dissertation has both security and machine learning facets, this chapter will highlight relevant related work within both fields. Sections 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, and 2.1.6 deal with security topics. Sections 2.2.1 and 2.2.2 highlight machine learning topics.

2.1 Security Background and Related Work

2.1.1 Data Representations

The n-gram representation was one of the earliest used for malware detection, and it has had great success at detecting obfuscated and polymorphic viruses [31, 48, 61, 66, 89, 90]. Given some sequence-based data, an n-gram is a contiguous region of length n that is contained within the sequence. To construct a feature vector using n-grams, first collect all consecutive token sequences of length n and sort them based on some criteria such as most frequently occurring n-grams or n-grams with the most information gain [66]. Then initialize the feature vector of length L with the number of times each of the top-L n-grams occurs in the program.

My method is related to n-grams in the sense that I use 2-grams to condition the transition probabilities for a Markov chain. The important distinction that my work makes is to use the transition probabilities instead of the 2-grams as the base data representation and to define graph kernels on the Markov chains.

The data transformation that I use, program trace to Markov chain, is similar to the Markov n-gram approach by Shafiq et al. [95,117]. Here, Shafiq et al. perform their analysis on static files such as MP3s, executable files, and compressed ZIP files and try to find an discriminative distribution of entropy for non-infected files. They set up a first order Markov chain between the bytes in their file, and compute the entropy using:

$$R = \sum_{i=0}^{n} \pi_i H(\mathbf{x}_i) \tag{2.1}$$

where $H(\mathbf{x}_i)$ is the entropy of row i in the transition probability matrix and π is the stationary distribution. A histogram is generated from this information and normalized to obtain a distribution, which approaches the Gaussian distribution as the number of sampled entropies increases due to the central limit theorem [95]. A threshold of 5 standard deviations from the mean is used to classify files as containing malware.

Data representations using n-grams are appropriate for sequence-based data, but malware, and more generally programs, have a very rich structure that cannot always be viewed as a sequence. For instance, the control flow graph of a program visualizes the code blocks and the jumps inherent in any non-trivial program. This type of data requires a general graph data representation [25].

2.1.2 Data Views

A *view* of malware is a feature set gathered from a specific analysis technique, such as the instructions taken from the disassembled programs. Malware analysis has been an active field of research in part due to the wealth of data views that are available to researchers.

These data views can generally be broken into two categories: static views and dynamic views. Each category has its own inherent set of strengths and weaknesses and, as I show later, can be combined to arrive at a superior description of what it means to be malicious.

Static Data Views

Static data views have a number of advantages in terms of computation because there is no need to devote hardware to run the executable. For this reason, the majority of signature-based antivirus software in use today uses the static representation. Antivirus software usually bases its analysis on strings of bytes that are extracted from a repository of malicious executables. Similarly, machine learning methods have used all of the bytes in the executable as their data [65, 89, 106]. The main disadvantage to this data view is its vulnerability to obfuscation techniques such as polymorphism that randomly reorder sections of the code making analysis of the binary less reliable [67].

The disassembled data has been another popular view in the literature [15, 96]. The disassembly is produced by programs such as IDA Pro [52] or `objdump` [82]. Unfortunately, some malware authors have been able to thwart attempts to disassemble their code by including encrypted packers [113], limiting the static trace to that of the unpacking code.

When the code can be disassembled, control flow graphs (CFG) can be constructed and these CFGs have been heavily studied in the literature [21, 25, 41, 67]. CFGs model the disassembled program as a series of basic blocks with edges that correspond to jumps in the control flow of the program (by returns, branches, etc.).

Portable Executable (PE) header information has also been used for static malware analysis [94]. The PE header contains a variety of information including the import address table, which provides the names of external libraries that the program references. Finally, statistics on the raw binary, such as size and entropy [75], have been used as static data views.

Dynamic Data Views

Dynamic data views have proved to be an invaluable resource to analysts trying to study encrypted and/or packed instances of malware. Some of the earliest work focused on collecting system calls from the monitored process [48]. In addition to system calls, tools have been developed that support the collection of assembly instructions that have been executed [35, 73].

Memory taint analysis is the process of checking that values such as jump addresses, registers, or code supplied input is generated by the correct source and not by a malicious program [29]. The BitBlaze framework [99] has components to monitor programs to determine if the integrity of its memory is maintained. Similar to static views, these dynamic views also have drawbacks. Malware authors have recently begun implementing code to check if their program is being monitored, and if it is, the program will either start executing benign instructions or move to a dormant state. These safeguards severely reduce the effectiveness of dynamic views [64]. Another disadvantage of dynamic data views are that they require significantly more resources in order to collect the data.

2.1.3   Malware Classification

The best known malware classification schemes are signature-based antivirus programs. To test whether a program is malicious or not, the antivirus program compares strings of bytes in its signature database to those found in the tested program, either by exact methods or approximate string matching [107]. This method is fast and has a low number of false positives, features that appeal to users. The major drawback to this approach is that it does not generalize well to malware instances that do not have a signature in its database. And given that malware authors have access to the antivirus programs, they can adjust how the polymorphic engines change the code to avoid detection by fuzzy heuristics.

To meet these challenges, several researchers began to develop less strict measures to detect malicious code. The general approach taken was to adopt one of the data views mentioned in the previous section, transform the data into a more convenient form, such as n-grams, and finally, to apply a common machine learning algorithm (decision trees, support vector machines, etc.) to that data. These methods saw dramatic improvements over classic signature-based approaches in terms of classification accuracy, typically as much as 15-20%, but at the expense of an increased number of false positives [65, 89, 106]. I improve on this approach by going beyond a single data view. I combine multiple views of the data, each with their own strengths and weaknesses, into a unified framework, typically gaining an improvement of 5-10% in classification accuracy.

Another method of malware classification uses deterministic finite automata (DFA) to create sequential models of system calls [24, 54]. The states in the DFA represent the internal state of the malware at a given time; symbols are the system calls; the transition function describes the sequence of system calls malware executes; the initial state is the beginning of the analysis; and the accepting states correspond to a detection of malicious activity [54]. Although DFAs are a valuable modeling tool in the malware context, I will restrict my attention in this dissertation to kernel-based methods and not consider them further.

2.1.4 Combining Information

Not all malware classification research has depended only on a single view of the program. Menahem et al. have combined different features of static files, including byte n-grams and imported DLLs, to build a series of classifiers based on various machine learning algorithms. They then use an ensemble learning algorithm to combine the results of the individual learners. As I will demonstrate, the lack of dynamic information severely hinders this approach.

There has been some research into combining static and dynamic features for the clustering problem. In [72], Leita et al. use the Anubis framework [4] to capture different aspects

of malware such as exploits, payloads, and behavioral features to then build different clusterings. Then they explore the relationships between the different clusterings. In contrast to that work, I base my analysis on different classes of data for both the dynamic and static views. [72] uses 17 static features for the exploit/payload/malware clusters, whereas I compare the Markov chain graphs based on the binary and disassembled information, control flow graphs, and other file information statistics. The behavioral features of Anubis focus on modifications to the Windows registry and file system, or interactions with other processes. My dynamic analysis is based on the Markov chain graphs derived from the dynamic instructions and system calls performed. It is important to note that the framework I propose is general enough to include the data views of [72] once an appropriate kernel is defined. Finally, I combine the information of the data views using a multiple kernel learning framework to build a single kernel describing the data, whereas the individual clusterings found in [72] are based solely on their respective data views.

2.1.5 Malware Clustering

There has been a rising interest in learning how to cluster malware so that researchers can gain insight into the current threat landscape [11, 56, 70, 91]. Because the majority of new viruses are derived from, or are composites of, established viruses, this information would enable faster responses and allow researchers to understand the new virus much more quickly. Much of the work in this field has again been based on a single data view and has used k-means or hierarchical clustering algorithms. In [11], Bayer et al. use behavioral characteristics of a program, such as manipulated registry keys or files, to build a feature set and then use an agglomerative hierarchical clustering algorithm to produce the clustering. Karim et al. [56] use n-perms (n-grams without an order restriction) on the binary program with an agglomerative hierarchical clustering algorithm. My work differs from the previous approaches by incorporating multiple data views to learn the clusterings, and using the Markov chain data representation.

2.1.6 Malware Phylogenetics

A *phylogeny* is a graph or tree that demonstrates evolutionary relationships within a population. Inferring accurate phylogenies for malware is beginning to be addressed in the research literature [33, 45, 46, 53, 116]. The ability to understand the lineage of a malware family in an automated way has several key benefits, the main being that it allows analysts respond quickly by leveraging information gained from having previously reverse engineered samples which are similar to the new sample. For example, when a malware analyst receives a new malware instance, she can save an enormous amount of time in dissecting and identifying it if a phyolgentic analysis can tell her that its a variant of Bagle [109]. Another important benefit for the security community is *attribution*: the ability to attribute the creation of malware to a known entity.

Within the malware literature, most work has centered around creating phylogenetic trees. For instance, in [116], Wagener et al. create a similarity matrix based on the system calls executed by the samples, and then use an agglomerative hierarchical clustering method to find the phylogenetic tree. Graph-pruning techniques have also been used [45] to find a tree. Gupta et al. begin with a fully connected similarity graph and incoming edges that are below a certain threshold are pruned for each node. All the remaining incoming edges are then pruned if their combined weight is less than a predefined threshold. In [45], malware similarity is based on text-based features gathered from McAfee's threat library database.

2.2 Machine Learning Background and Related Work

2.2.1 Multiview Clustering

Multiple kernel learning (MKL) has had a long history with many successful results [9, 62, 101]. The common theme of the majority of the work is that it is based on a modified

support vector machine with a classification-based objective function. My work deviates from this trend by finding a linear combination of kernels that explicitly optimizes a spectral clustering-based objective function. I have empirically shown that the weights resulting from this framework lead to better clustering results on several benchmark datasets as well as on a real-world malware dataset compared to state-of-the-art methods [68].

Both the pair-wise and centroid co-regularization schemes [68] I compare against have similar goals to my method: given multiple data views, find a feature space embedding that can be used in a clustering algorithm. My method differs in that it does not require a priori information about which data views are more informative. The pair-wise method finds multiple embeddings, and although each distinct embedding found contains information from the other views, one must choose which embedding to base the clustering on. The centroid method is more similar to my method as it finds a single embedding, but it relies on an external weight vector whereas my algorithm explicitly solves for the weight vector describing the importance of each view.

In [126], Zhou and Burges treat each view as a graph and define a random walk with the graph. They also extend the normalized cut problem of spectral clustering [74] to include multiple views. They combine the graphs by modifying the random walk to have a small probability of jumping to a different view. If the stationary probability mass of a subset of vertices (common among all views) is high, while leaving that subset in any given view is low, then this subset of vertices is considered a cluster. In their work they used α to denote the probability of jumping to a different view. This parameter is set before the algorithm runs by using a uniform distribution. Assuming that some views are more informative than others, one would want to have the majority of their random walk occur in those views, and therefore not having a uniform α. α is related to the weight vector I use, except that my model finds the importance of the different views with respect to a spectral clustering objective function.

2.2.2   Multiview Graphical Lasso

While there has not been any work done on a multiview graphical lasso that finds a single precision matrix, there has been some work that uses transfer learning [32, 44]. In the problem posed by Danaher et al., the goal is to find several Gaussian graphical models where the underlying data is drawn from related distributions. The canonical example for this class of methods is learning gene regulation networks for cancer and normal tissue. Both types of tissue share many edges in the network so one would want to leverage all the data possible, but these networks also have significant differences. This goal can be accomplished by adding an additional L_1 penalty to penalize the difference between the two learned Gaussian graphical models.

The main difference between the fused graphical lasso and my proposed approach is that I will find a single Gaussian graphical model conditioned on multiple views of the data. The fused graphical lasso will find multiple Gaussian graphical models, where a hyperparameter controls the amount of transfer learning between the different models.

Chapter 3

The Markov Chain Data Representation

Many current antivirus programs rely on signatures to classify programs as either malicious or benign. Signature-based approaches are popular due to the belief that they have a low false positive rate and low computational complexity on the end host, both of which are appealing for daily usage. Unfortunately, these schemes are easily defeated by simple code obfuscation techniques and 0-day attacks [25]. With the ease of code obfuscation and polymorphic viruses becoming more prevalent, other methods are becoming more attractive.

To combat these issues, several researchers began to look at more robust methods to detect malicious code. These methods generally revolve around n-gram analysis of the static binary or dynamic trace of the malicious program [31, 89, 90, 105]. These methods have shown great promise in detecting zero-day malware, but there are drawbacks. The two parameters generally associated with n-gram models are n, the length of the subsequences being analyzed, and L, the number of n-grams to analyze. For larger values of n and L, there is a much more expressive feature space that should be able to discriminate between malware and benign software more precisely. But with these larger values of n and L, the

curse of dimensionality is faced: the feature space becomes too large and there is not enough data to properly learn the model parameters. With smaller values of n and L, the feature space becomes too small and discriminatory power is lost.

The data representation developed in this chapter avoids the need to specify n and L. Instead, I model the data as a Markov chain represented by a weighted, directed graph. The instructions of the program are represented as vertices, and the weights of the edges are the transition probabilities of the Markov chain, which are estimated using a given data view.

The novel contribution is constructing a similarity, or kernel, matrix between the Markov chain graphs and using this matrix to perform classification. I use two distinct measures of similarity to construct the kernel matrix: a local measure comparing corresponding edges in each graph and a global measure which compares aspects of the graphs' topologies. This combination allows me to compare the Markov chain graphs using very different criteria, both local and global in nature, in a unified framework. Once the kernel matrix is constructed, I use support vector machines to perform the classification.

This chapter shows that the Markov chain data representation outperforms n-gram and signature-based methods on the two-class classification problem of malware versus benign programs. To examine this problem, I use a dataset with 1615 samples of malware and 615 samples of benign software. I also show that the algorithm can correctly discriminate between instances of the Netbull virus and other families of viruses. For these results, I use 13 samples of the Netbull virus, each with a different packer, and a random subsample consisting of 97 samples of unrelated malware. This result helps to validate the use of this data representation in a malware clustering/phylogenetic setting, as it shows the power of this method in classifying different examples of viruses.

mov	esi, 0x0040C000
lea	edi, [esi-0xB000]
push	edi
or	ebp, 0xFF
or	ebp, 0xFF
jmp	0x00419532
mov	ebx, [esi]
mov	ebx, [esi]
sub	esi, 0xFC
adc	ebx, ebx
jc	0x00419528
...	...
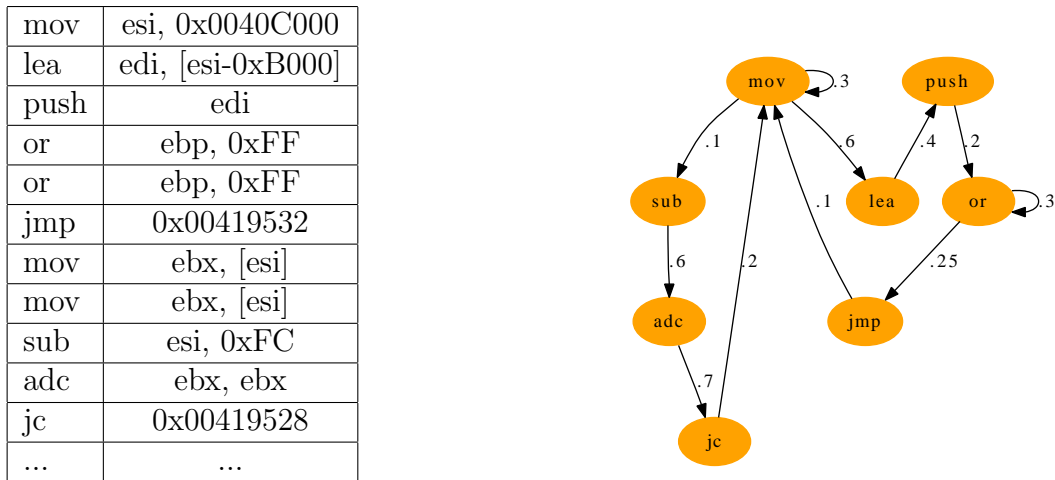
Figure 3.1: The left table shows an example of the trace data collected. A hypothetical resulting graph representing a fragment of the Markov chain is shown on the right. In a real Markov chain graph, the probabilities on all of the out-going edges would sum to 1.

3.1   Markov Chains for Malware

As an illustrative example, I focus on the dynamic trace data (collected according to Section 3.4.2), although this representation is suitable for any sequence-based data view. The dynamic trace data are the instructions the program executes, typically in a virtual machine to reduce the risk of contamination. Given an instruction trace $\mathcal{P}$, I am interested in finding a new representation, $\mathcal{P}'$, such that I can make unified comparisons in graph space while still capturing the sequential nature of the data. I achieved this by transforming the dynamic trace data into a Markov chain which I represent as a weighted, directed graph. A graph, $G = \langle V, E \rangle$, is composed of two sets, V and E. The elements of V are called vertices and the elements of E are called edges. In this representation, the edge weight, e_{ij}, between vertices i and j corresponds to the transition probability from state i to state j in the Markov chain, hence, I require the edge weights for edges originating at v_i to sum to 1, $\sum_{i \leadsto j} e_{ij} = 1$. I use an $n \times n$ ($n = |V|$) adjacency matrix to represent the graph, where each entry in the matrix, $a_{ij} = e_{ij}$ [63].

The nodes of the graph are the instructions the program executes. To find the edges of the graph, I first scan the instruction trace, keeping counts for each pair of successive instructions. After filling in the adjacency matrix with these values, I normalize the matrix such that all of the non-zero rows sum to one. This process of estimating the transition probabilities ensures a well-formed Markov chain. Figure 3.1 shows a snippet of trace data with a resulting fragment of a hypothetical instruction trace graph. The Markov chain graph can be summarized as $G = \langle V, E \rangle$, where

- V is the vertex set composed of unique instructions,

- E is the weighted edge set where the weights correspond to the transition probabilities and are estimated from the data.

The constructed graphs approximate the pathways of execution of the program, and by using graph kernels (Section 3.2), the local and global structure of these pathways can be exploited. Also, unlike n-gram methods where one must choose the top-L n-grams to use, doing the comparisons in graph space allows my methods to use more of the information contained in the instruction trace.

I experimented with another method for creating the instruction trace graphs, using a more expressive vertex set. In this method, I did not discard the arguments to the instructions but rather constructed vertices in the form $\langle operator, operand, operand \rangle$ where the operator is the instruction, and the operands are either null, or one of three types: register, memory, or dereference. This resulted in graphs with vertex sets of roughly 3,000 instructions. I did not use this representation due to poor initial performance with respect to accuracy and speed. I suspect this performance is a result of there not being enough trace data to accurately estimate the transition probabilities, similar to using a second-order Markov chain.

3.2 Constructing the Similarity Matrix

I use graph kernels [57] to compare the instruction trace graphs. A kernel, $K(\mathbf{x}, \mathbf{x}')$, is a generalized inner product and can be thought of as a measure of similarity between two objects [92]. The power of kernels lies in their ability to compute the inner product between two objects in a possibly much higher dimensional feature space, without explicitly constructing the feature space. A kernel, $K : X \times X \to \mathbb{R}$, is defined as:

$$K(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle \tag{3.1}$$

where $\langle \cdot, \cdot \rangle$ is the dot product and $\phi(\cdot)$ is the projection of the input object into feature space. A well-defined kernel must satisfy two properties: it must be symmetric (for all $\mathbf{x}$ and $\mathbf{y} \in X$: $K(\mathbf{x}, \mathbf{y}) = K(\mathbf{y}, \mathbf{x})$) and positive-semidefinite (for any $x_1, \ldots, x_n \in X$ and $\mathbf{c} \in \mathbb{R}^n$: $\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0$). Kernels are appealing in a classification setting due to the *kernel trick* [92], which replaces inner products with kernel evaluations. The kernel trick uses the kernel function to perform a non-linear projection of the data into a higher dimensional space, where linear classification in this higher dimensional space is equivalent to non-linear classification in the original input space.

My approach makes use of two types of kernels: a Gaussian kernel and a spectral kernel. The notions of similarity that these two kernels capture are quite distinct, and I found that they complement each other very well. The Gaussian kernel is defined by:

$$K_G(\mathbf{x}, \mathbf{x}') = \sigma^2 e^{-\frac{1}{2\lambda^2} \sum_{i,j}(x_{ij} - x'_{ij})^2} \tag{3.2}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are the weighted adjacency matrices of the Markov chains with the weights being the transition probabilities, σ and λ are the hyperparameters of the kernel function (determined through cross-validation), and $\sum_{i,j}$ sums the squared distance between corresponding edges in the weighted adjacency matrices. This kernel searches for local similarities between the adjacency matrices. The motivation behind this kernel is that two different classes of programs should have different pathways of execution, which would result in a low similarity score.

(a) Benign Software   (b) Malware
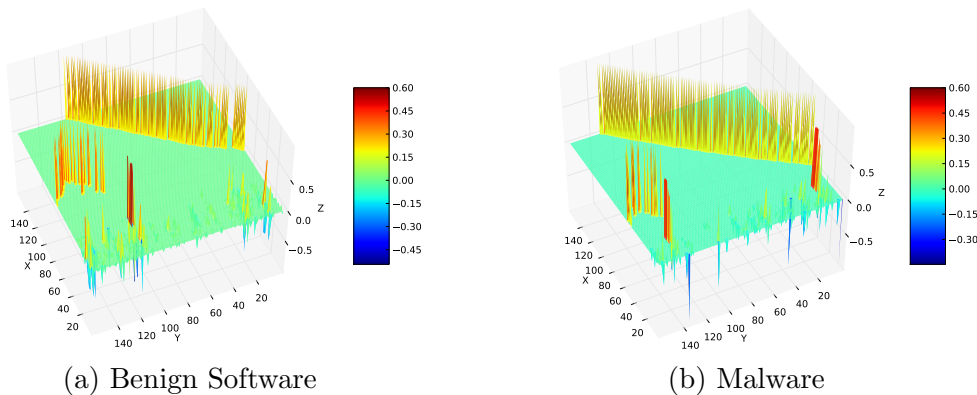
Figure 3.2: The eigenstructure of the Markov chain graph from two program traces. An example of benign software is shown in (a) and an example of malware is shown in (b).

I also use a kernel based on spectral techniques [26]. These methods use the eigenvectors of the graph Laplacian to infer global properties about the graph. The weighted graph Laplacian is a $|V| \times |V|$ matrix defined as:

$$\mathcal{L} = \begin{cases} 1 - \frac{e_{vv}}{d_v} & \text{if } u = v, \text{ and } d_v \neq 0, \\ -\frac{e_{uv}}{\sqrt{d_u d_v}} & \text{if } u \text{ and } v \text{ are adjacent,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

where e_{uv} is the weight between vertices u and v, and d_v is the degree of v. I take the eigenvectors associated with non-zero eigenvalues of $\mathcal{L}$, $\phi(\mathcal{L})$, as the new set of features. These eigenvectors encode global information about the graph's smoothness, diameter, number of components and stationary distribution among other things. With this information, the second kernel is constructed by using a Gaussian kernel on the eigenvectors:

$$K_S(\mathbf{x}, \mathbf{x}') = \sigma^2 e^{-\frac{1}{2\lambda^2} \sum_k ||\phi_k(\mathcal{L}(\mathbf{x})) - \phi_k(\mathcal{L}(\mathbf{x}'))||^2} \tag{3.4}$$

where $\phi_k(\mathcal{L}(\mathbf{x}))$ and $\phi_k(\mathcal{L}(\mathbf{x}'))$ are the eigenvectors associated with the weighted Laplacian of the adjacency matrices, $\mathcal{L}(\mathbf{x})$ and $\mathcal{L}(\mathbf{x}')$.

To give some intuition behind the spectral kernel, Figure 3.2 plots the eigenvectors of

the graph Laplacian for an example of benign software and an example of malware. The diagonal ridge in each figure represents all of the unused instructions in the trace, which are disconnected components in the graph. To construct K_S, only the top-k eigenvectors are used and this ridge information is discarded. The decision to use only the top-k eigenvectors is defended in Section 3.4.4. The interesting information of the graph, the actual program flow contained in the largest connected component, is found in the spikes and valleys at the bottom of Figures 3.2a and 3.2b. The eigenvectors of the Laplacian can be thought of as a Fourier basis for the graph [26]. Comparing these harmonic oscillations, encoded by the eigenvectors, between different types of software provides discrimination between structural features of the graph such as strongly connected components and cycles.

3.2.1 Algebra on Kernels

If two kernels, K_1 and K_2, are valid, then $K = K_1 + K_2$ is also a valid kernel [16]. This algebra on kernels allows for the elegant combination of kernels that measure different aspects of the input data, and it is the object of study in multiple kernel learning [9, 101]. The final kernel is a weighted combination of K_G and K_S:

$$K_C = \mu K_G + (1 - \mu)K_S \tag{3.5}$$

where $0 \leq \mu \leq 1$. A convex combination of kernels, where $\mu_i \geq 0$ and $\sum_i \mu_i = 1$ for some weight vector μ, also forms a valid kernel. Within the context of this chapter, μ is found using a cross-validation search where the candidate μ's are restricted to be in the range [.05, .95] with a step size of .05. Although more advanced techniques to search for the parameters of multiple kernel learning exist [101], I found this simple approach to be sufficient for the combination of these two kernels. With more than two kernels, grid search becomes significantly more complex making other alternatives more efficient.

3.3 Classifying Malware

I use support vector machines [22] to perform the classification. Support vector machines search for a hyperplane in the feature space that separates the points of the two classes with a maximal margin [22]. The hyperplane that is found by the SVM is a linear combination of the data instances, x_i, with weights, α_i. It is important to note that only points close to the hyperplane will have non-zero α's. These points are called support vectors. Therefore, the goal in learning SVMs is to find the weight vector, α, describing each data instance's contribution to the hyperplane. Using quadratic programming, the following optimization problem can be efficiently solved:

$$\max_{\alpha} \left(\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \right) \tag{3.6}$$

subject to the constraints:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.7}$$

$$0 \leq \alpha_i \leq C \tag{3.8}$$

In Equation 3.6, y_i is the class label of instance x_i, and $\langle \cdot, \cdot \rangle$ is the Euclidean dot product. Equation 3.7 constrains the hyperplane to go through the origin. Equation 3.8 constrains the α's to be non-negative and less than some constant C. C allows for soft-margins, meaning that some of the examples may fall between the margins. This helps prevent over-fitting the training data and allows for better generalization accuracy. The weight vector for the hyperplane is then defined to be:

$$\mathbf{w} = \sum_{i} \alpha_i y_i \mathbf{x}_i \tag{3.9}$$

With this current setup, only linear hyperplanes are possible in the d-dimensional space defined by the feature vectors of $\mathbf{x}$. By using the *kernel trick*, the data instances can be projected into a higher dimensional space and a linear hyperplane can be found in that

space, which would be equivalent to a non-linear hyperplane in the original d-dimensional space. The new optimization problem is:

$$\max_{\alpha} \left(\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \right) \tag{3.10}$$

Equation 3.6 and Equation 3.10 are identical with the exception that the dot product, $\langle \cdot, \cdot \rangle$, has been replaced with the kernel function $k(\cdot, \cdot)$.

Given α found in Equation 3.10, the decision function is defined as:

$$f(\mathbf{x}) = \text{sgn} \left(\sum_{i}^{n} \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i) \right) \tag{3.11}$$

which returns class +1 if the summation is ≥ 0, and class -1 if the summation is < 0. The number of kernel computations in Equation 3.11 is decreased because many of the α's are zero.

3.4 Experimental Setup

3.4.1 Data/Environment

To perform the experiments, I used a machine with quad Xeon X5570s running at 2.93GHz with 24GB of memory. The dataset is composed of two distinct groups which is used to answer two main questions:

1. What is the false and true positive performance when classifying benign versus malware samples?

2. Can the proposed method discriminate the Netbull virus, with different packers, from other malware samples?

To answer the first question, I collected 1615 instances of malware and 615 instances of benign software, as described in Section 3.4.2. To answer the second question, I used 13 instances of the Netbull virus with different packers, such as UPX [113] and ASprotect [5], and compared against a random subsample of 97 instances of malware.

3.4.2   Data Collection

The data collection technique uses the Ether analysis framework [35] to extract data from a Windows XP system. The Ether system was chosen as it guarantees some level of protection against hardware based virtual machine detection. The main protection mechanisms that need to be overcome are debugger and virtual machine detection, timing attacks, and host system modifications. Each of these violate the fundamental tenet that the analyzed system must not be altered in any manner. The end result of instruction tracing is a list of the in-order executed instructions. These instructions provide the input to the algorithm.

To analyze the data, I begin by copying the executable to the Ether analysis system. Then an instantiation of a Windows virtual machine is started, and after successful boot, the file is copied onto the virtual machine. Then, the Ether portion of Xen is invoked and the malware is started. The sample is allowed to run for five minutes, which has been shown to be sufficient [87]. It should be noted that the 5-minute heuristic does not take into account execution-stalling malware. With as much as 33% of malware exhibiting execution-stalling behavior [64], dynamic analysis by itself is not sufficient. This will be addressed in Chapter 4.

I found 160 unique instructions across all of the dynamic traces collected. These instructions are the vertices of the Markov chains. The representation ignores operands used with the 160 instructions. By ignoring operands, I remove sensitivity to register allocation and other compiler artifacts. It is important to note that rarely did the instruction traces make use of all 160 unique instructions, and therefore, the adjacency matrices of the instruction

trace graphs contain some rows of zeros. The decision to incorporate unused instructions in the model allowed for a consistent vertex set between all instruction trace graphs, and thus allowing for uniform comparisons in graph space.

3.4.3 Other Methods

To determine the validity of the Markov chain data representation, I compared it to a standard n-gram model [66] and 9 leading antivirus software programs that use signature-based methods. For the standard n-gram model, I chose the top-L n-grams to use by computing the information gain as suggested in [66]:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{y_i \in Y} \hat{P}(v_j, y_i) \log \frac{\hat{P}(v_j, y_i)}{\hat{P}(v_j)\hat{P}(y_i)} \tag{3.12}$$

where v_j represents whether the jth n-gram exists or not, $\hat{P}(v_j, y_i)$ is the percentage of data instances with class label y_i and value v_j, $\hat{P}(v_j)$ is the percentage of instances with value v_j, and $\hat{P}(y_i)$ is the percentage of instances with class label y_i. The intuition with the information gain criteria is that one should choose n-grams that are either highly correlated with the malware class or the benign class, but not both. Once the list of n-grams is sorted based on their information gain, the top-L are selected to train the classification algorithm. It is important to note that I used n-grams derived from the same dynamic traces on which the Markov chain representations were based and not on static information contained in the binary.

To find the best choices for the parameters n and L for the standard n-gram model, I varied n from 2 to 6 and L from 500 to 3,000 in increments of 500. For each choice of parameters, I ran both a support vector machine, with a linear kernel, Gaussian kernel, and d-order polynomial ($2 \leq d \leq 9$) kernel, and a k-nearest neighbor classifier ($1 \leq k \leq 9$), with the feature vector consisting of the top-L n-grams. I present the top-5 performing parameter combinations for the n-gram model.
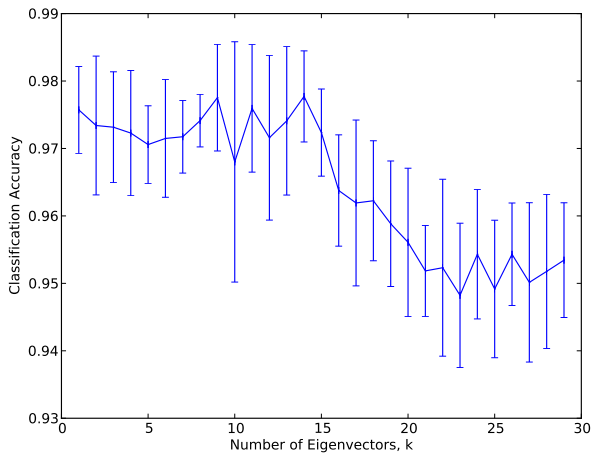
Figure 3.3: Classification accuracy of 50 instances of malware versus 10 instances of benign software as the number of eigenvectors are varied, k, of the spectral kernel. Results are averaged over 10 runs with the error bars being one standard deviation.

I also present the top-5 performing antivirus programs in the results. These are BitDefender [18], Kaspersky [58], Avira [8], F-Secure [40], and F-prot [37]. These results were gathered in September, 2010 with the currently up-to-date signature databases.

3.4.4 Selecting the k eigenvectors

To find the appropriate k, the number of eigenvectors used to classify the program traces in Equation 3.4, I performed a series of tests on an independent dataset of 50 malware program traces and 10 benign traces where I adjusted k using values ranging from 1 to 30. To reduce computation, I choose the smallest possible k that still maintains discriminatory power relative to classification accuracy.

Figure 3.3 shows the results of choosing k averaged over 10 runs with the error bars showing one standard deviation. The decreasing performance as k is increased is expected because as more eigenvectors are chosen, the feature space begins to overfit the training

Method	Accuracy (%)	FPs	FNs	AUC
Gaussian Kernel	95.70%	44	52	.9845
Spectral Kernel	90.99%	80	121	.9524
Combined Kernel	**96.41**%	47	**33**	**.9874**
n-gram (n=3, L=2500, SVM=3-poly)	82.15%	300	98	.9212
n-gram (n=4, L=2000, SVM=3-poly)	81.17%	327	93	.9018
n-gram (n=2, L=1000, 4-NN)	80.63%	325	107	.8922
n-gram (n=2, L=1500, SVM=2-poly)	79.82%	339	111	.8889
n-gram (n=4, L=1500, SVM=Gauss)	79.42%	354	105	.8991
BitDefender	73.32%	**0**	595	N/A
Kaspersky	53.86%	1	1028	N/A
Avira	49.60%	**0**	1196	N/A
F-Secure	43.27%	1	1264	N/A
F-Prot	42.96%	1	1271	N/A

Table 3.1: The classification accuracy of 615 instances of benign software versus 1615 instances of malware. Best performing method is bolded. The top 5 parameter choices for the n-gram model are presented as well as the top-5 performing signature-based antivirus programs. FP = false positive, FN = false negative, AUC = area under the ROC curve, Accuracy = TP+TN/(# of samples)

data, which results in lower accuracy on the test set. Given these results, I set $k = 9$ to maximize classification accuracy for the next experiments. Alternatively, k could be selected using cross-validation on a validation dataset for each experiment.

3.5 Results

3.5.1 Benign versus Malware

I now explore the validity of the multiple kernel learning method as an alternative to n-gram and signature-based virus detection methods. Table 3.1 presents the results of the three different kernels and the n-gram methods using 10-fold cross-validation. The top-5 performing antivirus programs are also presented. For the n-gram methods, I used the

(a) Gaussian Kernel   (b) Spectral Kernel   (c) Combined Kernel
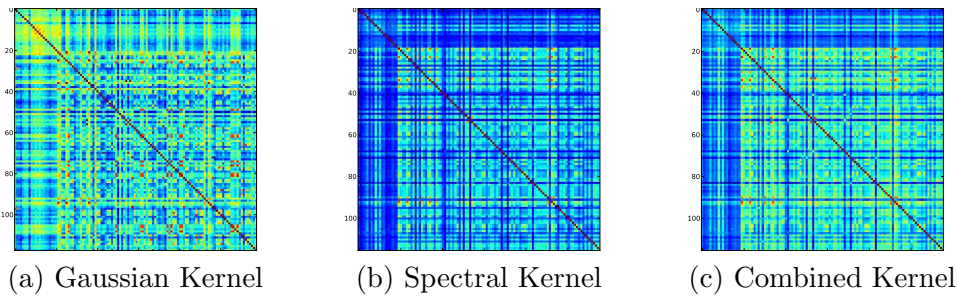
Figure 3.4: The heat maps of the kernel (similarity) matrix for benign software versus malware. Benign programs are the first 19 instances in the top left. Warmer colors represent higher measures of similarity. The smaller block in the upper left of each figure is the benign software and the larger lower right is the malware. The axes are the indices of the samples.

same parameters as discussed in the previous section. The best results with respect to overall accuracy for the n-grams were achieved when $n = 4$, $L = 1000$ and a support vector machine with a second order polynomial kernel was used. It is interesting that three antivirus programs, as well as the methods based on the Markov chain data representation, labeled the same benign executable as being malicious. This could be a result of a noisy dataset or a bad signature.

Both machine learning approaches, graph kernels and n-grams, were able to easily outperform the standard antivirus programs with respect to overall accuracy. However, it should be noted that antivirus programs operate under the assumption of a "wild list" [28], meaning that the signature database will only contain signatures that the antivirus company believes to be currently propagating in the wild. The malware that was tested was collected over a 6 month period 6 months prior to the results presented in Table 3.1. It is very possible that these antivirus programs had signatures of a larger percentage of the data, and then pruned these signatures as they were no longer circulating in the wild.

Despite n-grams outperforming the antivirus programs, the Markov chain data representation still provided a significant improvement over n-gram methods. These results reinforce the hypothesis that learning with the Markov chain graphs improves accuracy. Table 3.1 also

Method	Accuracy (%)	FPs	FNs	AUC
Gaussian Kernel	**99.09**%	**1**	**0**	**.9965**
Spectral Kernel	96.36%	4	0	.9344
Combined Kernel	**100.00**%	**0**	**0**	**1.00**
n-gram (n=4, L=1000, SVM=2-poly)	94.55%	5	1	.8776
n-gram (n=4, L=2500, SVM=Gauss)	93.64%	6	1	.8215
n-gram (n=6, L=2500, SVM=2-poly)	92.73%	6	2	.8432
n-gram (n=3, L=1000, SVM=2-poly)	89.09%	12	0	.6173
n-gram (n=2, L=500, 3-NN)	88.18%	12	1	.6334

Table 3.2: The classification accuracy of 13 instances of the Netbull virus with different packers versus 97 instances of malware. Best performing method is bolded.

(a) Gaussian Kernel (b) Spectral Kernel (c) Combined Kernel
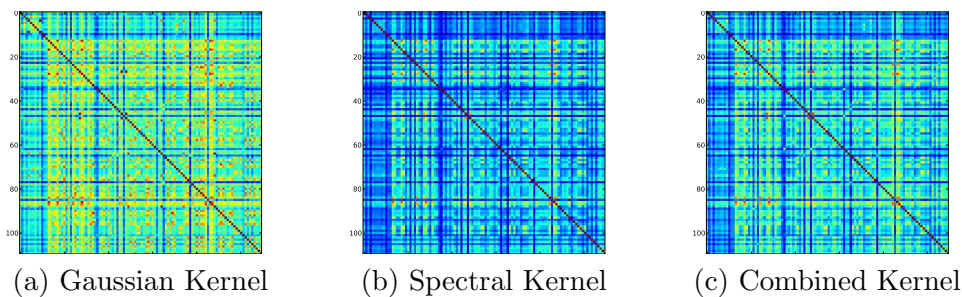
Figure 3.5: The heat maps of the kernel matrix for the Netbull virus with different packers versus malware. Netbull are the first 13 instances in the top left.

illustrates that a combined kernel, which uses the local and global structural information about the Markov chain graph as discussed in Section 3.2, improves performance over the standalone kernels.

Figure 3.4 shows the heat maps (the values for the similarity matrix) for the 3 kernels I tested against. For visual purposes, I only show kernel values for 19 benign samples and 97 malware samples. The program traces that are more similar will have warmer colors. The block structure seen in this figure indicates that these kernels are able to discriminate between the two classes of software.

With the current dataset, I have more examples of malware than I do benign software. This is a by-product of the availability of the benign trace data. This data skew might contribute to the false-positives found in both my method and the *n*-gram methods. In a production setting, a more diverse and extensive set of benign trace data would be needed to alleviate this problem.

3.5.2   Netbull versus Malware

The second set of experiments evaluates the performance of these algorithms with respect to their ability to differentiate between different types of malware. This is an important question as I will use this methodology in a clustering/phylogenetics setting in later chapters. This dataset was composed of 13 instances of the Netbull virus with different packers and a random subsample of 97 instances of malicious code from the main malware dataset. I limited the number of other families of viruses to 97 due to the effects of data skew. The results are summarized in Table 3.2.

These results are promising because the method using the combined kernel can correctly classify all instances of the Netbull virus despite this being a very skewed dataset. The *n*-gram methods were less successful on this task. After the top-3 parameter choices for the *n*-grams, these models quickly devolved into predicting the majority class for all instances. This is a known problem in machine learning algorithms [23].

The kernels for this dataset are displayed in Figure 3.5 and have a similar block structure to Figure 3.4. This is important because it validates the approach's ability to distinguish between somewhat similar pieces of malware. These results also validate my choice of data representation and associated kernels in a kernel-based clustering environment [74].

Component	Time
Gaussian Kernel	147.91 ± 9.54
Spectral Kernel	550.55 ± 32.90
SVM Optimization	0.16 ± 0.05
Classifying New Instance	0.54 ± 0.07
Total Offline	698.45 ± 57.44
Total Online	0.54 ± 0.07

Table 3.3: Timing results for the computation time for each step of the method. All results are in seconds with one standard deviation given.

3.5.3   Timing Results

In this section I explore the computation time for the proposed method. As stated previously, there are two main components to this approach; computing the graph kernels and performing the support vector machine optimization (Equations 3.5 and 3.10), which can be done offline, and the classification of a new instance (Equation 3.11), which is done online. I used the dataset composed of 1,615 samples of malicious programs and 615 samples of benign programs.

As Table 3.3 illustrates, the majority of the method's time is spent computing the kernel matrices. It took 698.45 seconds to compute the full kernel matrices. However, this computation can be performed offline once, so it will not slow down a production system. The online component of classifying a new instance took 0.54 seconds as shown in Table 3.3. The majority of this time is spent computing the kernel values between the new instance and the labeled training data as described in Equation 3.11.

The number of kernel computations is decreased due to the support vector machine finding a sparse set of support vectors. The PyML implementation of the SVM typically found ∼350 support vectors. There are other forms of support vector machines [55] that search for sparser solutions, which would speed up this online component by reducing the number of support vectors, and thus the number of kernel computations. The complexity

analysis in included in Appendix C.1.

3.6   Conclusion

With the advent of polymorphic code and obfuscated viruses, signature-based malware detection is becoming outdated [25]. To combat this, many researchers have began drawing ideas from machine learning to create more flexible detection algorithms. Many of these approaches have centered around using n-gram based statistics to classify new instances as being either benign or malware.

The Markov chain data representation presented in this chapter extends the n-gram methodology by using 2-grams to condition the transition probabilities of a Markov chain, and then treats that Markov chain as a graph. Treating the Markov chain as a graph allows the use of graph kernels to construct a similarity matrix between instances in the training set. I use two distinct measures of similarity to construct the kernel matrix: a Gaussian kernel, which measures local similarity between the graphs' edges, and a spectral kernel, which measures global similarity between the graphs. Given the kernel matrix, I can then train a support vector machine to perform classification on unlabeled samples.

I demonstrated the performance of the Markov chain data representation on two problems. The first investigated whether the method could properly discriminate between instances of malware and benign software. I showed that with the combined kernel my method was able to outperform n-gram and signature-based methods, while maintaining a low false positive rate. The second problem tested whether the method could discriminate between different types of malware. I compared the Netbull virus with different packers to a set of other instances of malware. The method was able to perfectly classify these instances. This result is promising for using these kernels in a clustering setting, which would allow researchers to more quickly understand the dynamics of a new virus.

Chapter 4

Malware Classification: Bridging the Static/Dynamic Gap

While classifying malicious programs based on dynamic instruction traces has shown great promise, there are several limitations to relying on a single view of the data. As I will mention in Section 4.1.2, the collection of dynamic information can be detected by the malicious program. When a sufficiently sophisticated malicious program has detected that it is being traced or being run in a virtual machine, it will then either stall its execution or perform benign activities, causing a great deal of confusion for the reverse engineer in charge of the analysis. In this case, the reverse engineer would fall back to purely static techniques.

The novel contribution of this chapter is to show how to combine different data views using multiple kernel learning [101] to arrive at a new classification system that integrates all of the available information about a program into a unified framework [2]. The key insight of my approach is that each data view provides complementary information about the true nature of a program, but no one view contains all of this information. I aim to construct a classification system that contains all aspects of a program's true intended behavior. I begin by defining a kernel, a positive semi-definite matrix where each entry in

the matrix is a measure of similarity between a pair of instances in the dataset, for each data view. I then use multiple kernel learning [9, 101] to find the weights of each kernel, create a linear combination of the kernels, and finally use a support vector machine [22] to perform classification.

This framework is particularly appealing for three reasons. First, it combines both static and dynamic features in a way that allows the learning algorithm to take advantage of both simultaneously. Second, it is extendable in the sense that future data views could be easily added to the model without complicating the final result. Finally, the method presented in this chapter is highly parallelizable: computing the kernel values for testing new malware can be carried out in parallel, the implication being that larger datasets can easily be handled.

I present my results on a dataset composed of 776 benign programs and 780 malicious programs. In addition to this dataset, I test the methods on a separate validation dataset composed of 20,936 malicious samples. For each program I collect six data views: the static binary, the disassembled binary file, the control flow graph from the disassembled binary file, a dynamic instruction trace, a dynamic system call trace, and a file information feature vector composed of information gathered from all of the previous data views. For the binary file, disassembled file, and two dynamic traces, I build kernels based on the Markov chain graphs; for the control flow graph, I use a graphlet kernel [97]; and for the file information feature vector, I use a standard Gaussian kernel as explained in Section 4.1.2.

I show that integrating multiple data views increases overall classification performance with regard to accuracy, receiver operating characteristic (ROC) curves, and area under the ROC curve (AUC). I also provide results examining the efficacy of each individual data view with regard to different metrics, including the time from receiving the sample to making a classification decision. I report kernel combinations (in addition to the combination of all six data views) which can achieve reasonably high performance if time and computing resources are at a premium. I also demonstrate some of the pitfalls of the dynamic and static methodologies, such as, static data views struggling with packed instances. Instances whose

entropies deviated from the mean also caused problems, where in the dataset used the mean of malware was 7.52 and the mean of benign was 6.34,

4.1 The Deficiencies of Individual Views for the Malware Problem

As mentioned in the introduction to this chapter, malware authors have developed methods to obfuscate their malware. The typical story is a back-and-forth between malware authors and malware analysts: the analysts find a new way to detect malware, and the malware authors respond by creating protection mechanisms to avoid this detection. In the following sections, I give some example. The two main classes of data used by analysts are static, data that does not require the program to be executed, and dynamic, data that needs to be executed (in most cases in a virtual machine to reduce the chance of contamination). There have been many protections put in place to hinder analysis in both of these views, making either unreliable by itself.

4.1.1 Static Data Views

Encrypted malware was one of the first techniques employed by malware authors [121]. This was originally used to avoid signature detection by generating different encryption keys when the malware was copied, thus creating unique binaries at each step of the infection. Encrypted malware cannot be easily reverse engineered, making many static views unreliable. For instance, disassembling the code would result in the disassembly of the decrypting code, but not the actual malicious code.

4.1.2   Dynamic Data Views

Once reverse engineers realized that static views of the code could be easily compromised, they turned their attention to dynamic analysis. For safety reasons, during analysis malicious code is normally run in a virtual machine behind a firewall to prevent further propagation. Malware authors responded to these methods by developing execution-stalling techniques [64]. Execution-stalling prevents the malware from running for some prescribed amount of time, assuming that the reverse engineers will give up after some relatively short period of time. A slightly more advanced method of execution-stalling looks at where the malware is being run, halting if it detects it is being run from a virtual machine with the assumption being that only malware analysts would run programs within a virtual machine. This type of defense may become less common as virtual machines become more prevalent.

4.2   Integrating Multiple Views for Improved Performance

In this section, I highlight the six data views I have chosen to use in my model. It is important to note that while these views have all been well-studied in the literature $[1, 15, 48, 65, 67]$, they are by no means a complete survey of the literature. The methods reviewed in Section 4.2.2 can be easily extended to take advantage of any additional views that an analyst has at his or her disposal.

4.2.1   Data Views

I aim to cover some of the most popular data views that have been used for malware classification in the literature. These data views also try to capture many of the different views of a program in the hopes that, while a malicious executable can disguise itself in some views, dis-

guising itself in every view while maintaining malicious intent will prove to be substantially more difficult. I use three static data views: the binary file, the disassembled binary, and the control flow graph of the disassembled binary. I use two dynamic data views: the dynamic instruction trace and the dynamic system call trace. Finally, I use a file information data view, which contains seven statistics that provide a summary of the previous data views.

Binary. I use the raw byte information contained in the binary executable to construct my first data view. There is a long history of using this type of data to classify malware [65, 106]. Generally, the bytes are used in an *n*-gram framework to construct a feature vector that is then given to some machine learning classification algorithm (i.e. boosted decision trees [65]). In contrast to these methods, I use 2-grams to condition a Markov chain and then perform classification in graph space as explained in Section 4.2.2. In the Markov chain, the byte values (0-255) correspond to different vertices in the graph, and the transition probabilities are estimated by the frequencies of the 2-grams.

Disassembled. The opcodes of the disassembled program have also been used to generate malware detection schemes [15, 96]. To generate the disassembled code, I use IDA Pro [52]. Once I have the disassembled code, I build a Markov chain similar to the way the Markov chain for the binary files was built. Instead of the byte values being the vertices in the graph, I use the disassembled instructions for the vertices in the graph. Unfortunately, the number of unique instructions found in the disassembled files ($\sim$1200) resulted in very large Markov chains that overfitted the data resulting in poor initial performance. This is in part due to the *curse of dimensionality* [16]: the feature space becomes too large and there is not enough data to properly learn the model parameters. Furthermore, some instructions perform identical or very similar tasks, resulting in many transitions that are identical yet treated as distinct. To combat this, I used several categorizations, each with increasing complexity. The coarsest categorization contained eight categories (math, logic, privileged, branch, memory, stack, nop, and other). The other categorizations had 34, 68, 77, 86, 154,
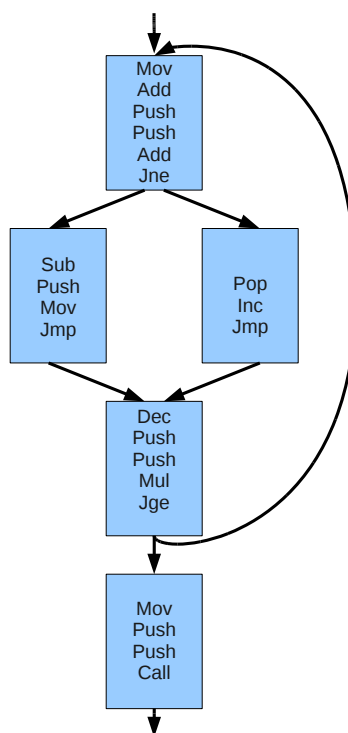
Figure 4.1: An example of a control flow graph demonstrating jumps.

and 172 categories. I found the categorization with 86 categories to perform the best. This categorization had separate categories for most of the initial 8086/8088 instructions as well as categories for some extended instruction sets such as SSE and MMX. Further research into an optimal categorization that better represents program behavior is currently being considered and is discussed in Section 7.2.

Control Flow Graph. The use of control flow graphs has become a popular means to perform malware classification [25, 67]. A control flow graph is a graph representation that models all of the paths of execution that a program might take during its lifetime (Figure 4.1). In the graph, the vertices are the basic blocks, sequential code without branches or jump targets, of the program, and the edges represent the jumps in control flow of the

program. One of the advantages of this representation is that it has been shown to be very difficult for a polymorphic virus to create a semantically similar version of itself while modifying its control flow graph enough to avoid detection [67]. To compute the similarity between different control flow graphs, I use a simplified kernel based on previous work in the literature [67]. This kernel works by counting similarly shaped subgraphs of a certain size, and is explained in detail in Section 4.2.2.

Dynamic Instruction Traces. Due to the limits of static analysis discussed in Section 4.1.1, I chose to include two dynamic data views, the instruction traces and the system call traces collected over a five minute run using the KVM virtual machine [69] and the Intel Pin program [73]. Dynamic instructions traces are known to produce highly accurate malware classification results [1,31]. Similar to the disassembled data, I used the same categorization of 86 categories and constructed the Markov chain in the same manner.

Dynamic System Call Traces. System call traces have been another popular dynamic data view [12, 48, 93]. Over the 1556 traces, I recorded 2460 unique system calls. I used the Markov chain graph representation, and like the disassembled instruction set, found that treating each unique system call as a vertex in the Markov chain led to poor initial performance. I therefore grouped the system calls into 94 categories where each category represents semantically similar groups of system calls, such as painting to the screen, writing to files, or cryptographic functions.

Miscellaneous File Information. For this data view, I collected seven pieces of information about the various data views described previously. This data is summarized in Table 4.1. I look at the entropy and the size of the binary file. Similar to previous work on entropy [75, 95], I found the average entropy of the benign files in the dataset to be 6.34 and the average entropy of the malicious files in the dataset to be 7.52. I also have a binary feature to look at whether the binary executable has a recognizable packer such as UPX [113] or Armadillo [112]. To find whether a file was packed or not, I used the PEID signature method

Statistic	Malware	Benign
Entropy	7.52	6.34
Binary Size	0.799	2.678
Packed	47.56%	19.59%
Num Vertices (CFG)	5,829.69	10,938.85
Num Edges (CFG)	7,189.58	13,929.40
Num Static Instrs	50,982	72,845
Num Dynamic Instrs	7,814,452	2,936,335

Table 4.1: Summary of the file information statistics used: the average entropy, average size of the binary (in megabytes), average number of vertices and edges in the control flow graph, the average number of instructions in the disassembled files, and the average number of instructions/system calls in the dynamic traces. The percentage of files known to packed is also given.

[86]. For the disassembled binary feature, I took the number of instructions found in the disassembled file. I also use the number of edges and the number of vertices in the control flow graph. Finally, I took the sum of the number of dynamic instructions and dynamic system calls as the last feature.

4.2.2   Multiple Kernel Learning

In this section, I first describe how I transform the six data views of Section 4.2.1 into more convenient representations. I then show how it is possible to define kernels, or similarity measures, that are able to accurately compare these data views in their new representations. Finally, I describe a method of multiple kernel learning that finds a linear combination of these kernels which can then be used in a support vector machine setting.

Data Representations. The six canonical data views described in Section 4.2.1 can be grouped into three sets. The miscellaneous file information that I collect can be represented as a simple feature vector of length seven where each of the seven statistics corresponds to

mov	esi, 0x0040C000
lea	edi, [esi-0xB000]
push	edi
or	ebp, 0xFF
or	ebp, 0xFF
jmp	0x00419532
mov	ebx, [esi]
mov	ebx, [esi]
sub	esi, 0xFC
adc	ebx, ebx
jc	0x00419528
...	...
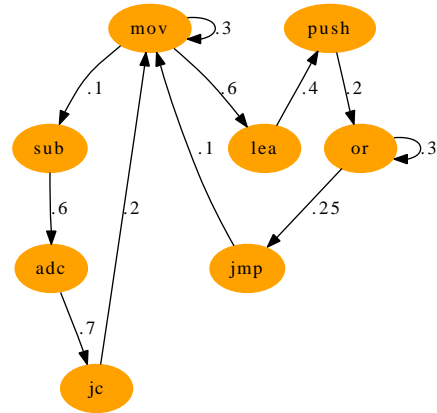
Figure 4.2: The left table shows an example of the trace data collected. A hypothetical graph representing a fragment of the Markov chain derived from the trace is shown on the right. In a real Markov chain graph, all of the out-going edges would sum to 1.

a feature. The control flow graphs are represented in the same way as the standard in the literature [25, 67], where the basic blocks are nodes in a graph and directed edges represent the control flow between different blocks. For the third set, the raw binary, disassembled binary, dynamic instruction trace, and dynamic system call trace, I use the Markov chain representation described in Chapter 3 and illustrated in Figure 4.2.

Kernels.

As described in Section 3.1, a kernel, $K(\mathbf{x}, \mathbf{x}')$, is a generalized inner product and can be thought of as a measure of similarity between two objects [92]. For the Markov chain representations and the file information feature vector, I use a standard squared exponential kernel:

$$K_{SE}(\mathbf{x}, \mathbf{x}') = \sigma^2 e^{-\frac{1}{2\lambda^2} \sum_i (\mathbf{x}_i - \mathbf{x}'_i)^2} \tag{4.1}$$

where $\mathbf{x}_i$ represents one of the seven features for the file information data view, or a transition probability for the Markov chain representations. σ and λ are the hyperparameters of the

(a) Binary SE (b) File Info SE (c) CFG 4-graphlet

(d) Dynamic Instruction SE (e) Static Instruction SE (f) System Call SE
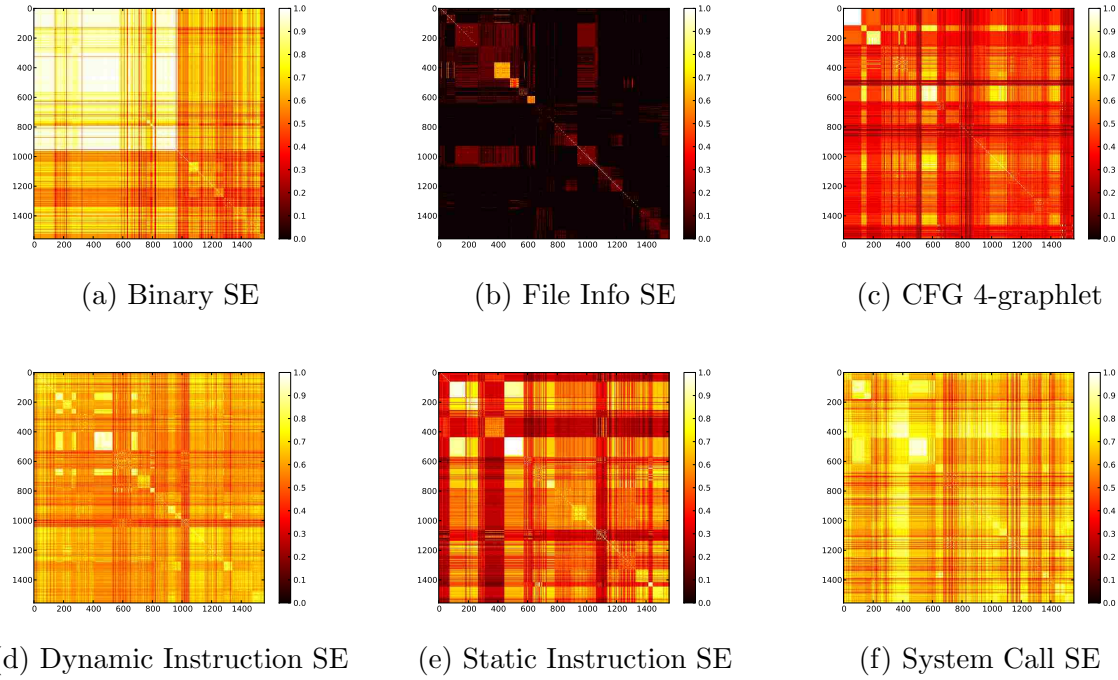
Figure 4.3: Heatmaps for the six individual kernels. The first 780 samples (the top left block in the heatmaps) are the malware samples and the second 776 samples (the bottom right block) are the benign samples. The off diagonal blocks are the similarities between malware and benign samples.

kernel function (estimated through cross-validation), and $\sum_{i,j}$ sums the squared distance between the corresponding features.

For the control flow graph data view, I attempted to find a kernel that closely matched previous work in the literature [67]. Although the approach I chose does not take the instruction information of the basic blocks into account, I selected the graphlet kernel due to its computational efficiency [97]. A k-graphlet is defined as a subgraph, of a graph G, with the number of nodes of the subgraph equal to k. If $\vec{f}_G$ is a a feature vector, where each feature is the number of times a unique graphlet of size k occurs in G, the normalized
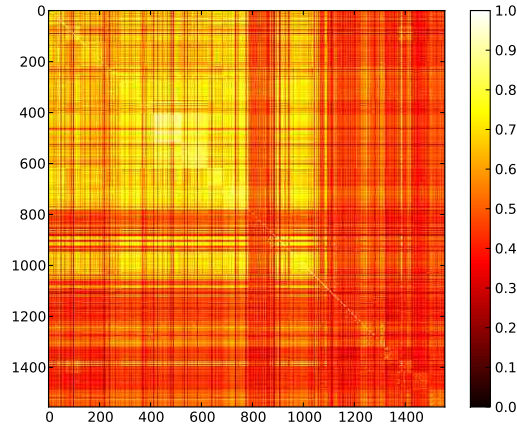
Figure 4.4: Heatmap for all six kernels combined with the weights found using multiple kernel learning.

probability vector is:

$$\vec{D}_G = \frac{\vec{f}_G}{\text{\# of all graphlets of size } k \text{ in } G} \tag{4.2}$$

and the graphlet kernel is defined as:

$$K_g(G, G') = \vec{D}_G^T \vec{D}_{G'} \tag{4.3}$$

I experimented with graphlets of size $k \in \{3, 4, 5\}$ and found $k = 4$ to be optimal with respect to both classification accuracy and AUC.

Figure 4.3 shows the heatmaps for the six individual kernels and Figure 4.4 shows the heatmap of the combined kernel with the weights of the individual kernels being found using multiple kernel learning (see Equation 4.8). The block structure observed in these figures is interesting because it shows that the kernels and data views I selected are able to discriminate between malware and benign samples. This is apparent in Figure 4.4 where the top left block (the similarity between the malware samples) has very high values compared with the rest of the image.

A set of M valid kernels, K_1, K_2, …, K_M, and some vector β where $\beta_i \geq 0 \; \forall i$ and $\sum_i \beta_i = 1$, can be combined into a valid kernel K_{comb} [16],

$$K_{comb} = \sum_{1 \leq i \leq M}^{M} \beta_i K_i \tag{4.4}$$

This algebra on kernels allows us to combine kernels that measure very different aspects of the input data, or even different views of the data, and it is studied by multiple kernel learning [9, 101].

Multiple Kernel Learning. The goal of classical kernel-based learning with support vector machines is to learn the weight vector, α, describing each data instance's contribution to the hyperplane that separates the points of the two classes with a maximal margin [22] and can be found with the following optimization problem:

$$\min_{\alpha} \underbrace{\left(\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{n} \alpha_i \right)}_{S_k(\alpha)} \tag{4.5}$$

subject to the constraints:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{4.6}$$

$$0 \leq \alpha_i \leq C$$

where y_i is the class label of instance x_i. Equation 4.6 constrains the α's to be non-negative and less than some constant C. C allows for soft-margins, meaning that some of the examples may fall between the margins. This helps prevent over-fitting the training data and allows for better generalization accuracy [22].

With multiple kernel learning, each individual kernel's contribution, β, must also be found such that:

$$K_{comb}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{M} \beta_k K_k(\mathbf{x}_i, \mathbf{x}_j) \tag{4.7}$$

is a convex combination of M kernels with $\beta_k \geq 0$, where each kernel, K_k, uses a distinct set of features [101]. In the current case, each distinct set of features is a different view of the data (Section 4.2.1). The general outline of the algorithm is to first combine the kernels with $\beta_k = 1/M$, find α, and then iteratively continue optimizing for β and α until convergence. Complexity analysis is given in Appendix C.2.

To solve for β, assuming a fixed set of support vectors (α), the following semi-infinite linear program has been proposed [101]:

$$\begin{aligned}
\max \quad & \theta & (4.8)\\
\text{w.r.t.} \quad & \theta \in \mathbb{R}, \beta \in \mathbb{R}^K
\end{aligned}$$

subject to the constraints:

$$\mathbf{0} \leq \beta \tag{4.9}$$

$$\sum_k \beta_k = 1$$

$$\sum_{k=1}^{M} \beta_k S_k(\alpha) \geq \theta$$

for all $\alpha \in \mathbb{R}^N$ with $\mathbf{0} \leq \alpha \leq \mathbf{1}C$ and $\sum_i y_i \alpha_i = 0$, and where $S_k(\alpha)$ is defined in Equation 4.5. M is the number of kernels to be combined. This is a semi-infinite linear program because all of the constraints in Equation 4.9 are linear, and there are infinitely many of them, one for each $\alpha \in \mathbb{R}^N$ satisfying $\mathbf{0} \leq \alpha \leq \mathbf{1}C$ and $\sum_i y_i \alpha_i = 0$ [47]. θ cannot go to ∞ because of the constraint $\sum_{k=1}^{M} \beta_k S_k(\alpha) \geq \theta$. Finding the maximal *theta* corresponds to finding a saddle-point of the following min-max optimization problem:

$$\max_{\beta} \min_{\alpha} \sum_{k=1}^{M} \beta_k S_k(\alpha) \tag{4.10}$$

If α is the optimal solution, then $\theta = \sum_{k=1}^{M} \beta_k S_k(\alpha)$ would be minimal satisfying Equation 4.9 for all α. More details on θ can be found in [103].
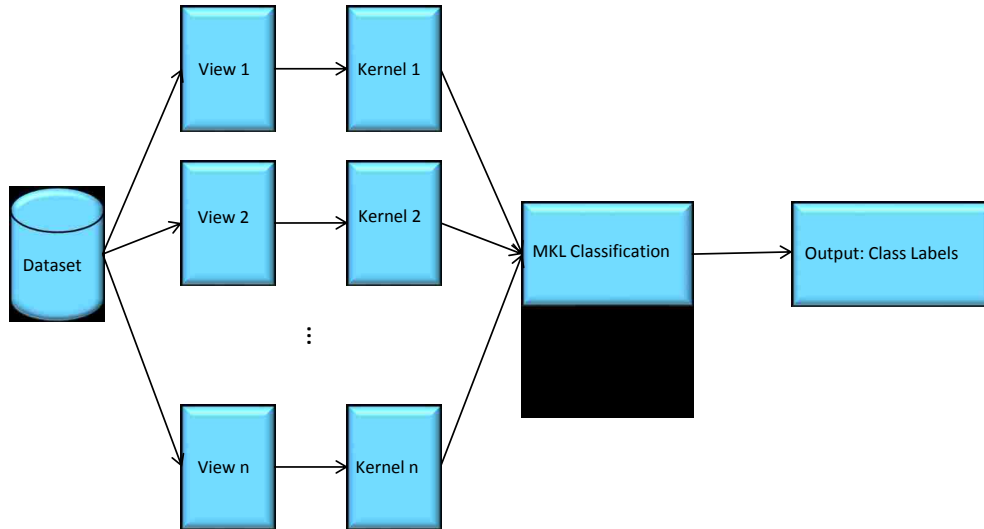
Figure 4.5: Architecture diagram for the MKL algorithm.

To find solutions for α and β, an iterative algorithm was proposed that first uses a standard support vector machine algorithm to find α (Equation 4.5), then fixes α to the solution, and solves Equation 4.8 to find β. Although this algorithm is known to converge, there are no known convergence rates [47]. Therefore, the following stopping criterion was proposed [101]:

$$\epsilon^{t+1} \geq \epsilon^t := \left| 1 - \frac{\sum_{k=1}^{M} \beta_k^t S_k(\alpha^t)}{\theta^t} \right| \tag{4.11}$$

This method of multiple kernel learning has been found to be very efficient. Solving for α and β with as many as one million examples and twenty kernels has been shown to take just over an hour [101]. It is important to note that this optimization problem only needs to be solved once, as the support vectors (α) and kernel weights (β) can be used to classify all of the newly collected data. A general architecture diagram of this method is given in

Figure 4.5.

4.3 Experimental Setup

In this section, I present my results on a dataset composed of 1,556 samples, 780 malicious programs and 776 benign programs. I also present results on a separate dataset composed of 20,936 malicious samples.

The metrics I use to quantify the results are classification accuracy (TPs+TNs/(# of samples)), receiver operating characteristic (ROC) curves [16], area under the ROC curve (AUC) [20], and the average time it takes to classify a new instance. I look at three combination strategies: just static views, just dynamic views, and a combination with all six of the data views. I compare the combined kernel method against methods based on a single data view. I conclude this section with some interesting observations based on these results.

To perform the experiments, I used a machine with quad Xeon X5570s running at 2.93GHz with 24GB of memory. To perform the multiple kernel learning, I used the modular Python interface of the Shogun Machine Learning Toolbox [102].

4.4 Results

Accuracy. Table 4.2 presents results for the individual kernels as well as the combined kernels using 10-fold cross validation. The three best performing antivirus programs (out of 11 that I had available licenses) are also shown. For the antivirus program results, it is important to emphasize that the malicious dataset was not composed of zero-day malware, but rather malware that is at least 9 months to one year old. Despite this, the worst performing data view based on the file information feature vector still had ∼6% better

Method	Acc (%)	FPs	FNs	AUC
All Six Data Views	**98.07**%	16	**14**	**.9978**
Three Static Views	96.14%	36	24	.9934
Two Dynamic Views	88.75%	88	87	.9509
Binary	88.11%	93	92	.9437
Disassembled Binary	89.97%	71	85	.9483
CFG (4-graphlets)	88.05%	88	98	.9361
Dynamic Instructions	87.34%	92	105	.9335
Dynamic System Call	87.08%	88	113	.9368
File Information	84.83%	126	110	.9111
Avira	78.46%	4	331	n/a
BitDefender	75.26%	7	378	n/a
Kaspersky	71.79%	**0**	439	n/a

Table 4.2: The classification accuracy, number of false positives and false negatives, and the full AUC values for 776 instances of benign software versus 780 instances of malware. Statistically significant winners are bolded.

Method	.01	.05	.1	.25	.5	Full AUC
All Six Data Views	**.9467**	**.9867**	**.9933**	**1.0**	**1.0**	**.9978**
Three Static Views	.9224	.9634	.9812	**1.0**	**1.0**	.9934
Two Dynamic Views	.5000	.8487	.8882	.9605	.9803	.9509
Binary	.5369	.7919	.8523	.9262	.9597	.9437
Disassembled Binary	.5574	.7347	.8699	.9628	.9878	.9483
CFG (4-graphlets)	.4182	.6814	.8724	.9378	.9675	.9361
Dynamic Instructions	.3401	.6395	.7211	.9116	.9796	.9335
Dynamic System Call	.5266	.7337	.8580	.9586	.9763	.9368
File Information	.0946	.4527	.7703	.9054	.9730	.9111

Table 4.3: AUC values for the full ROC curve as well as values for five different false positive rates: .01, .05, .1, .25, and .5. The combined kernel composed of all six data views performs the best at all values.

classification accuracy than the best antivirus program. All but one of the false positives found by the antivirus programs were actually confirmed to be true positives as discussed later in this section.
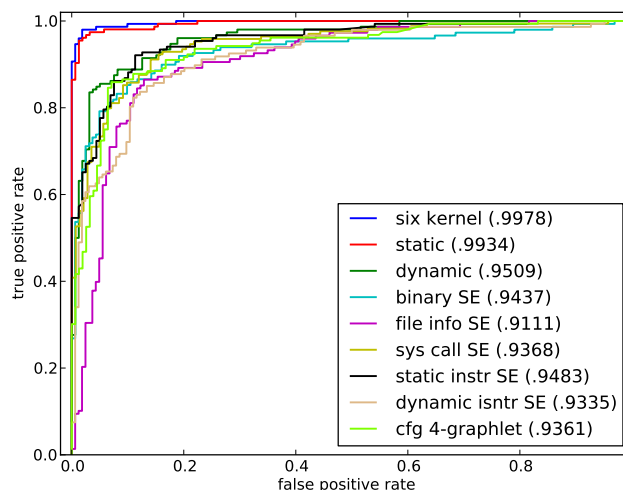
(a) Full ROC Image
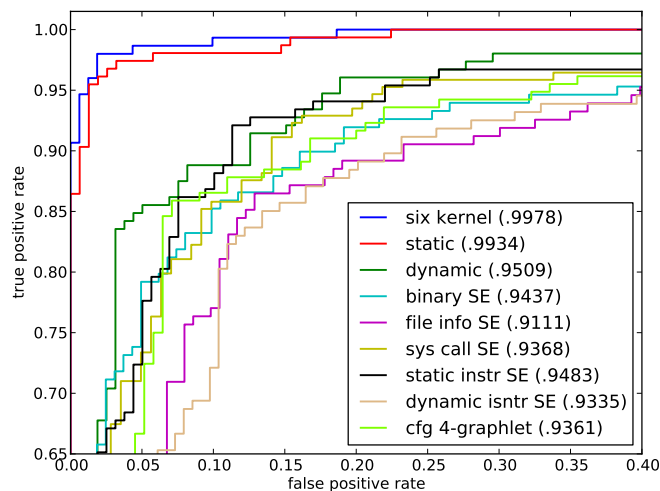
(b) Zoomed ROC Image

Figure 4.6: ROC curves for all six individual kernels, a kernel based on the three static views, a kernel based on the two dynamic views, and the combined kernel composed of all six data views. It is easy to see that the kernel based on all six data views performs significantly better than the other kernels. The full AUC values for the kernels are are listed in legend in parenthesis.

The best performing method was the combined kernel that used all six data views and achieved an accuracy of 98.07%. Although the static views also performed very well (96.14%), adding dynamic information improved overall performance with respect to accuracy and false positives. All of the single data views achieved between 84% to 89% accuracy with the single data view winner being the disassembled binary at 89.97%.

ROC Curves / AUC Values. To analyze the different data views with respect to different false positive thresholds, I looked at the ROC curves and various ROC points representing different false positive rates. Figure 4.6 plots all the ROC curves together (including the combined kernels) along with a zoomed version of the curve. Figure 4.6 (b) is particularly informative as the combined kernel, which includes all six data views, clearly performs better than any single data view or the two other combined kernels for all false positive rates. If there are certain time and/or other resource constraints, Figure 4.6 (b) also shows that reasonably high results can be achieved by just using the kernel based on the three static views which is valuable in time-sensitive environments.

Table 4.3 displays the full AUC value, as well as the AUC values for five different false positive rates: 0.01, 0.05, 0.1, 0.25, and 0.5. By integrating all six data views, an AUC value of .9467 can be achieved with a .01 false positive rate, significantly higher than any other kernel based on a single data view, which adds support for the power of my approach and multiple kernel learning in general. The file information data view performs the worst with respect to this metric, and as I explain at the end of this section, I expect this is due to misclassifying a large percentage of files that are packed and/or have abnormal entropy values.

Learned Kernel Weights. The kernel weights learned from Equation 4.8 are an interesting way to look at how informative different data views are with respect to the classification accuracy. The weights I found for the different data views are shown in Table 4.4. It is interesting to note that the weights are *not* representative of how well the single data view

Data View	All Views	Static Views	Dyn Views
Binary	.2248	.2671	N/A
Disassembled	.2576	.3284	N/A
CFG	.1559	.4046	N/A
Dyn Instrs	.3299	N/A	.5817
System Calls	0.0	N/A	.4183
File Info	.0319	N/A	N/A

Table 4.4: The kernel weights learned from Equation 4.8 for the different kernel combinations examined.

does at classification. For example, the dynamic instruction view has the highest kernel weight among the weights for all six views, but in these experiments, the binary, disassembled and CFG all individually have higher accuracy than the dynamic instruction view.

Speed. Due to the fact that computing the kernel for each dataset, finding the kernel weights for the combined kernels, and finding the support vectors for the support vector machine are all essentially $\mathcal{O}(1)$ operations (all of these calculations need only to be performed once, offline), I will focus the analysis of the timing results on the average amount of time it takes to classify a new instance. The time to find the kernel weights and support vectors for the kernel composed of all six data views, averaged over 10 runs, was only 0.86 seconds.

Given a new program instance to classify, there are two or three steps to be performed, depending on whether a dynamic data view is used:

1. Run the instance in a virtual machine keeping a log of the instructions and system calls the program performs.

2. Transform the data view into the appropriate data representation.

3. Classify the transformed data instance according to Equation 3.11.

In the timing results, I allow five minutes to collect the dynamic trace data. Transforming
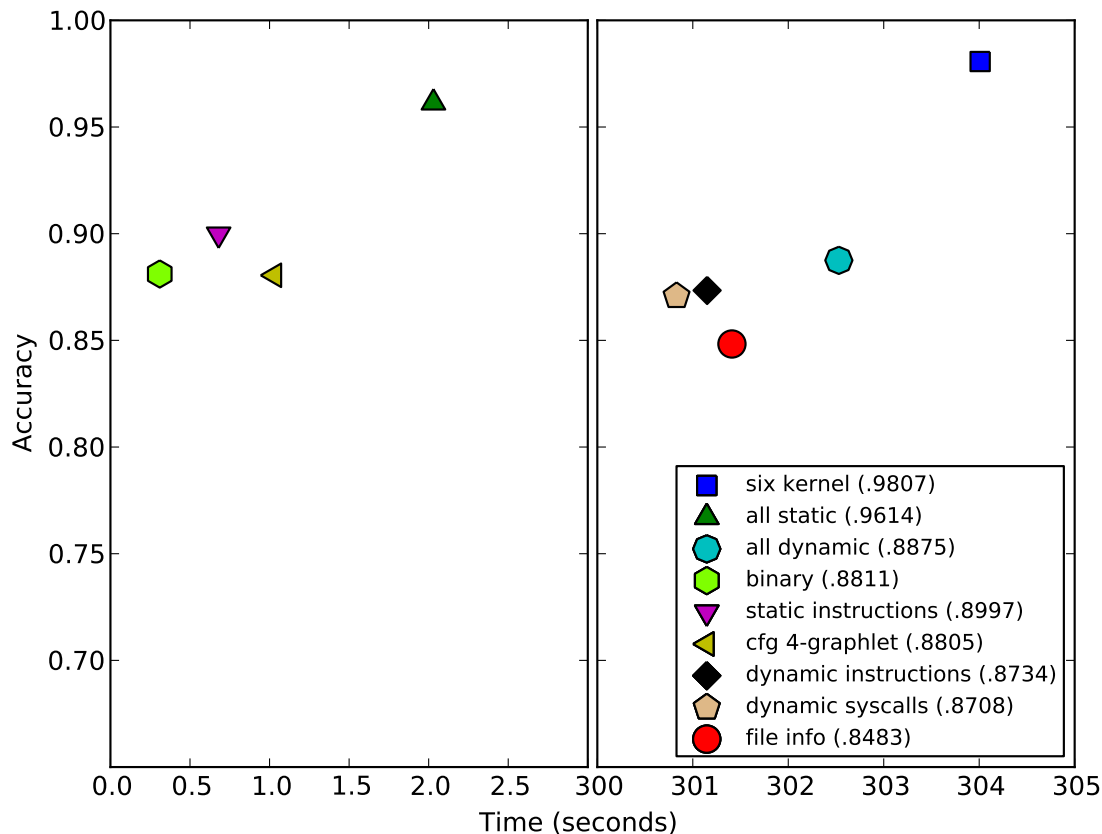
Figure 4.7: Plot demonstrating the trade-off between accuracy and time to classify. Time is in seconds and the x-axis is the time it takes to first collect the dynamic trace (for the dynamic data views), transform the data instance into its representation, and finally classify the instance.

the data to its appropriate representation could mean several different things for each of the different data views. For instance, the data might have to be disassembled, the 2-grams in the disassembled data found, and finally the Markov chain built; the control flow graph might have to be built and the number of graphlets with a specific structure found; in the case of the file information data view, the statistics of Table 4.1 will have to be collected; etc.

Method	Trace	Data Transformation	Classify	Total
All Six Data Views	300.0s	3.12s	0.21s	303.52s
Three Static Views	n/a	1.18s	0.13s	2.03s
Two Dynamic Views	300.0s	1.94s	0.06s	302.53s
Binary	n/a	0.26s	0.05s	0.31s
Disassembled Binary	n/a	0.63s	0.05s	0.68s
CFG (4-graphlets)	n/a	0.97s	0.06s	1.03s
Dynamic Instructions	300.0s	1.10s	0.04s	301.15s
Dynamic System Call	300.0s	0.82s	0.01s	300.83s
File Information	300.0s	1.41s	0.01s	301.41s

Table 4.5: The average time it takes from receiving the raw data until a classification decision. This time includes running the program for the dynamic views (5 minutes is allowed for the tracing), transforming the data to the appropriate representation (which includes the time to disassemble the code, collecting all of the 2-grams from the trace files, building the Markov chains, etc.), and finally computing the class from the decision function of Equation 3.11.

The timing results, which are broken down into the three stages, are presented in Table 4.5 and are pictorially featured in Figure 4.7. These results were averaged over the entire dataset. Classifying an instance takes very little time, and the only real bottleneck is the time to collect the dynamic trace. The combined kernel composed of the three static views is especially impressive as it has great performance and it takes only 2.03 seconds on average to transform the data and make a classification decision. Using the combined kernel based on the static data views, assuming 2.03 seconds per data instance, it would be possible to classify up to 42,561 new samples each day. Again, the methods presented in this chapter are highly parallelizable, and these methods will scale with more computational resources because the kernel computations for classifying a new program instance can all be done in parallel.

Testing on a Large Malware Sample. As explained earlier in this section, obtaining a large benign dataset is difficult. On the other hand, besides the 780 malicious programs discussed earlier, I had an additional 20,936 malware samples at my disposal from the Los

Method	Accuracy (%)
All Six Data Views	**97.97**%
Three Static Views	95.27%
Two Dynamic Views	91.57%
Binary	86.76%
Disassembled Binary	88.23%
Control Flow Graph (4-graphlets)	85.92%
Dynamic Instructions	88.42%
Dynamic System Call	86.38%
File Information	84.46%
BitDefender	57.55%
Avira	56.01%
Kaspersky	55.32%

Table 4.6: The classification accuracy on a validation dataset consisting on 20,936 malicious samples.

Alamos National Laboratory repository. I am treating this data as a validation set to test the generalization accuracy of my methods. I first train on all of the 1,556 samples (780 malicious and 776 benign), find the β's and α's, and finally classify the held out 20,936 malicious samples as either benign or malicious according to Equation 3.11.

The results for this experiment are shown in Table 4.6. The classification accuracies are similar to those of previous section with the exception that the signature-based antivirus programs do worse. This is due to the fact that in the original dataset, these methods would always get close to 100% accuracy on the benign samples giving a positive skew to their results because they had very few false positives on this dataset. As with the previous results, by combining data views in the multiple kernel learning framework, I was able to achieve a large increase in classification accuracy. The results presented in Table 4.6 suggest that these methods would generalize very well to larger datasets given that they performed well on this validation set.

Observations. A well-known problem in any supervised machine learning setting is the

integrity of the training dataset. In training the classifier, I assumed that the labeled benign samples in the dataset were actually benign. This was reasonable as the executables were taken from clean installations of commercial software. To test this hypothesis, I ran the classifier based on the combined kernel, with all six data views, using 10-fold cross validation 50 times and counted how many times a data instance was classified incorrectly. I found 19 data instances that were consistently misclassified, 8 which were labeled as malicious and 11 which were labeled as benign. 5 of the 11 benign samples were found to actually be malicious using VirusTotal [114]. It is interesting that this method was able to reduce the original 1,556 instance dataset to a manageable size of 19 suspicious samples, making closer manual inspection of these files more manageable. Note that I did not correct the dataset for the experiments in this chapter, and these 5 files are still considered false positives in the previous results.

Traditional static analysis techniques have been shown to be insufficient given the rise of newer malware obfuscation techniques [79]. Due to these limitations, I chose to include dynamic data views to improve malware classification despite the time constraints these data collection methods impose. To further analyze the pitfalls of the static data views, I again ran the combined kernel with all six data views, a kernel with all of the static data views, a kernel with all of the dynamic data views, and the six separate kernels, one for each of the six different data views 50 times, keeping track of the files that were consistently misclassified with respect to each kernel. In each of the 50 runs, 10-fold cross-validation was performed with a random split.

Table 4.7 shows the percentage of files consistently misclassified which were packed. The kernel based on the binary data had significant problems classifying packed benign instances, as one would expect, with 43.75% of the false positives being packed (only 19.59% of all benign instances in the training data were packed). On the other hand, using dynamic data views, the percentage of false positives that was packed is the same as the packed percentage of the training data, which suggests that the kernels based on these data views were not

Method	Benign	Malicious
All Six Data Views	**0.00**%	70.00%
Three Static Views	20.00%	36.84%
Two Dynamic Views	18.75%	45.95%
Binary	43.75%	43.14%
Disassembled Binary	10.20%	53.85%
CFG (4-graphlets)	13.89%	55.10%
Dynamic Instructions	20.00%	38.24%
Dynamic System Call	21.62%	**32.56**%
File Information	28.09%	34.31%
Full Dataset Average	19.59%	47.56%

Table 4.7: Percentage of files that were packed and misclassified over all 50 runs with the different kernels. Note the average percentage of packed files in the entire dataset is 19.59% and 47.56% for benign and malicious files, respectively.

Method	Benign	Malicious
All Six Data Views	7.43	6.77
Three Static Views	7.41	6.91
Two Dynamic Views	6.26	7.50
Binary	7.42	7.01
Disassembled Binary	6.15	7.58
CFG (4-graphlets)	6.12	7.66
Dynamic Instructions	6.29	7.57
Dynamic System Call	6.41	7.55
File Information	7.77	5.98
Full Dataset Average	6.34	7.52

Table 4.8: Average entropy of files misclassified over all 50 runs with the different kernels. Note that the average entropies over the entire dataset are 6.34 and 7.52 for benign and malicious files respectively.

deceived by the packer. Although dynamic traces of packed files also have an unpacking "footprint," it has been shown that 5 minutes for a dynamic trace is enough time for a significant number of the instructions to represent the true behavior of the program [87].

Table 4.8 shows the average entropy of files that were consistently misclassified. The link between entropy and files being packed is well-known [75], therefore similar results to Table 4.7 are seen. Again, the two dynamic views' classification accuracies seem to be independent of the entropy as the average entropy of the files they misclassify corresponds to the average entropy of the entire dataset. Also, much like the previous results, the binary data view had problems with classifying instances whose entropies differed significantly from the norm. As expected, the file information data view had the most problems with entropy (remember that entropy is one of the seven features for this view), with average entropy of misclassified benign and malicious files being 7.77 (average in dataset: 6.34) and 5.98 (average in dataset: 7.52), respectively.

Having a dynamic tracing tool that is able to evade detection from the program being traced is essential to get an accurate picture of how the program actually behaves in the wild. Unfortunately, some malware can detect if it is being run in a virtual environment and/or being traced [12]. I chose the Intel Pin program [73] because it allowed the collection of both instructions and system calls simultaneously, is stable, and is currently being supported by Intel. Unfortunately, it does not make an effort to be a transparent tracing tool like the Ether framework [35]. The dynamic instruction data view's classification accuracy in this Chapter is lower than that reported in the literature [1,31], and I suspect that this is in part due to Intel Pin not being transparent and the malware altering its own behavior.

To test this hypothesis, I looked at 8 malicious data instances that were consistently misclassified (over 50 runs) by the kernel based on the dynamic instruction data view. I first collected the dynamic traces with both Intel Pin and Ether and then computed the kernel values between the resulting Markov chains of these traces. These results are displayed in Table 4.9. Note that a kernel value of zero represents completely orthogonal behavior between the two traces (no instruction transitions in common) and a kernel value of one represents exactly the same program behavior between the two traces, which would be highly unlikely even if the same tracing tool was used for both samples (a kernel value of 0 is also highly

Malware Sample	$K(\text{Trace}_{pin}, \text{Trace}_{ether})$
sample0	.9010
sample1	.8392
sample2	.8171
sample3	.7719
sample4	.6424
sample5	.5864
sample6	.5399
sample7	.3725

Table 4.9: The kernel values between two Markov chains of the same program's dynamic instruction trace, one trace run with Intel Pin, and one trace run with the Ether framework. The kernel values were computed using Equation 4.1.

unlikely for obvious reasons). Although this is a coarse measure as to whether the program alters its behavior, it does give some useful information as to why these instances could have been classified incorrectly. The lower values of sample4, sample5, sample6, and sample7 are particularly interesting and suggest that their dynamic instruction traces are substantially different under the different tracing tools.

4.5 Conclusion

In this chapter I have shown that significant benefits, with respect to both classification accuracy and number of false positives, can be gained when malware researchers use all of the information about executables that are available to perform classification, and not just a single data view. I was able to achieve an accuracy of 98.07% on a dataset of 780 malware and 776 benign instances. I showed that while I had 16 false positives in this dataset, several of these were later confirmed to be true positives. The ROC curve analysis showed significant increases in performance when the data views were combined, and also that acceptable performance can be achieved with just static views in a resource constrained

environment.

I demonstrated several interesting observations about the results, illustrating some of the pitfalls of both static analysis and dynamic analysis techniques. Namely that static data views have problems classifying instances that are packed and/or have abnormal entropy values. I also gave some support to the importance of having a dynamic tracing tool that is capable of evading detection from the malware so that a truly representative sample of how the malware actually behaves in the wild can be collected.

An important distinction should be made about the target audience of this line of research, namely, that it is not intended to be used in a home computing environment. Significant computational resources need to be devoted to collect the different views. A use case would be first collect a program to be analyzed, either through user submissions or live data feeds (e.g. harvesting email attachments). Then, send this file to a cluster of machines devoted to collecting dynamic traces and other appropriate views. Finally, send the final classification score to your incidence response team for further action.

Chapter 5

Multiple Kernel Learning Clustering

Accurately classifying executables as malicious or benign is only the first step towards mitigating an attack. Clustering malicious programs has become an important next step for decreasing the response time of a computer security incident. The majority of new malware is created through simple modifications of older instances, such as adding some new functionality or slightly modifying the code base to avoid detection [124]. Typically, a reverse engineer is assigned to understand the capabilities of a new instance of malware and how to mitigate the damage it has caused. Being able to assign the unknown instance to a known family of malicious programs, as illustrated in Figure 5.1, allows the reverse engineer to apply techniques learned from reverse engineering previously seen instances of this family, greatly reducing the time it takes to develop an appropriate response.

Similar to the previous chapters, I begin by defining a kernel, a general measure of similarity, for each view and wish to find a linear combination of these kernels that can be used in a spectral clustering framework. Although well-established methods have been proposed to solve the multiple kernel learning problem in a classification setting [9, 62, 101], multiple kernel learning with a clustering objective function has had far less exposure.

In this work, I propose a novel solution to the multiple kernel learning (MKL) problem
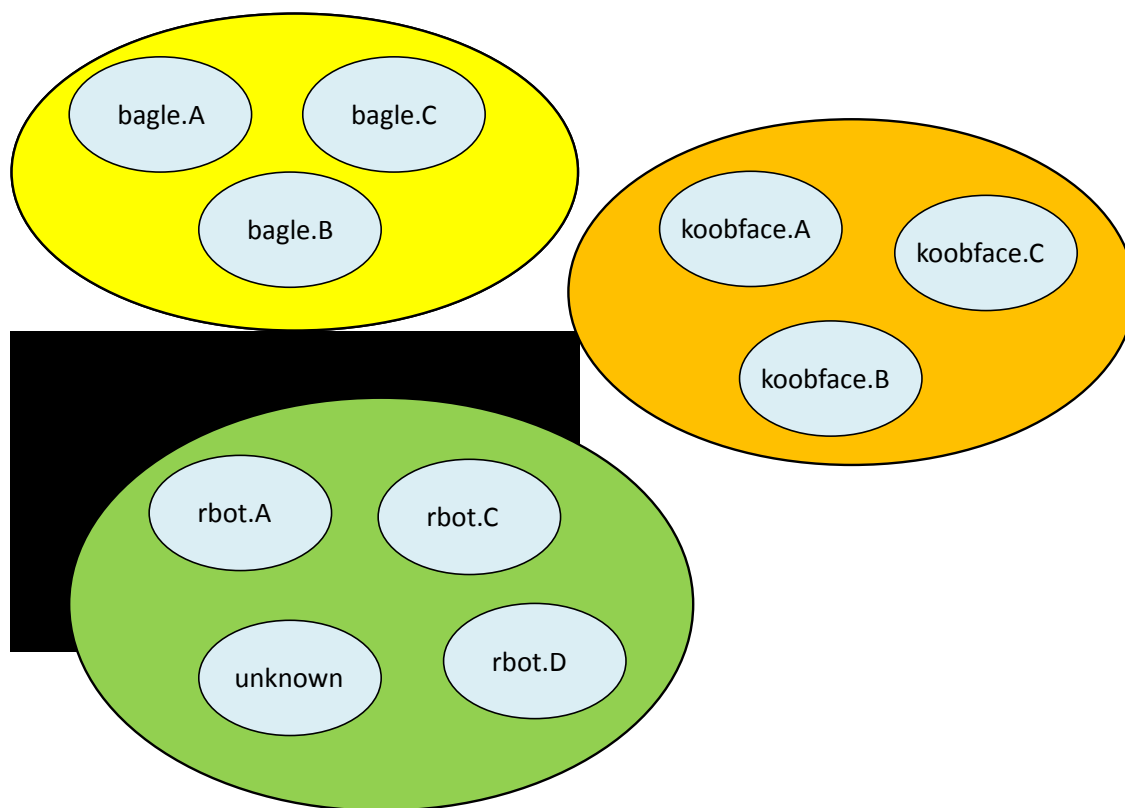
Figure 5.1: An example of clustering malware.

based on a spectral clustering objective function [74]. I show how to modify the unnormalized and the normalized Laplacians [26], which have both been used with success in spectral clustering, so that they can explicitly take the multiple views of a dataset into account. A spectral clustering objective function was chosen for its ability to take advantage of kernel matrices and for its superior results on a wide range of clustering problems [74, 118, 122]. I show how this new objective function can be formulated as an equivalent semidefinite program and efficiently solved using off-the-shelf interior point methods [3].

The primary benefit of the proposed approach is its ability to cluster instances based on multiple views with no a priori information as to which view is the most informative, something that is needed for some earlier attempts at clustering with multiple views [68, 126].

Another important benefit is that the kernel weight vector found can be viewed as a way to rank the importance of the individual views. Some forms of data collection are expensive, e.g. collecting dynamic traces requires a large amount of resources and real-time. It would be useful to determine the value of each view and focus resources on collecting the views that seem to contribute the most to the clustering problem.

I use a variety of datasets to demonstrate the efficacy of the proposed methods, including several datasets taken from the UCI machine learning repository [38]. In addition to the UCI datasets, I have a dataset composed of 606 malicious executables separated into 12 distinct malicious families. Families consist of malware with a common codebase and inherited functionality. For example, there are unique instances of the Bagle family which is associated with mass-mailing spam and unique instances of the koobface family which is famously linked to gathering login information and sending unwanted Facebook messages. I compare the proposed method to several baselines: the best single view, simple kernel addition, multiple kernel learning based on a classification objective function, and two recently proposed methods using co-regularized spectral clustering [68]. The adjusted Rand index [49] is used to quantify the results.

5.1   MKL Clustering Algorithm

Let the multiple view dataset be denoted by $\mathcal{D} = D_1 \cup \cdots \cup D_M$ where there exist M distinct views, and each view is composed of n feature vectors, one for each data instance. In all of the work that follows, I begin by defining a kernel on each view. As stated in the previous chapter, the goal of multiple kernel learning is to find an optimal weight vector, β, for the linear combination of kernels, $\sum_k^M \beta_k K_k$, such that $\beta_k \geq 0$ and $\sum_k^M \beta_k = 1$. In the spectral clustering framework presented here, each kernel matrix is treated as a weighted adjacency matrix.

Typically, multiple kernel learning algorithms have been developed to extend the support

vector machine (SVM) optimization problem [101]:

$$\max_{\beta \in \mathbb{R}^k} \min_{\alpha \in \mathbb{R}^n} \quad \sum_{k=1}^{M} \beta_k S_k(\alpha) \tag{5.1}$$

$$\text{s.t.} \quad \beta \geq \mathbf{0}$$

$$\sum_{k}^{M} \beta_k = 1$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

for all $\alpha \in \mathbb{R}^n$ with $\mathbf{0} \leq \alpha \leq \mathbf{1}C$, where C allows for soft margins [22], and where

$$S_k(\alpha) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K_k(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{n} \alpha_i \tag{5.2}$$

Equation 5.2 is the classic support vector machine optimization problem for a single kernel, $K_k(\cdot, \cdot)$. Equation 5.1 constructs a maximum-margin hyperplane separating two classes with respect to both the support vectors, α, and the kernel weights, β.

While the kernel weights found with Equation 5.1 are optimal for a classification setting, it is not clear that they are optimal for the clustering domain. Furthermore, it is unrealistic to have labeled data for all domains, making Equation 5.1 unusable. This section describes an optimization framework that modifies the spectral clustering objective function to take multiple views into account. I begin by defining the spectral clustering problem with a single view, then go on to show how to extend spectral clustering to take multiple views into account for both the unnormalized and normalized Laplacian. Finally, I show how to state the optimization problem as a semidefinite program and outline an algorithm to find the optimal β's.

Given an adjacency matrix K and a diagonal matrix D with entries $d_{ii} = \sum_j k_{ij}$, spectral clustering finds an approximation to the relaxed mincut problem for graphs, optimizing RatioCut [74] by using the unnormalized graph Laplacian:

$$L_{un} = D - K \tag{5.3}$$

and optimizing normalized cut [74] by using the normalized graph Laplacian:

$$L_{norm} = I - D^{-1}K \tag{5.4}$$

The goal is to find an indicator matrix associating each vertex in the graph to a partition of the graph. The relaxed version of mincut that spectral clustering solves allows entries of the indicator matrix to be real-valued with the following formulation:

$$\min_{U \in \mathbb{R}^{n \times k}} \quad \text{tr}(U^T L U) \tag{5.5}$$
$$\text{s.t.} \quad U^T U = I$$

where U can be directly solved for as the top k eigenvectors associated with the smallest non-zero eigenvalues of L. With U found, a clustering algorithm such as k-means can be run with the rows of U serving as the features.

To extend Equation 5.5 to take multiple views of the data into account, Equations 5.3 and 5.4 need to be modified to use the multiple view version of the unnormalized Laplacian:

$$L_{un}(\beta) = \sum_i^M \beta_i D_i - \sum_i^M \beta_i K_i \tag{5.6}$$

and the multiple view version of the normalized Laplacian:

$$L_{norm}(\beta) = I - \left(\sum_i^M \beta_i D_i \right)^{-1} \left(\sum_i^M \beta_i K_i \right) \tag{5.7}$$

The problem is to now find β such that Equation 5.5 is minimized:

$$\min_{U \in \mathbb{R}^{n \times k}} \quad \text{tr}(U^T L(\beta) U) \tag{5.8}$$
$$\text{s.t.} \quad U^T U = I$$

To solve for both U and β, I propose an iterative algorithm. Assuming no a priori information as to which view is more informative, I begin by initializing $\beta_i = 1/M$. U is
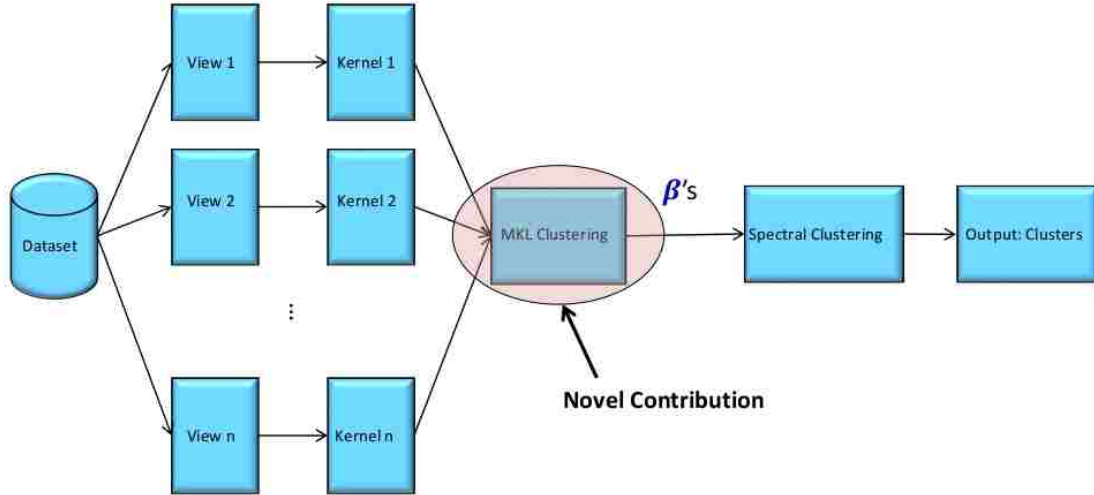
Figure 5.2: Architecture diagram for the MKL clustering algorithm.

then found through an eigendecomposition as is standard with spectral clustering. With an initial U, β can be found with a semidefinite program.

Using the identity that the trace function is equivalent to the sum of the eigenvalues of a matrix, $\text{tr}(A) = \sum_i \lambda_i$, and the relationship between the eigenvalues and the singular values for a matrix, $\sigma_i = \sqrt{\lambda_i}$, the optimal β vector with respect to Equation 5.8 given U can be solved for with the following semidefinite program:

$$
\begin{aligned}
min \quad & ||A(\beta)||_* + \frac{1}{2}\beta^T C \beta \\
s.t. \quad & G\beta \preceq h
\end{aligned}
\tag{5.9}
$$

where

$$G = \begin{bmatrix} -1 & 0 & \cdots \\ 0 & \ddots & 0 \\ \vdots & 0 & -1 \\ \cdots & 1 & \cdots \\ \cdots & -1 & \cdots \end{bmatrix} \tag{5.10}$$

and

$$h = [\mathbf{0} \ 1 \ -1]^T \tag{5.11}$$

The top negative identity matrix of G and $\mathbf{0}$ vector in h enforce non-negative β's while the last two constraints force the β's to sum to one. $|| \cdot ||_*$ is the nuclear norm defined as $\sum_{i=1}^{n} \sigma_i$ where the σ_i's are the singular values of the matrix. The quadratic regularization term smooths the β vector according to the diagonal matrix C.

$A(\beta)$ in Equation 5.9 is a linear combination of matrices, $A(\beta) = \sum_{i=1}^{M} \beta_i A_i$, which is built by manipulating Equations 5.6 and 5.7. A_i serves as a proxy for the objective of Equation 5.8, and a linear combination of the A_i's is able to be used due to the linearity of the trace function. For the unnormalized case:

$$A_i = U^T (D_i - K_i) U \tag{5.12}$$

and for the normalized case:

$$A_i = U^T \left(-K_i \left(\sum_{j}^{M} \beta_j D_j \right)^{-1} \right) U \tag{5.13}$$

The derivations for Equations 5.12 and 5.13 are given in Appendices A.1 and A.2. The β_j's of Equation 5.13 and the U matrix are the values found in the previous iteration.

Algorithm 1 outlines my framework for finding the optimal kernel weights for multiple kernel learning with a clustering objective function. The final steps of the algorithm follow the spectral clustering framework, but using the multiple view Laplacian, $L(\beta)$, in place of the standard, single-view Laplacian. Figure 5.2 gives a pictorial overview of the process.

Algorithm 1 Multiple kernel learning clustering algorithm, iteratively finds optimal β and U, and then performs spectral clustering.

Require: initial β

   $U \leftarrow$ solve Equation 5.8

   score $\leftarrow \text{tr}(U^T L(\beta) U)$

   while score is decreasing **do**

      $\beta \leftarrow \min \; ||A(\beta)||_* + \frac{1}{2}\beta^T C \beta$

         s.t. $G\beta \preceq h$

      $U \leftarrow \min_{U \in \mathbb{R}^{n \times k}} \text{tr}(U^T L(\beta) U)$

         s.t. $U^T U = I$

      score $\leftarrow \text{tr}(U^T L(\beta) U)$

   end while

   construct $L(\beta)$

   find $U^{n \times k}$ with Equation 5.8

   Let $x_i \in \mathbb{R}^k$ be the new feature vector corresponding to
      the ith row of U

   $C_k \leftarrow k$-means with **x** as the new feature space

   return Clusters, C_k

5.1.1   Practical Concerns

Typically, β in Algorithm 1 would be initialized to reflect expert assumptions as to which data views are a priori the most informative. Because it is not realistic to have expert knowledge in all applications, in all of the results I present, I set each initial $\beta_i = 1/M$ where M is the number of views.

    The new feature space constructed by spectral clustering consists of a $U^{n \times k}$ matrix, where the columns are eigenvectors and the rows correspond to the new feature vectors, one for each instance in the dataset. Choosing an appropriate k is application dependent and often

Name	Num Instances	Num Attributes	Num Classes
iris	150	4	3
wine	178	13	3
wdbc	569	32	2
ecoli	336	8	8
glass	214	10	6

Table 5.1: UCI Datasets

relies on heuristics [74]. I use the heuristic $k = \lfloor \log(n) \rfloor$ where n is the number of instances in the dataset. This heuristic has been shown to work well on a variety of datasets [74].

The regularization term in Equation 5.9 depends on C, which controls how sparse the algorithm will make the weight vectors of the solution, with higher values resulting in a less sparse solution. In all of the experiments presented in this chapter, I set $C = I \cdot 0.1$.

5.2   Experimental Setup

5.2.1   UCI Datasets

I first compare my methods on several datasets taken from the UCI machine learning repository [38]. These datasets are given in Table 5.1 with the number of instances, attributes, and classes. Because none of these datasets naturally has multiple views, several synthetic views were created by defining different types of kernels, as well as kernels with different parameter values. The first kernel used was the Gaussian kernel:

$$K_G(\mathbf{x}, \mathbf{x}') = e^{-\frac{\lambda}{2} \sum_i (\mathbf{x}_i - \mathbf{x}'_i)^2} \tag{5.14}$$

I created three artificial views using this kernel for each UCI dataset by adjusting λ by different orders of magnitude (i.e. for the iris dataset I used values of 1.0, 0.1, and 0.01). A

Family Name	Number of Instances
bagle	49
bifrose	61
hupigon	76
koobface	31
ldpinch	44
lmir	69
rbot	11
sdbot	26
swizzor	76
vundo	56
zbot	43
zlob	64

Table 5.2: Malware Families.

polynomial kernel was also used:

$$K_P(\mathbf{x}, \mathbf{x}') = (\alpha \mathbf{x} \cdot \mathbf{x}')^p \tag{5.15}$$

where $p \in \{1, 2, 3\}$. This gives a total of six synthetic views for each of the UCI dataset.

5.2.2 Malware Dataset

The dataset which motivated this work is based on 606 samples of malicious programs separated into 12 distinct families, as labeled from antivirus vendors, and was collected over a 6 month period beginning in August 2011. The family names and the number of instances from each family are listed in Table 5.2.

Six distinct views of malware were used as described in the previous chapters, and include the binary bytes, disassembled instructions, the control flow graph, dynamic instructions, system calls, and a general file information feature vector.

Four of the six data views are sequence-based. The sequences are used to condition the

transition probabilities of a Markov chain as shown in Figure 4.2 and described in Chapter 3. For each of the data views based on the Markov chain representation, I use a Gaussian kernel (Equation 5.14) with $\mathbf{x}$ being the entries in the transition probability matrix of the given instance. Although more advanced graph kernels exist, I found this simple kernel to perform the best under most circumstances. I refer the reader to Chapter 4 for an in-depth description on computing kernels for each of the six views.

5.2.3   Competing Methods

The previously described method finds a weight vector, β, such that Equation 5.8 is minimized. The baseline methods I compare to include the most informative view, i.e., $\{\beta_i = 1, \beta_{j\neq i} = 0\}$. For my experiments I looked at every available view and only report the view with the best results. I also use a uniform combined kernel where $\{\forall i | \beta_i = 1/M\}$. The final method compared against that revolves around finding β is based on classic multiple kernel learning with a classification objective function [101]. To perform this optimization, I used the SHOGUN machine learning toolbox [102].

While I propose first combining the multiple views according to the weight vector β, and then finding the spectral embedding, U, used in the k-means step, other recent work has proposed a co-regularization scheme to combine kernels [68]. The first method proposed in [68] used pair-wise regularization:

$$\min_{U_i \in \mathbb{R}^{n \times k}} \quad \mathrm{tr}\left(U_i^T \left(L_i + \lambda \sum_{j \neq i}^{M} U_j U_j^T \right) U_i \right) \tag{5.16}$$
$$\text{s.t.} \quad U_i^T U_i = I$$

Each U_i is initialized by solving the spectral clustering objective function for that specific view. After initialization of the single views, Equation 5.16 is repeatedly solved for U_i,

holding all other U_j constant. After convergence, choosing which view to use for k-means is still an open question and often relies on a priori information. In my experiments, I clustered on all views and report the best results.

The second co-regularized spectral clustering algorithm presented is the centroid method [68]:

$$\min_{U_* \in \mathbb{R}^{n \times k}} \quad \text{tr}\left(U_*^T \left(\sum_i^M \lambda_i U_i U_i^T \right) U_* \right) \tag{5.17}$$
$$\text{s.t.} \quad U_*^T U_* = I$$

Like the pair-wise method, the centroid method is an iterative algorithm, first initializing U_i according to the Laplacian for the ith view, and then solving Equation 5.17 for U_*. Then, at the beginning of each iteration, it solves for U_i, $i = 1, \ldots, M$, with the modified Laplacian, $L_i + \lambda_i U_* U_*^T$ before solving for the updated U_*.

Unlike the pair-wise co-regularization, the centroid method finds a single feature space embedding, U_*. Although a single embedding is found, a priori information about the importance of the views is still explicit in Equation 5.17 as λ_i. In my experiments, I used several values for λ_i, including the uniform $\mathbf{1}$ vector, the classic MKL weights, and the weights found with both my unnormalized and normalized approaches. Only the results for the best performing weights are reported.

5.3 Results

Table 5.3 displays my results in terms of the adjusted Rand index [49] on both the UCI datasets and the malware dataset consisting of 606 malware instances from 12 distinct malicious families. Although each dataset had labels attached to it, the labels were only used to compute the adjusted Rand index and to compute the weights for the classification-based MKL algorithm. Neither my proposed algorithm nor the competing methods of [68] make

	SDP unnorm	SDP norm	Best View	Uniform Combination	Classic MKL	Centroid	Pair-wise
Malware	.8747	**.8768**	.8174	.8381	.8531	.8702	.8477
iris	.8923	**.9414**	.8030	.8343	.8236	.9124	.8687
wine	.4137	**.4331**	.3744	.4035	.3813	.4249	.3924
wdbc	.5072	**.5941**	.4510	.4863	.5019	.5463	.4913
ecoli	.6981	.7285	.6368	.6866	.7086	**.7342**	.6701
glass	.2737	**.2941**	.2318	.2438	.2613	.2705	.2318

Table 5.3: Adjusted Rand Index

use of these labels. My algorithm based on the normalized Laplacian performed the best on 4 out of the 5 UCI datasets, but not on the ecoli dataset, where the centroid method was the overall winner. As stated previously, this method needs a priori information describing which views of the data are more informative, with this information being encoded in the λ_i's of Equation 5.17. The λ values for the ecoli dataset that performed the best were the β's found with the presented normalized Laplacian approach.

The multiple kernel learning algorithms based on the unnormalized and normalized Laplacians have superior performance compared to all other methods on the malware dataset. I observed that the instances that were clustered incorrectly were very similar to the other malicious programs in that cluster. For instance, the Lmir and Hupigon trojans had some crossover in the clusters found, but both have many components related to keylogging and password stealing, implying that their views are similar. Likewise, the rbot and sdbot malware had some misplaced instances, but they both share a common lineage. sdbot is derived from the agobot family, and rbot is derived from sdbot. Both have irc command and control functionality with the distinction that sdbot is more focused on DDoS attacks and rbot has more keylogging features. Thus, although they were technically incorrect, their clustering could still provide useful information to reverse engineers.

Table 5.4 lists the time taken in seconds to perform the optimization. For my methods, as well as multiple kernel learning based on a classification objective function, this means finding

	SDP unnormalized	SDP normalized	Classic MKL	Centroid	Pair-wise
Malware	5.2363	5.8886	**3.8057**	14.1613	11.2255
iris	0.5322	0.8390	**0.4861**	2.5610	1.4104
wine	0.9268	1.0301	**0.6810**	3.2838	2.5101
wdbc	4.7849	5.0183	**3.3595**	12.1373	10.9162
ecoli	4.0267	4.7754	**2.1583**	7.7050	6.7499
glass	1.4543	1.7962	**0.7340**	3.4259	2.3781

Table 5.4: Time to perform optimization, in seconds

the weight vector, β, and the feature space embedding, U, of the Laplacian associated with the combined kernel. For the pair-wise and centroid methods, this means finding the U_v's and U_* of Equation 5.16 and Equation 5.17. Although classic MKL is the fastest algorithm on all applications, the presented methods are faster than the competing multiview clustering algorithms. In Section 5.4.1, I mention several ideas that could greatly decrease the runtime of my methods and improve the scalability of my methods.

5.4   Conclusion

In this chapter, I presented a novel multiple kernel learning method which is based on a clustering objective function. Specifically, I incorporated multiple views into the spectral clustering objective function, and showed how to optimize the multiple kernel learning weight vector using a semidefinite program. This algorithm iteratively finds the weight vector which is optimal with respect to the spectral clustering objective function which computes the feature space embedding used in the k-means algorithm.

I developed these techniques for use in malware clustering, which has become an important problem for the rapid response of a malware incident. Malicious programs are a natural fit for multiple kernel learning methods as they contain many, well researched views such as dynamic traces, binary data, the disassembled file, and the control flow graph. I have shown

that this method has superior performance compared to several baseline methods as well as two state-of-the-art multiple view spectral clustering algorithms.

5.4.1 Future Work

Because the proliferation of malware has drastically increased over the years, the scalability of any proposed method is very important. The eigendecomposition, which must be performed during each iteration of the algorithm to get $U \in \mathbb{R}^{n \times k}$, has a naïve runtime of $\mathcal{O}(n^3)$ and becomes prohibitive for larger datasets. For the purposes of the presented algorithm, only the top-k eigenvectors need to be acquired, which has been shown to be efficiently computable by a decentralized algorithm [60]. Having a distributed means to computing the top-k eigenvectors would greatly increase the scalability of this method.

Antivirus vendors have a constant incoming supply of malware, and models that can incrementally cluster these instances would be very advantageous. There has been some work on incremental spectral clustering techniques. In [81], the authors show how to update the eigenvectors used for clustering in the event of the addition or deletion of an instance. This is important for several reasons, namely avoiding the computation of recomputing all of the eigenvectors. Also, newer instances of malware are constantly being collected and older instances of malware become less relevant to the current malicious landscape and should be pruned from the dataset. An interesting direction for future research would be to extend their work to take multiple data views into account.

Chapter 6

Malware Phylogenetics

When beginning the process of understanding a new, previously unseen sample of malware, it is advantageous to leverage the information gained from reverse engineering previously seen members of that instance's *family* because techniques learned from related instances can often be applied to a new program instance. A malware family is a group of related malware instances which share a common codebase and exhibit similar functionality (e.g. different branches in a software repository). Unlike Chapter 5 where the goal was to find clusters representing the families, this chapter focuses on extracting a more detailed picture about the relationships between the malware instances within a given family.

In contrast to Figure 5.1, Figure 6.1 exhibits the proposed output of the ideal phylogenetic algorithm. This figure clearly shows the evolution of the bagle virus [59] as a directed graph. The information presented in Figure 6.1 is invaluable for a reverse engineer tasked with understanding specific instances within a malware family as well as the general evolution of the family. A key problem in developing an appropriate response to a new malware infection is that of *attribution*: determining the author or organization behind the malware. The creation of a phylogenetic graph is a critical first step in malware attribution.

In this chapter I describe a method based on the graphical lasso that finds a Gaussian
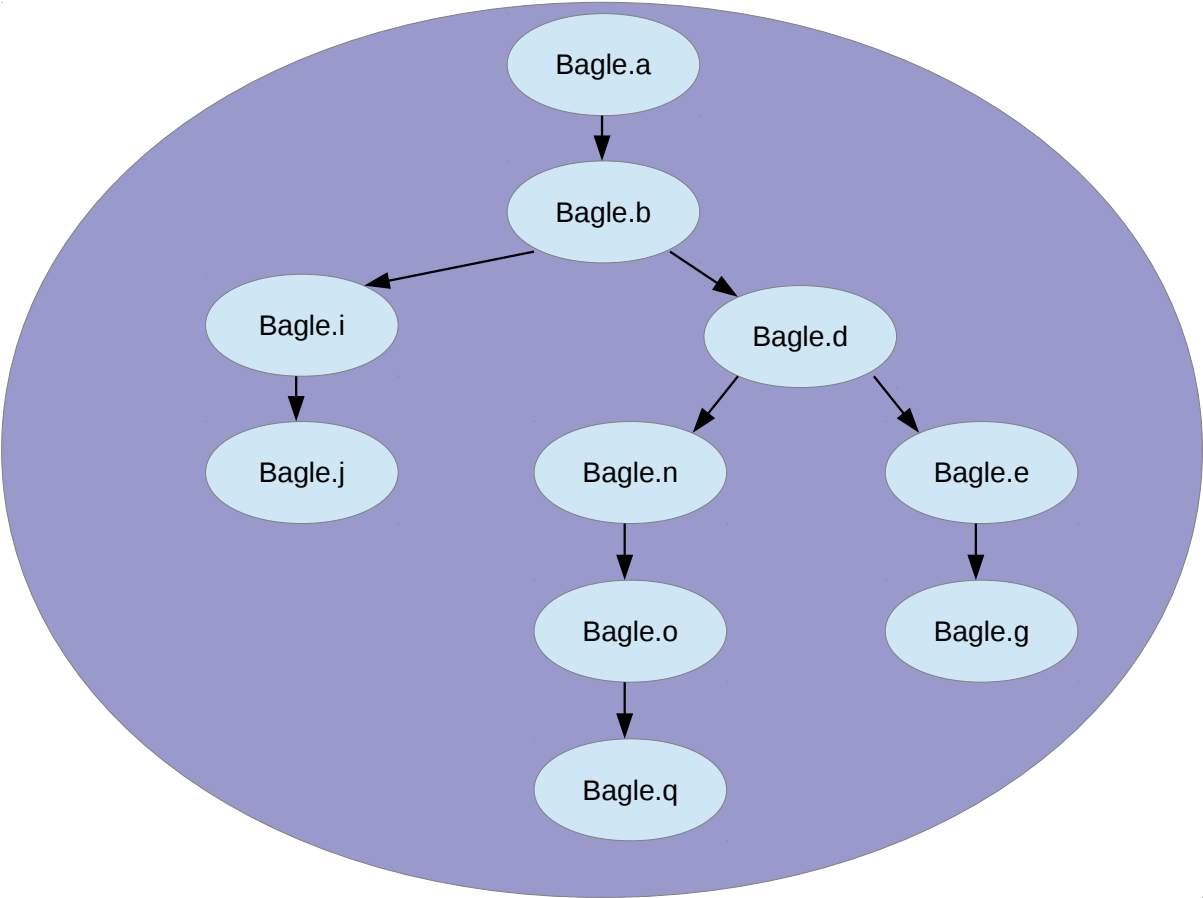
Figure 6.1: An example of creating a phylogenetic graph for the bagle worm.

graphical model where the malware instances are represented as nodes, and the evolutionary relationships are represented as edges. I present a novel extension to a standard algorithm that shows how to incorporate multiple views of the data. Finally, I mention several heuristics to force directionality within the Gaussian graphical model, allowing analysts to see exactly how the family has evolved.
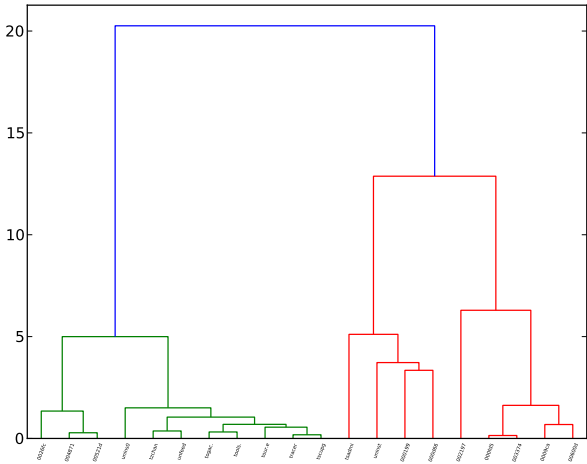
6.1 Distance-Based Phylogenetic Reconstruction

Computing phylogenetic trees has been an active area of research within the malware domain, where the trees are constructed by common techniques motivated from biology [56, 72]. Although maximum-likelihood phylogenetic methods [51] are popular in the biology community [43, 50], their computational complexity poses severe limitations in the malware domain. Given a multiple sequence alignment [36] and a specific phylogenetic tree, the likelihood for each character in the alignment must be computed. With n sequences and s states, the computational complexity of computing the likelihood for the ith state is $\mathcal{O}(s^{n-2})$ [51]. This is expensive yet tractable when considering data such as DNA where there are only 4 states (A,G,T,C), but in the malware domain where the assembly instructions are the states, s is typically in the hundreds. In addition to the computational complexity, there exists non-sequence-based data such as the control flow graph, which is not suited to sequence-based analysis. The control flow graph still contains important information about malware phylogeny, which we would not want to discard. Therefore, I only compare my proposed method to distance-based methods, where the previously defined kernels can be easily adapted.

6.2 Graphical Lasso for Phylogenetic Reconstruction

Most malware phylogeny techniques in the literature rely on bifurcating tree-based approaches [33, 46, 53, 116]. The shortcoming of these methods is that they do not give ancestral relationships to the data. I pose a new problem: attribution-based phylogenetic reconstruction where the goal is to find a graph, $\mathcal{G} = \langle V, E \rangle$. The vertices of the graph, V, are the instances of malware and the directed edges of the graph, E, represent phylogenetic relationships between the data such as "child of" or "parent of".

In Figure 6.2, I show the difference between the output of the two approaches. Both approaches contain valuable information, but I argue that the information on Figure 6.2

(a) Typical Hierarchical Clustering
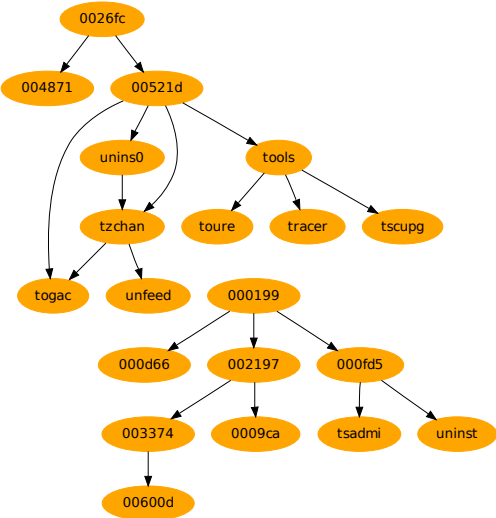
(b) Phylogenetic Reconstruction

Figure 6.2: The difference between the two proposed clustering approaches on the same underlying data set. The typical hierarchical clustering that has been done in the past is shown in (a), and an example of the output of the proposed attribution-based phylogenetic reconstruction method is shown in (b).

(b) is more helpful from a reverse engineering standpoint. The main advantage of the new approach is that the graph from attribution-based phylogenetic reconstruction explicitly states the phylogenetic relationships between instances of malware. For a reverse engineer, knowing that an instance of malware is similar to known malware, say from the same family, rapidly speeds up the reverse engineering process as the analyst will have a general idea of the mechanisms used by the sample. It is more informative to know the set of parents from which a given sample's functionality is derived. Malware having multiple parents is becoming increasingly common [45]. Having malware which is a composite of several other samples is becoming more widespread as malware authors are beginning to use standard software engineering practices, thus making the reuse of code more prevalent.

6.2.1 Overview of Graphical Lasso

The solution I propose is unsupervised and based on the graphical lasso (glasso) [39], which estimates the phylogenetic graph by finding a sparse precision matrix based on the combined kernel matrix. Glasso maximizes the Gaussian log-likelihood with respect to the precision matrix, $\Theta = \Sigma^{-1}$, of the true covariance (kernel) matrix, Σ:

$$\max_{\Theta}\{\log(\det(\Theta)) - \mathrm{tr}(K\Theta) - ||\Theta \circ P||_1\} \tag{6.1}$$

where K is the sample covariance matrix, which in this case means the kernel matrix for a given view of malware. $|| \cdot ||_1$ is the classic L_1 norm used in standard lasso, P is a matrix penalizing specific edges of the precision matrix, and $\circ$ is the Hadamard product. There have been several efficient algorithms developed to solve Equation 6.1 that I take advantage of in this work [10, 39].

By finding the precision matrix with an L_1 penalty, I am seeking a sparse graph that captures the conditional independencies of the true covariance matrix. For instance, if there exists three examples of malware with a direct lineage (x_c is derived from x_b, x_b is derived from x_a), then the naïve approach of creating links between similar examples would create a

completely connected graph between these samples. This would not be unreasonable as they are all similar. The strength of glasso is that it leverages the precision matrix to discover that x_c and x_a are conditionally independent given x_b, which would omit the edge between x_c and x_a.

6.2.2 Modifying Graphical Lasso for Multiple Views

Equation 6.1 solves the problem of finding a Gaussian graphical model for a single view. The obfuscation techniques of malware make this an insufficient solution as individual views can often be unreliable as shown in Chapter 4. Instead, the following multiview problem should be solved:

$$\max_{\Theta,\beta} \left\{ \log(\det(\Theta)) - \text{tr}\left(\sum_{i=1}^{M} (\beta_i K_i)\, \Theta \right) - ||\Theta \circ P||_1 - \lambda ||\beta||_2 \right\} \tag{6.2}$$

Using the linearity of the `trace` function and rearranging terms:

$$\min_{\Theta,\beta} \quad \left\{ \sum_{i=1}^{M} \beta_i \text{tr}(K_i\Theta) - \log(\det(\Theta)) + ||\Theta \circ P||_1 + \lambda ||\beta||_2 \right\} \tag{6.3}$$

$$\text{s.t.} \quad \sum_i \beta_i = 1$$

$$\forall_i \beta_i \geq 0$$

The algorithm I employ is based on alternating projections, first finding the optimal Θ while holding β fixed and then finding the optimal β while holding Θ fixed. As I mentioned previously, there are many efficient algorithms to solve for the optimal Θ [10,39]. To solve for the optimal β assuming a fixed Θ, I first note that $-\log(\det(\Theta)) + ||\Theta \circ P||_1$ is independent of β and can therefore be ignored in the optimization of β leaving us with:

$$\min_{\beta} \left\{ \sum_{i=1}^{M} \beta_i \text{tr}(K_i\Theta) \right\} \tag{6.4}$$

Algorithm 2 Multiple View Graphical Lasso, iteratively finds optimal β and Θ, return Θ.

Require: initial β

$\text{score}_0 \leftarrow \log(\det(\Theta)) - \text{tr}\left(\sum_{i=1}^{M}(\beta_i K_i)\,\Theta\right) - ||\Theta \circ P||_1$

while $\text{score}_t < \text{score}_{t-1}$ **do**

$\quad a_i = \text{tr}(K_i \Theta)$

$\quad \beta \leftarrow \min \ a^T \beta + \frac{1}{2}\beta^T C \beta$

$\qquad \text{s.t.} \quad G\beta \preceq h$

$\quad K \leftarrow \sum_{i=1}^{M} \beta_i K_i$

$\quad \Theta \leftarrow \max_\Theta \{\log(\det(\Theta)) - \text{tr}(K\Theta) - ||\Theta \circ P||_1\}$

$\quad \text{score}_t \leftarrow \log(\det(\Theta)) - \text{tr}\left(\sum_{i=1}^{M}(\beta_i K_i)\,\Theta\right) - ||\Theta \circ P||_1$

**end while**

return precision matrix, Θ

If I let $a_i = \text{tr}(K_i \Theta)$, this problem can be stated as a quadratic program [19] allowing for the use of many efficient algorithms:

$$\min_{\beta} \quad a^T \beta + \frac{1}{2}\beta^T C \beta \qquad\qquad (6.5)$$
$$\text{s.t.} \quad G\beta \preceq h$$

where the simplex constraint forces β to sum to 1 and be non-negative, and $\frac{1}{2}\beta^T C \beta$ removes the incentive of having $\beta_i = 1, \beta_{j\neq i} = 0$. In the case of the degenerate solution, $\beta_i = 1, \beta_{j\neq i} = 0$, this procedure reduces to a feature selector basing all further optimizations on the one best view. Intuitively, a more robust solution would make use of all available information. I defend this intuitive claim in Section 6.2.6. Algorithm 2 outlines the procedure for finding the optimal precision matrix, Θ, when given multiple views of the data.

6.2.3 Leveraging Clusters in Graphical Lasso

While Algorithm 2 solves Equation 6.3, I found that the uniform penalization of the
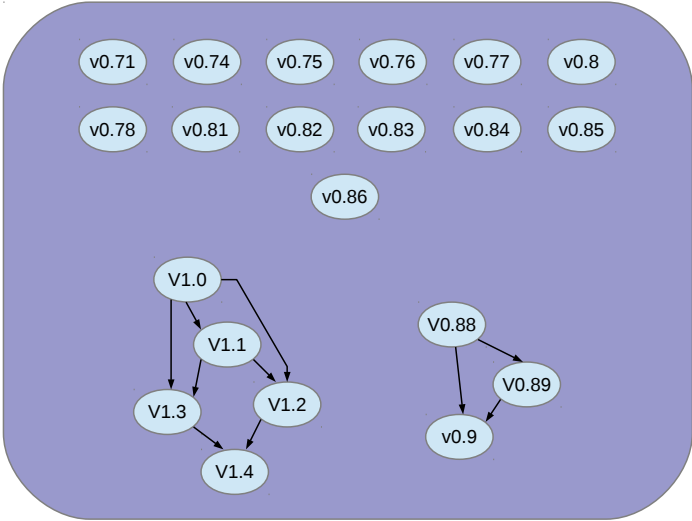
(a) MKL glasso without clustering.
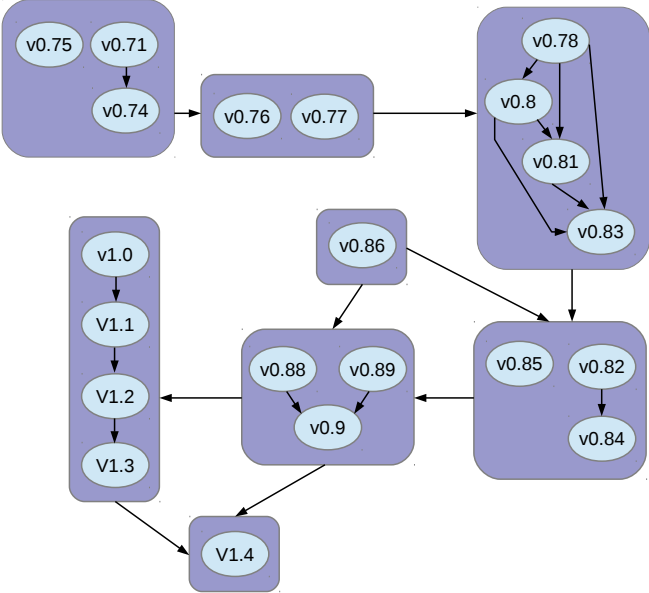
(b) MKL glasso with clustering.

Figure 6.3: NetworkMiner MKL glasso results with and without clustering.

Algorithm 3 Multiple View Graphical Lasso, iteratively finds optimal β and Θ, return Θ. Leverages cluster information to have different penalization throughout the precision matrix

---

Require: initial β

    clusters, $C \leftarrow$ Algorithm 1

    for $c_k \in C$ **do**

        $\rho \leftarrow (\sum_{i,j}^{|c_k|} K_{i,j})/|c_k|^2$

        $\Theta_k \leftarrow$ perform Algorithm 2 with penalization ρ on cluster c_k

    end for

    $K' \leftarrow$ compute inter-cluster similarity matrix

    $\Theta_C \leftarrow$ perform Algorithm 2 with K'

    return precision matrices, Θ_C, Θ_k

L_1-norm led to an interesting phenomenon, namely, the resulting precision matrix was globally sparse with a small subset of the nodes being highly connected. This suggests that a uniform penalty is not appropriate for this problem. Figure 6.3 highlights the advantages of performing a clustering pre-processing step. In Figure 6.3a, many of the different versions of NetworkMiner [80] are unconnected, effectively giving the analyst no information as to their evolution. On the other hand, Figure 6.3b shows how clustering the instances can be a huge advantage. Once similar versions of the program are clustered together, penalizing the L_1-norm uniformly within that cluster becomes more appropriate.

Leveraging clusters in the multiview graphical lasso is straightforward. First, the clusters, $C = c_1, \ldots, c_n$, are found using Algorithm 1 of Chapter 5. Then, the multiview graphical lasso (Algorithm 2) is applied to each cluster, adjusting the penalty term appropriately. To adjust the penalty, I used a heuristic, $(\sum_{i,j}^{|c_k|} K_{i,j})/|c_k|^2$, which effectively penalizes self-similar clusters heavily, allowing those clusters to be more sparse. Finally, I create a new set of similarity matrices, but instead of measuring the similarity between different instances in the dataset, this similarity matrix measures similarity between the different clusters. This matrix is simply created by taking the average similarity between every $x \in c_i$ and $y \in$

c_j. Performing the multiview glasso on these matrices finds the conditional independences between the different clusters. Algorithm 3 details the procedure.

6.2.4   Forcing Directionality

The major drawback to the graphical lasso approach is that it does not find directed graphs, but rather finds an undirected Gaussian graphical model. Because of this, it cannot identify parent/child relationships, meaning that when a link between x_a and x_b is found, there is no information as to whether x_b is the child of x_a or vice-versa. I have experimented with several heuristics to determine the directionality of the edges in the Gaussian graphical model.

The first method, and seemingly the most accurate, is contained within the portable executable (PE) header information. Here, there is a 32-bit value that contains the time and date the PE file was compiled as seconds since January 1st, 1970 00:00:00. Unfortunately, malware authors often obfuscate this field, either setting it to all 0's, or even worse, setting it to a seemingly legitimate, but inaccurate date. As a reference point for the prevelance of this type of defense, the Bagle malware dataset had 9 out of 25 samples where this field was obfuscated.

In the case where directionality cannot be reliably inferred from the PE header timestamp, I fallback to using entropy, under the assumption that as programs evolve, they become more complex. This has been shown to be a reasonable metric in the software engineering literature [71]. While this is undoubtedly not correct for all programs, it has proven to be a somewhat reliable last resort. In addition, it seems to be a more robust measure of a program's evolution compared to the intuitive measure of a program's size. For both Algorithms 2 and 3, this method of forcing directionality can be considered a post-processing step on the precision matrices, Θ.
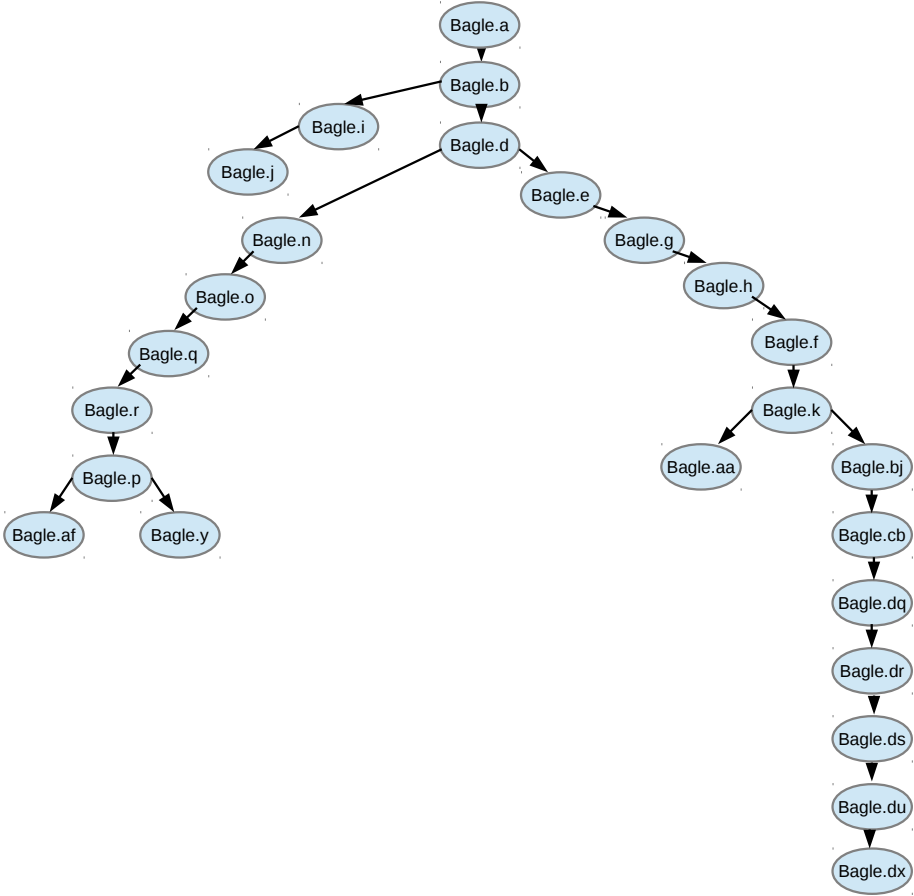
Figure 6.4: The ground truth phylogeny for the bagle worm.

6.2.5 Data

To be able to accurately quantify my results, I must have access to ground truth, more precisely, the true phylogenetic graph of the program. As one can imagine, this is a very difficult, time-consuming process. Fortunately, it is made easier for some benign programs as their subversion or github repositories can be used to gather this information. Unfortunately, malware authors do not typically use these tools in an open setting, making the phylogenetic graphs of malware much more difficult to obtain. To understand the evolution of a malware family, I elicited the help of several experts at Los Alamos National Laboratory that are a part of the computer security incident response team. These experts used several sources of

Dataset	Variants
Mytob	a, b, c, f, j, n, t, v, x, ar, au, aw, bf, bi, ef, eg, ej, jb, ja, jd
Koobface	a, b, d, e, g, h, k, q, t, y, z, ab, ad, af, ai, ak, ba, bc, bn
Bagle	a, b, d, e, f, g, h, i, j, k, n, o, p, q, r, y, aa, af, bj, cb, dq, dr, ds, du, dx
NetworkMiner	.71, .74, .75, .76, .77, .78, .8, .81, .82, .83, .84, .85, .86, .88, .89, .9, 1.0, 1.1, 1.2, 1.3, 1.4
Mineserver	05e8b7ae002496a810c68dda2d9783a9e3c2a013
	0e290772a2e387be832d587596f983a4cf4c8b1f
	20c721a331d411b193c20ab917b877cc50756dea
	306bf349f9a4297b2cee4373f6b38c5b2dc18c6b
	3761930a9cc5e9a2e2fd7d7ac0ab8bc961ff6f64
	387585fd85761ddab76032c86cf425c452a23bc5
	6bf36f39020acd3b4765ac7d6e98b77e2be0c33f
	7120ec0443c43a722c68c833405742350d17d93a
	97bab2ea91a303de3b1545d388fae576300acd47
	a6ac3e47b3f4b2ad22b56b5f21effc73838f9538
	bc932c828f94406c46db979d01a09f3e55e16fad
	d9088e55d618954f0f467cb1e5d60d7c171e50ad
	f694a73d1dc360c008bf8ccefa1b8afbf75cdc3e

Table 6.1: The variants used for each family. For the malicious families, the variants are labels a → z, aa → az, etc. in order of first detection in the wild. NetworkMiner uses the official version numbers supplied by the authors, and Mineserver uses the sha1, which is searchable from the github repository.

information to come up with an informed graph depicting the evolution of malware, such as the time the malware was first seen in the wild, the compile-time timestamp, the functionality of the sample, and the obfuscation methods employed by the sample. While the phylogenetic graphs the domain experts have manually found are by no means 100% accurate, they do provide a reasonable baseline in a setting where ground truth is not available.

I use three malicious programs: Mytob, Koobface, and Bagle. These samples were obtained from VX Heaven [115]. The specific variants used are listed in Table 6.1. Mytob is a mass-mailing worm that seeks a user's address book to then send itself to all of that user's contacts [111]. Several variations also have the ability to exploit network vulnerabilities to

spread without the need of its internal SMTP engine. The Koobface worm spreads through social networking sites with the intent of installing software for a botnet [110]. This worm updates a user's status with a link to some type of external media content, that is actually another copy of the worm. Bagle is another mass-mailing worm, that creates a botnet [109]. It has the ability to download external content, as well as spread through file-sharing networks, such as Kazaa, by placing a copy of itself in the "share" folder. The expert derived ground truth phylogeny of the bagle worm is shown in Figure 6.4, with all ground truth phylogenies being shown in Appendix B.

In addition to the malicious programs, I validate my methods on two benign programs, NetworkMiner and Mineserver, due to the ground truth for these graphs being more readily available. NetworkMiner is a network forensics anaylsis tool specializing in packet sniffing and parsing PCAP files [80]. Mineserver is a way to host worlds in the popular Minecraft game [78].

6.2.6 Results

Table 6.3 lists all the results for the 5 datasets previously described with respect to precision, recall, and F-norm. Figure 6.5 shows a hypothetical ground truth graph with a hypothetical reconstructed graph. In this example, the precision would be 2/4 as 2 of the 4 reconstructed edges are true positives with respect to the ground truth, and the recall would be 2/3 as the reconstructed graph contains 2 out of the 3 true edges with respect to the ground truth. The Frobenius norm, or F-norm, between 2 matrices is defined as:

$$||A - B||_F = \sqrt{\sum_i \sum_j (A_{ij} - B_{ij})^2} \tag{6.6}$$

which I compute by taking the Frobenius norm between the reconstructed graph and the ground truth graph. I compare my multiview glasso + clustering approach to my multiview glasso approach and regular glasso with and without the clustering preprocessor for both
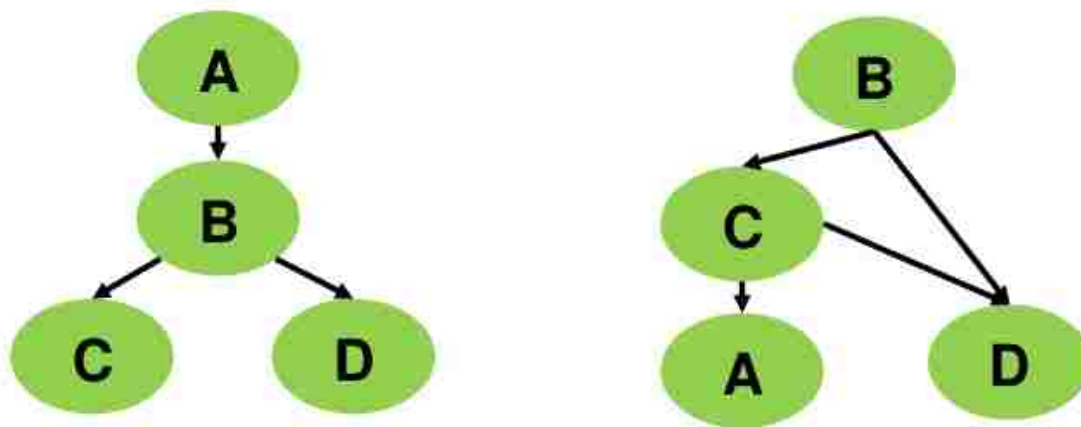
Figure 6.5: On the left is a hypothetical ground truth graph and on the right is a hypothetical reconstructed graph. In this example the precision would 2/4 and the recall would be 2/3.

the best single view and a uniform combination of views. I also compare my approach to the Gupta algorithm [45] and a simple baseline, the minimum spanning tree.

As Table 6.3 demonstrates, the proposed method performs well on a variety of datasets, both malicious and benign. The Gupta algorithm performs well with respect to precision, and even out-performs my algorithm on the mineserver dataset, but this is mainly because it finds sparser graphs, where precision will naturally be higher. The mineserver dataset is interesting for two reasons: the ground truth is known with absolute certainty, and merges and branches are present. Both of these cases are present in most real-word software engineering projects including malware. Figure 6.6 shows the ground truth as well as the graph acquired with the multiview glasso + clustering method. As the figure demonstrates, I was able to recover the majority of the branches and merges, and can recover most of the evolutionary flow of the program.

The cophenetic correlation coefficient is a measure of how well a hierarchical clustering algorithm models the original distances. In this setting, the distances are based on the

	Hierarchical Clustering	MKLGlasso+Clust
networkminer	.3749	**.5361**
mineserver	.4432	**.6886**
bagle	.1167	**.1733**
mytob	.1098	**.3808**
koobface	.1424	**.2463**

Table 6.2: Cophenetic correlation coefficients comparing hierarchical clustering to my proposed approach.

translated kernel values, between the data points [98], and is given by:

$$c = \frac{\sum_{i<j}(x(i,j) - x)(t(i,j) - t)}{\sqrt{\left[\sum_{i<j}(x(i,j) - x)^2\right]\left[\sum_{i<j}(t(i,j) - t)^2\right]}} \tag{6.7}$$

In Equation 6.7, $t(i,j)$ is the distance between i and j by the model found, t is the average distance between all instances found by the model, $x(i,j)$ is the ground truth distance between i and j, and x is an average of the ground truth distance between all instances.

Table 6.2 compares the cophenetic correlation coefficient between the MKL glasso + clustering method I have presented and the UPGMA hierarchical clustering algorithm [125] that has been typically used in this domain [56,72]. This table was included for completeness to compare to hierarchical clustering methods, and these results should not be taken out of context. While my method outperforms hierarchical clustering, this is mainly because it is not restricted to dendrogram structures (i.e. bifurcating trees) [125], but rather have a much more expressive model space.

Dataset	Method	F-norm	Precision	Recall
NetworkMiner	MKLGlasso+Clust	**4.5826**	**.4857**	**.85**
	MKLGlasso	5.5678	.3514	.65
	Glasso-Best View	6.0	.2895	.55
	Glasso-Best View+Clust	5.3852	.3902	.80
	Glasso-Uniform Comb	6.1644	.3043	.70
	Glasso-Uniform Comb+Clust	5.0	.4360	**.85**
	Gupta	5.0	.3810	.40
	Minimum Spanning Tree	5.6569	.35	.70
MineServer	MKLGlasso+Clust	**4.0**	0.7222	**0.8125**
	MKLGlasso	5.4772	0.5833	0.2188
	Glasso-Best View	5.8242	0.4118	0.1935
	Glasso-Best View+Clust	4.8134	0.4510	0.3871
	Glasso-Uniform Comb	5.6711	0.4314	0.1875
	Glasso-Uniform Comb+Clust	4.4655	0.4902	0.4194
	Gupta	4.7958	**0.8462**	0.3438
	Minimum Spanning Tree	7.4833	0.0	0.0
Bagle	MKLGlasso+Clust	5.7446	**0.20**	**0.3333**
	MKLGlasso	9.5394	0.0964	0.125
	Glasso-Best View	10.7731	0.0704	0.1176
	Glasso-Best View+Clust	**5.5813**	0.1480	.0909
	Glasso-Uniform Comb	10.2921	0.0812	0.0980
	Glasso-Uniform Comb+Clust	9.6476	.1351	.1220
	Gupta	6.5574	0.12	0.125
	Minimum Spanning Tree	8.3667	0.0208	0.0417
Mytob	MKLGlasso+Clust	7.9373	**0.1563**	**0.5263**
	MKLGlasso	8.5348	0.0988	0.2258
	Glasso-Best View	8.7388	0.0864	0.1935
	Glasso-Best View+Clust	8.2184	0.1282	0.2903
	Glasso-Uniform Comb	10.2766	0.0617	0.1951
	Glasso-Uniform Comb+Clust	8.8117	0.1081	0.2683
	Gupta	**6.0828**	0.05	0.0526
	Minimum Spanning Tree	7.2801	0.0526	0.1053
Koobface	MKLGlasso+Clust	**5.2915**	0.5812	**0.5**
	MKLGlasso	5.3852	0.2917	0.3889
	Glasso-Best View	6.8551	0.2391	0.3171
	Glasso-Best View+Clust	6.0427	0.2821	0.3235
	Glasso-Uniform Comb	6.6043	0.2195	0.2927
	Glasso-Uniform Comb+Clust	5.9486	0.3023	0.3636
	Gupta	5.9161	0.3158	0.3333
	Minimum Spanning Tree	7.2111	0.0278	0.0556

Table 6.3: Phylogenetic graph reconstruction results in terms of F-norm, Precision, and Recall.
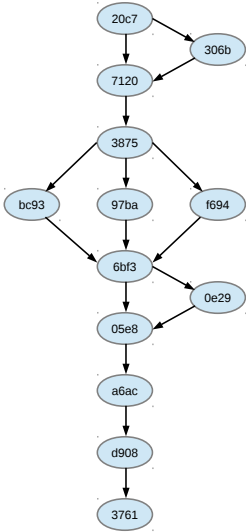
(a) Ground truth
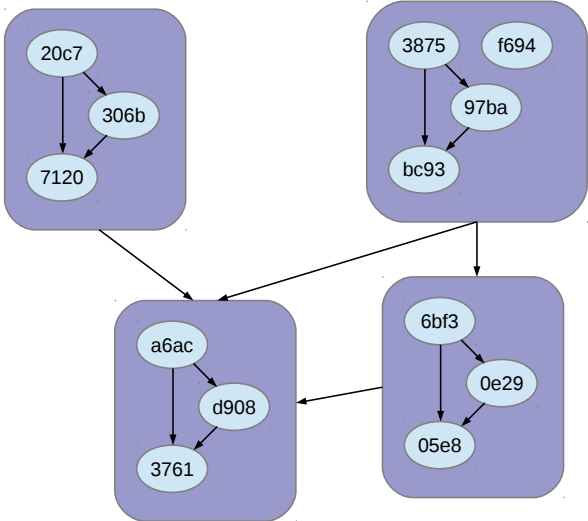
(b) MKL Glasso+Clustering

Figure 6.6: Comparison between mineserver ground truth and the network identified by Algorithm 3

Chapter 7

Conclusions

The primary thesis of this dissertation is that incorporating multiple views of malware provides a significant benefit compared to looking at only a single view for classification, clustering, and phylogenetic reconstruction. Multiple views of data can be used to increase the performance of machine learning algorithms for a variety of fields, but it is particularly important in the malware context because any single view is susceptible to a fixed obfuscation strategy. While it is easy to make a single malware view unreliable, it is much harder to obfuscate against *all* views simultaneously. I have shown that multiple views do indeed improve performance for a variety of tasks including classification, clustering, and phylogenetic graph reconstruction. To enable this analysis, I also developed a number of machine learning algorithms that can exploit multiple data views.

7.1 Discussion

Finding a data representation that worked well with the sequence-based data derived from the malware was the first step of this research. In Chapter 3 I presented the Markov chain data representation. I showed how to define a kernel, or similarity function, on these Markov

chains that can capture similarity on both the local and global structure of the Markov chains. I modeled the dynamic instruction traces as Markov chains and the results presented demonstrated that this representation was superior to that of the n-gram representation, which had been extensively used in the malware literature.

Once I established the data representation, I then showed the importance of combining multiple views of the malware samples. The first problem I approached was that of classification. Fortunately, there is a rich history of multiple view learning in the machine learning literature. I was able to apply existing multiple kernel learning algorithms to solve the multiview classification problem for malware, and I presented results that supported my hypothesis that combining multiple views of malware would increase classification accuracy.

When analyzing malware, a simple yes/no answer is a good start, but knowing the relationships between a new sample and previously seen samples is much more informative. The ability to group malware samples into similar families has the potential to drastically reduce the time to reverse engineer samples. For instance, members from the computer security incidence response team at Los Alamos National Laboratory can spend several weeks analyzing a new sample of malware. But, they can analyze a sample from a known family, in which they have had previous experience reverse engineering, within several hours to several days. I investigated this clustering problem in Chapter 5. Unlike the classification problem, I had to develop novel machine learning algorithms to solve the multiple view clustering problem. Earlier work required a priori information about the weights of the views, which can be time-consuming to gather in the malware domain. These weights could also drastically change between datasets. I showed that my multiview clustering approach is superior to previous approaches used in the malware domain. I also showed that these methods outperform competing methods on several machine learning benchmark datasets.

Finally, I presented my work on malware phylogenetics in Chapter 6. This line of research will both help reverse engineers understand the evolution of malware they are currently studying and help them attribute malware instances to known authors. I created a novel

extension to graphical lasso to allow phylogenetic analysis from multiple views with no a priori information on the importance of the individual views. Again I showed that incorporating multiple views of the data increased performance of the system on several datasets, both benign and malicious.

There is a constant "arms race" between the people developing malicious code and those detecting and understanding it. In this dissertation, I have introduced the multiview framework for several important analysis problems, while developing novel machine learning algorithms, applicable to fields outside of malware, when appropriate. I have shown how the methods of this dissertation have succeeded despite the constant use of advanced obfuscation techniques within the malware datasets. Moving toward a multiview analysis framework is one way to raise the bar for malware authors, forcing them to devote more time and resources into developing solutions that avoid detection.

7.2   Future Work

A theme throughout this dissertation was that of helping malware analysts do their job more effectively. They are under severe time constraints to understand the malware's functionality and respond appropriately. The tools developed in the previous chapters will give the analysts a significant head start, but there are future research avenues that could provide further help.

Understanding the functions of a given piece of malware is a tedious process that can last anywhere from a day to months, depending on the malware's complexity. This process usually requires the analyst to hand label different sections of the function call graph and to slowly connect the "dots" into a picture of the malware's logic. This provides an exciting area to apply and extend the methods described in this dissertation.

This problem can be posed as a multi-class classification problem, where the labels would

be the function the subroutine performs (e.g. data exfiltration), instead of the normal labeling of benign vs malware. Two problems arise in this formulation: a data problem and a methodology problem. First, getting a labeled dataset would require hundreds, or even thousands, of hours from a skilled reverse engineer. And second, the Markov chain data representation may have to be adjusted to account for a significantly shorter sequence of instructions. For instance, the program instruction traces used throughout this dissertation typically had several million instructions, whereas functions within a program typically have several hundred.

In addition to new science directions, there are some interesting engineering applications for the research already completed. It would be interesting to implement the phylogenetic graph reconstruction presented in Chapter 6 within a system that classifies malware. This would allow for a much deeper understanding of the network's threat landscape, and would allow the analysts tasked with attribution to do their jobs more efficiently.

Appendices

Appendix A

Multiple Kernel Learning Clustering Proofs

A.1   Finding A_i for the Unnormalized Laplacian

The multiple views version of the unnormalized Laplacian is defined as:

$$L_{un}(\beta) = \sum_i^M \beta_i D_i - \sum_i^M \beta_i K_i$$

Plugging the unnormalized Laplacian into Equation 5.8:

$$\text{tr}\left(U^T \left(\sum_i^M \beta_i D_i - \sum_i^M \beta_i K_i\right) U\right)$$

By rearranging terms:

$$= \text{tr}\left(U^T \left(\sum_i^M \beta_i (D_i - K_i)\right) U\right)$$

$$= \text{tr}\left(\sum_i^M \beta_i U^T (D_i - K_i) U\right)$$

Using the linearity of the trace function:

$$= \text{tr}\left(\beta_0 U^T (D_0 - K_0)U\right) + \cdots + \text{tr}\left(\beta_M U^T (D_M - K_M)U\right)$$

Finally, define $A_i = U^T (D_i - K_i) U$.

A.2 Finding A_i for the Normalized Laplacian

The multiple views version of the normalized Laplacian is defined as:

$$L_{norm}(\beta) = I - \left(\sum_i^M \beta_i D_i\right)^{-1} \left(\sum_i^M \beta_i K_i\right)$$

Again, plugging the normalized Laplacian into Equation 5.8:

$$\text{tr}\left(U^T \left(I - \left(\sum_i^M \beta_i D_i\right)^{-1} \left(\sum_i^M \beta_i K_i\right)\right) U\right)$$

$$= \text{tr}(U^T I U) - \text{tr}\left(U^T \left(\sum_i^M \beta_i D_i\right)^{-1} \left(\sum_i^M \beta_i K_i\right) U\right)$$

Note that $\text{tr}(U^T I U) = \gamma$ is a constant and rearrange terms:

$$= \gamma - \text{tr}\left(U^T \left(\sum_i^M \beta_i K_i \left(\sum_j^M \beta_j D_j\right)^{-1}\right) U\right) \tag{A.1}$$

By commuting the negative sign to inside the trace function, A_i for the normalized case is now defined as:

$$A_i = U^T \left(-K_i \left(\sum_j^M \beta_j D_j\right)^{-1}\right) U$$

Appendix B
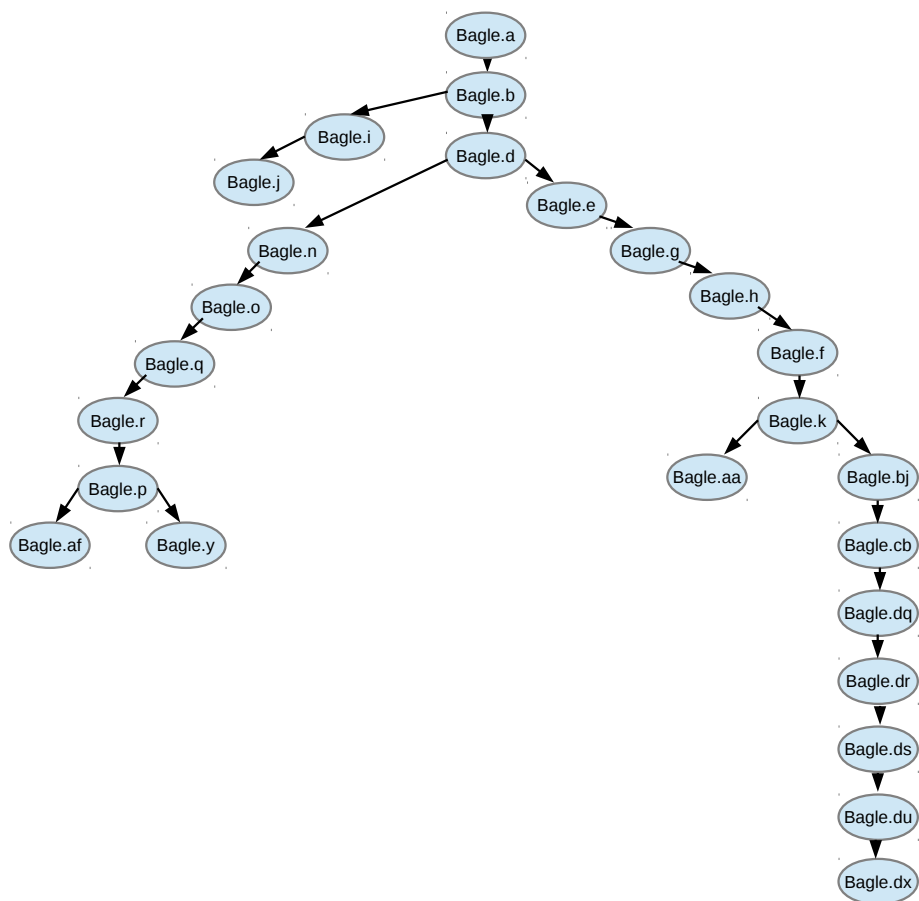
Ground Truth Phylogenetic Networks

Figure B.1: The ground truth phylogeny for the bagle worm.

B.1   Bagle

Bagle is a mass-mailing worm, that also creates a botnet [109]. It has the ability to download external content, as well as spread through file-sharing networks, such as Kazaa, by placing a copy of itself in the "share" folder. The specific variants used are highlighted in Table 6.1.
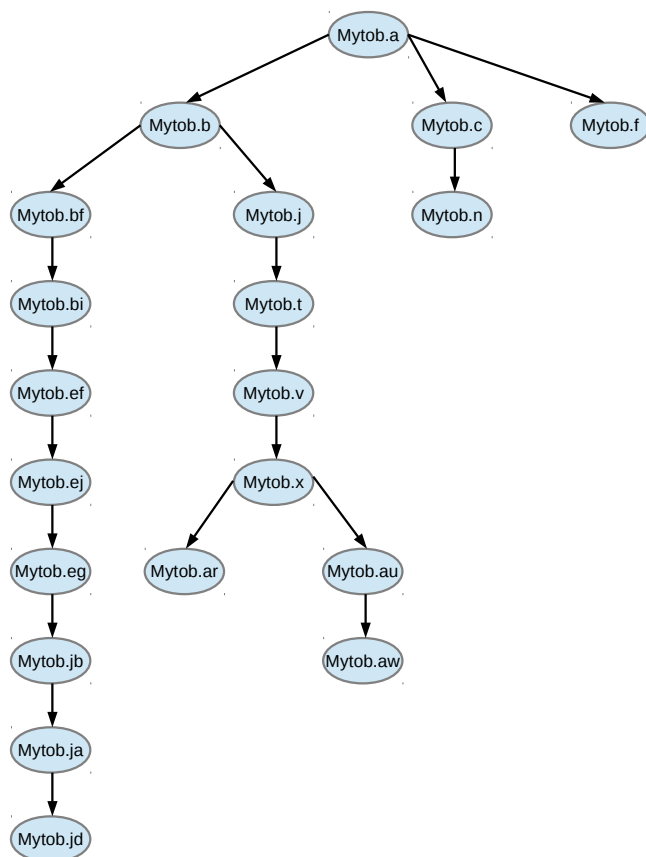
Figure B.2: The ground truth phylogeny for the mytob worm.

B.2 Mytob

Mytob is a mass-mailing worm that seeks a user's address book to then send itself to all of that user's contacts [111]. Several variations also have the ability to exploit network vulnerabilities to spread without the need of it's internal SMTP engine. The specific variants used are highlighted in Table 6.1.
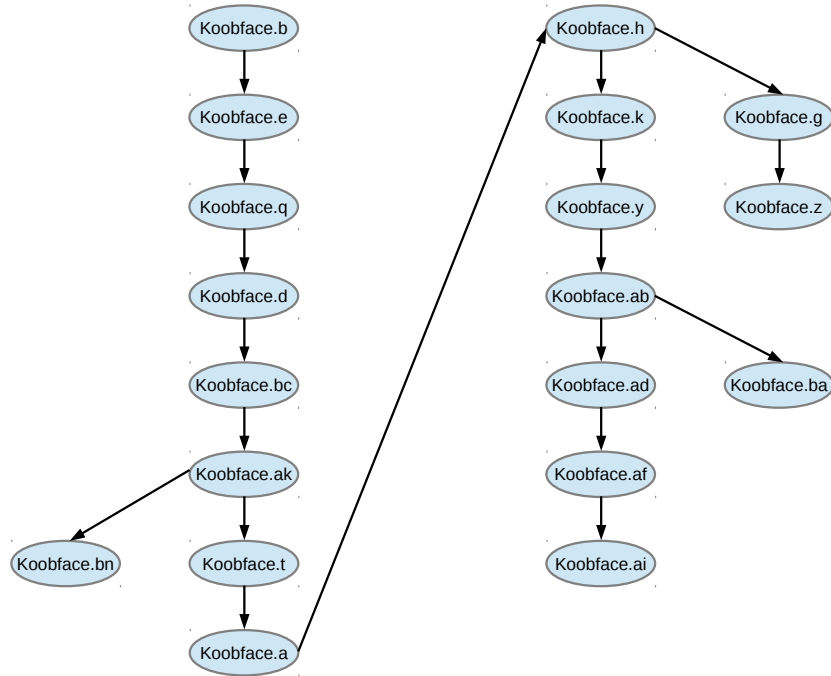
Figure B.3: The ground truth phylogeny for the koobface worm.

B.3   Koobface

The Koobface worm spreads through social networking sites with the intent of installing software for a botnet [110]. This worm updates a user's status with a link to some type of external media content, that is actually another copy of the worm. The specific variants used are highlighted in Table 6.1.
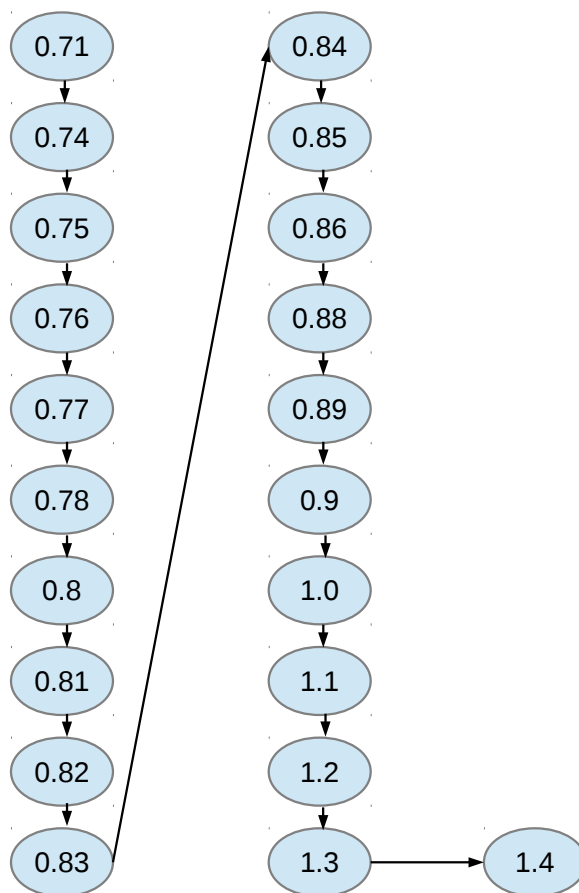
Figure B.4: The ground truth phylogeny for the Networkminer program.

B.4 NetworkMiner

NetworkMiner is a network forensics anaylsis tool specializing in packet sniffing and parsing PCAP files. The NetworkMinder ground truth phylogeny was taken from its public subversion repository [80]. The specific variants used are highlighted in Table 6.1.
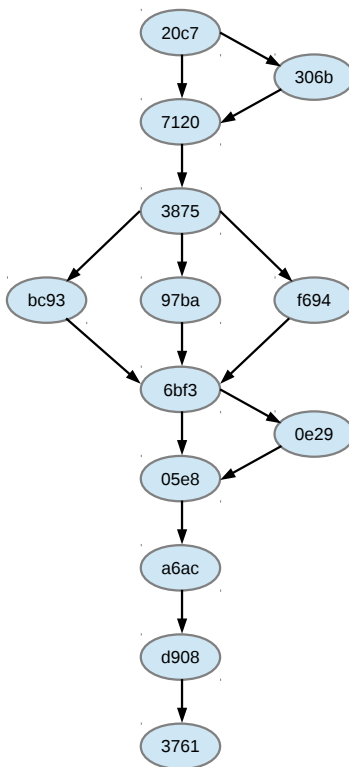
Figure B.5: The ground truth phylogeny for the mineserver program.

B.5 Mineserver

Mineserver is a way to host worlds in the popular Minecraft game. The Mineserver ground truth was sampled from its github repository [78]. The specific variants used are highlighted in Table 6.1.

Appendix C

Computational Complexity of the Methods

C.1   Complexity of the SVM

The computational complexity of the support vector machine needs to broken down into two phases: training and testing. To begin the training, a kernel must first be defined. The kernels that I have used are explained in Section 3.2. In the case of the Markov chain data representation with n samples and a node set V, computing the Gaussian kernel (Equation 3.2) is $\mathcal{O}(n^2 \cdot |V|^2)$. For the spectral kernel (Equation 3.4) computing the top-k eigenvectors of Laplacian has $\mathcal{O}(k \cdot |V|^2)$ complexity bringing the total computational complexity of this kernel to $\mathcal{O}(n^2 k \cdot |V| + k \cdot |V|^2)$. Training of the support vector machine can be in $\mathcal{O}(n \cdot |\alpha|)$ where $|\alpha|$ is the number of support vectors found [85] bringing the total computational complexity for the training of the combined kernel of Chapter 3 to $\mathcal{O}(n^2 \cdot |V|^2 + n^2 k \cdot |V| + k \cdot |V|^2 + n \cdot |\alpha|)$.

Given a new sample, a kernel function, and the support vectors (α) given from the training step, the sample can be classified with Equation 3.11. This function has computational

complexity of $\mathcal{O}(|\alpha| \cdot (k \cdot |V|^2 + |V|^2))$ for the combined kernel approach of Chapter 3. Empirical timings for these methods are given in Table 3.3

C.2   Complexity of the Multiview SVM

The computational complexity for the multiple kernel learning support vector machine is composed of two parts: one to calculate the β vector and one for the α vector. Computing α given β has the same complexity of normal SVM training and is explained in Appendix C.1. Computing β has been shown to have complexity $\mathcal{O}(|\alpha_t|^3 \cdot M|)$ where $|\alpha_t|$ is the number of non-zero support vectors at iteration t and M is the number of views [88]. Furthermore, this algorithm has been shown to converge in $\mathcal{O}(\log(M)/\epsilon^2)$ iterations where ϵ is related to the stopping criteria (see Equation 4.11) [101]. Therefore, the training complexity for M views with a Gaussian kernel on the Markov chains would be $\mathcal{O}(\sum_i^M (n^2 \cdot |V_i|^2) + \log(M)/\epsilon^2 \cdot (|\alpha_t|^3 \cdot M|))$

The testing phase of the multiview SVM is the same as the single SVM with more kernels and the complexity for a Gaussian kernel on each view is $\mathcal{O}(|\alpha| \cdot (\sum_i^M |V_i|^2))$. Empirical timing results for the Multiview SVM can be found in Table 4.5 and Figure 4.7.

C.3   Complexity of the Multiview Clustering

The multiview clustering algorithm of Chapter 5 is an iterative algorithm, and while there is no formal convergence proof, I will note that the algorithm converged within 10 iterations in all of the experiments. The algorithm is composed of two main parts: computing the spectral clustering objective function to get U (Equation 5.8) and computing a semidefinite program to find β (Equation 5.9). Computing Equation 5.8 can be done in $\mathcal{O}(n^3)$ time where n is the number of samples [74]. Equation 5.9 has a computational complexity of $\mathcal{O}(n^6)$ in

the worst case but has been shown to be $\mathcal{O}(n^3)$ in the average case [17].

Empirical times to compute the optimization for the multiview clustering algorithms based on the normalized and unnormalized Laplacian are presented in Table 5.4.

C.4 Complexity of the Multiview Graphical Lasso

Similar to the multiview clustering algorithm, the multiview graphical lasso algorithm is an iterative algorithm with two main components. First, graphical lasso is solved given some β vector specifying the weights of each view. Graphical lasso has a computational complexity of $\mathcal{O}(n^3)$ where n is the number of samples [119]. Next, β is updated with a quadratic program, which has a computational complexity $\mathcal{O}(n^3)$ where n is again the number of samples [120]. This algorithm converged within 15 iterations for all datasets presented in Chapter 6.

Appendix D

Gaining Access to the Code/Data

Access to the code/data referenced herein is restricted. If you are an employee or a contractor to the U.S. Government and are a U.S. citizen, you may be able to obtain access to the data. Furthermore, Industrial and University partners with an active CRADA agreement with LANS, LLC and U.S. citizenship may also be able to obtain access to the data. To do so, please send an email to cyber@lanl.gov.

In Chapter 3, the support vector machine was trained using PyML [13]. For the results of Chapter 3, I compared against 9 leading antivirus products: BitDefender [18], Kaspersky [58], Avira [8], F-Secure [40], F-prot [37], Symantec [108], ClamAV [27], McAfee [77], and AVG [7]. These results were gathered in September 2010 with the most current signature databases.

In Chapter 4, the multiple kernel support vector machine was trained using the Shogun library [102]. For these results, I compared against 11 leading antivirus products: BitDefender [18], Kaspersky [58], Avira [8], F-Secure [40], F-prot [37], AVG [7], ClamAV [27], Sophos [104], Symantec [108], McAfee [77], and Avast [6]. These results were gathered in September 2011 with the most current signature databases.

Appendix D. Gaining Access to the Code/Data

The semidefinite program in Chapter 5 and the quadratic program in Chapter 6 were both solved for by using the CVXOPT python software for convex optimization [30]. The graphical lasso optimization problem of Chapter 6 was solved by using the `glasso` R package [42].

The malware from Chapters 3 and 4 were in part supplied by the Offensive Computing malware repository [83], with other malware samples and benign samples being supplied by a LANS, LLC CRADA partner. The malware from Chapter 5 was supplied by a LANS, LLC CRADA partner. In Chapter 6, the three malicious families, i.e. Mytob, Koobface, and Bagle, were obtained from VX Heaven [115]. The specific variants used are listed in Table 6.1. The benign families, i.e. NetworkMiner and Mineserver, are available to download from sourceforge for NetworkMiner [80] and github for Mineserver [78]. The variants used are listed in Table 6.1.

References

[1] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane, *Graph-Based Malware Detection using Dynamic Analysis*, Journal of Computer Virology (2011), 1–12.

[2] Blake Anderson, Curtis Storlie, and Terran Lane, *Improving Malware Classification: Bridging the Static/Dynamic Gap*, Proceedings of the Fifth ACM Workshop on Security and Artificial Intelligence, 2012, pp. 3–14.

[3] _____, *Multiple Kernel Learning Clustering with an Application to Malware*, IEEE Twelfth International Conference on Data Mining, 2012, pp. 804–809.

[4] *ANUBIS: A Platform for the Dynamic Analysis of Malware*, Accessed 17 September 2013. `http://anubis.iseclab.org/`.

[5] *ASPack Software*, Accessed 17 September 2013. `http://www.aspack.com/asprotect.html`.

[6] *Avast*, Accessed 27 February 2014. `http://www.avast.com/en-us/index`.

[7] *AVG*, Accessed 27 February 2014. `http://www.avg.com/us-en/homepage`.

[8] *Avira*, Accessed 27 February 2014. `http://www.avira.com/en/index`.

[9] Francis R. Bach, Gert R. G. Lanckriet, and Michael I. Jordan, *Multiple Kernel Learning, Conic Duality, and the SMO Algorithm*, Proceedings of the Twenty-First International Conference on Machine Learning, 2004.

REFERENCES

[10] Onureena Banerjee, Laurent El Ghaoui, and Alexandre d'Aspremont, *Model Selection Through Sparse Maximum Likelihood Estimation for Multivariate Gaussian or Binary Data*, Journal of Machine Learning Research **9** (2008), 485–516.

[11] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda, *Scalable, Behavior-Based Malware Clustering*, ISOC Network and Distributed System Security Symposium, 2009.

[12] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda, *Dynamic Analysis of Malicious Code*, Journal of Computer Virology **2** (2006), 67–77.

[13] Asa Ben-Hur, *Pyml: Machine Learning in Python*, Accessed 17 September 2013. `http://pyml.sourceforge.net/`.

[14] Steffen Bickel and Tobias Scheffer, *Multi-View Clustering*, Proceedings of the Fourth IEEE International Conference on Data Mining, 2004, pp. 19–26.

[15] Daniel Bilar, *Opcodes as Predictor for Malware*, International Journal of Electronic Security and Digital Forensics **1** (2007), 156–168.

[16] Christopher M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] Pratik Biswas, Tzu-Chen Lian, Ta-Chung Wang, and Yinyu Ye, *Semidefinite Programming Based Algorithms for Sensor Network Localization*, ACM Transactions on Sensor Networks (TOSN) **2** (2006), no. 2, 188–220.

[18] *BitDefender*, Accessed 27 February 2014. `http://www.bitdefender.com/`.

[19] Stephen Boyd and Lieven Vandenberghe, *Convex Optimization*, Cambridge University Press, New York, NY, USA, 2004.

[20] Andrew P Bradley, *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*, Pattern Recognition **30** (1997), no. 7, 1145–1159.

REFERENCES

[21] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga, *Detecting Self-Mutating Malware using Control-Flow Graph Matching*, Detection of Intrusions and Malware and Vulnerability Assessment, 2006, pp. 129–143.

[22] Christopher J. C. Burges, *A Tutorial on Support Vector Machines for Pattern Recognition*, Data Mining and Knowledge Discovery **2** (1998), 121–167.

[23] Claire Cardie and Nicholas Nowe, *Improving Minority Class Prediction using Case-Specific Feature Weights*, Proceedings of the Fourteenth International Conference on Machine Learning, 1997, pp. 57–65.

[24] Baudouin Le Charlier, Abdelaziz Mounji, Morton Swimmer, and Fachbereich Informatik, *Dynamic Detection and Classification of Computer Viruses using General Behaviour Patterns*, Proceedings of the Virus Bulletin Conference (1995).

[25] Mihai Christodorescu and Somesh Jha, *Static Analysis of Executables to Detect Malicious Patterns*, Proceedings of the Twelfth USENIX Security Symposium, 2003, pp. 169–186.

[26] Fan R. K. Chung, *Spectral Graph Theory (CBMS Regional Conference Series in Mathematics, No. 92)*, American Mathematical Society, 1997.

[27] *ClamAV*, Accessed 27 February 2014. `http://www.clamav.net/lang/en/`.

[28] Jedidiah R Crandall, Roya Ensafi, Stephanie Forrest, Joshua Ladau, and Bilal Shebaro, *The Ecology of Malware*, Proceedings of the Workshop on New Security Paradigms, 2009, pp. 99–106.

[29] Jedidiah R Crandall, S Felix Wu, and Frederic T Chong, *Minos: Architectural Support for Protecting Control Data*, ACM Transactions on Architecture and Code Optimization (TACO) **3** (2006), no. 4, 359–389.

[30] Joachim Dahl and Lieven Vandenberghe, *Cvxopt: Python Software for Convex Optimization*, Setembro, 2009.

[31] Jianyong Dai, Ratan Guha, and Joohan Lee, *Efficient Virus Detection Using Dynamic Instruction Sequences*, Journal of Computers **4** (2009), no. 5.

REFERENCES

[32] P. Danaher, P. Wang, and D. M. Witten, *The Joint Graphical Lasso for Inverse Covariance Estimation Across Multiple Classes*, ArXiv e-prints (November 2011), available at `1111.0324`.

[33] Craig Darmetko, Steven Jilcott, and John Everett, *Inferring Accurate Histories of Malware Evolution from Structural Evidence*, The Twenty-Sixth International FLAIRS Conference, 2013.

[34] *DDoS Attacks: The Zemra Bot*, Accessed 4 February 2014. `http://www.symantec.com/connect/blogs/ddos-attacks-zemra-bot`.

[35] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee, *Ether: Malware Analysis Via Hardware Virtualization Extensions*, Proceedings of the Fifteenth ACM Conference on Computer and Communications Security, 2008, pp. 51–62.

[36] Robert C Edgar and Serafim Batzoglou, *Multiple Sequence Alignment*, Current Opinion in Structural Biology **16** (2006), no. 3, 368 –373.

[37] *F-Prot*, Accessed 27 February 2014. `http://www.f-prot.com/`.

[38] A. Frank and A. Asuncion, *UCI Machine Learning Repository* (2010).

[39] Jerome Friedman, Trevor Hastie, and Robert Tibshirani, *Sparse Inverse Covariance Estimation with the Graphical Lasso*, Biostatistics **9** (2008), no. 3, 432–441.

[40] *F-Secure*, Accessed 27 February 2014. `http://www.f-secure.com/en/web/home_us/home`.

[41] Debin Gao, Michael Reiter, and Dawn Song, *BinHunt: Automatically Finding Semantic Differences in Binary Programs*, Information and Communications Security, 2008, pp. 238–255.

[42] *Graphical Lasso R Package*, Accessed 28 February 2014. `http://statweb.stanford.edu/~tibs/glasso/`.

[43] Stephane Guindon and Olivier Gascuel, *A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood*, Systematic Biology **52** (2003), no. 5, 696–704.

[44] Jian Guo, Elizaveta Levina, George Michailidis, and Ji Zhu, *Joint Estimation of Multiple Graphical Models*, Biometrika (2011).

REFERENCES

[45] Archit Gupta, Pavan Kuppili, Aditya Akella, and Paul Barford, *An Empirical Study of Malware Evolution*, First International Communication Systems and Networks and Workshops, 2009, pp. 1–10.

[46] Matthew E Hayes, *Simulating Malware Evolution for Evaluating Program Phylogenies*, Ph.D. Thesis, 2008.

[47] Rainer Hettich and Kenneth Kortanek, *Semi-Infinite Programming: Theory, Methods, and Applications*, SIAM Review **35** (1993), 380–429.

[48] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji, *Intrusion Detection Using Sequences of System Calls*, Journal of Computer Security **6** (January 1998), no. 3, 151–180.

[49] Lawrence Hubert and Phipps Arabie, *Comparing Partitions*, Journal of Classification **2** (1985), 193–218.

[50] John P. Huelsenbeck and Keith A. Crandall, *Phylogeny Estimation and Hypothesis Testing Using Maximum Likelihood*, Annual Review of Ecology and Systematics **28** (1997), 437–466.

[51] Daniel H. Huson, Regula Rupp, and Celine Scornavacca, *Phylogenetic Networks: Concepts, Algorithms and Applications*, Systematic Biology (2011).

[52] *IDA Pro*, Accessed 17 September 2013. `http://www.hex-rays.com/products/ida/index.shtml`.

[53] Dimitris Iliopoulos, Christoph Adami, and Peter Szor, *Darwin Inside the Machines: Malware Evolution and the Consequences for Computer Security*, arXiv preprint arXiv:1111.2503 (2011).

[54] Grégoire Jacob, Hervé Debar, and Eric Filiol, *Behavioral Detection of Malware: From a Survey Towards an Established Taxonomy*, Journal of Computer Virology **4** (2008), no. 3, 251–266.

[55] Yuh jye Lee and Olvi L. Mangasarian, *RSVM: Reduced Support Vector Machines*, Data Mining Institute, Computer Sciences Department, University of Wisconsin, 2001, pp. 00–07.

REFERENCES

[56] Md. Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida, *Malware Phylogeny Generation Using Permutations of Code*, Journal of Computer Virology **1** (2005), 13–23.

[57] H. Kashima, K. Tsuda, and A. Inokuchi, *Kernels for Graphs*, MIT Press, 2004.

[58] *Kaspersky*, Accessed 27 February 2014. `http://usa.kaspersky.com/`.

[59] *Kaspersky Lab Report: The Bagle Botnet*, Accessed 17 September 2013. `http://www.securelist.com/en/analysis/162656090/The_Bagle_botnet`.

[60] David Kempe and Frank McSherry, *A Decentralized Algorithm for Spectral Analysis*, Journal of Computer and System Sciences **74** (2008), no. 1, 70–83.

[61] Jeffrey O. Kephart, Gregory B. Sorkin, William C. Arnold, David M. Chess, Gerald J. Tesauro, and Steve R. White, *Biologically Inspired Defenses Against Computer Viruses*, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, 1995, pp. 985–996.

[62] Marius Kloft, Ulf Brefeld, Sören Sonnenburg, and Alexander Zien, *lp-Norm Multiple Kernel Learning*, Journal of Machine Learning Research **12** (2011), 953–997.

[63] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang, *Effective and Efficient Malware Detection at the End Host.*, Proceedings of the Eighteeth USENIX Security Symposium, 2009, pp. 351–366.

[64] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel, *The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code*, Proceedings of the Eighteenth ACM Conference on Computer and Communications Security, 2011, pp. 285–296.

[65] J. Zico Kolter and Marcus A. Maloof, *Learning to Detect and Classify Malicious Executables in the Wild*, The Journal of Machine Learning Research **7** (2006), 2721–2744.

[66] Jeremy Z. Kolter and Marcus A. Maloof, *Learning to Detect Malicious Executables in the Wild*, Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2004, pp. 470–478.

REFERENCES

[67] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna, *Polymorphic Worm Detection Using Structural Information of Executables*, Recent Advances in Intrusion Detection, 2006, pp. 207–226.

[68] Abhishek Kumar, Piyush Rai, and Hal Daume III, *Co-Regularized Multi-view Spectral Clustering*, Advances in Neural Information Processing Systems, 2011, pp. 1413–1421.

[69] KVM, Accessed 17 September 2013. `http://www.linux-kvm.org/page/Main_Page`.

[70] Gregoire Jacob Lakshmanan Nataraj S. Karthikeyan and B. Manjunath, *Malware Images: Visualization and Automatic Classication*, Proceedings of VizSec, 2011.

[71] Manny M Lehman, *Laws of Software Evolution Revisited* (1996), 108–124.

[72] Corrado Leita, Ulrich Bayer, and Engin Kirda, *Exploiting Diverse Observation Perspectives to Get Insights on the Malware Landscape*, IEEE/IFIP International Conference on Dependable Systems and Networks, 2010, pp. 393–402.

[73] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*, ACM SIGPLAN Conference on Programming Language Design and Implementation (2005), 190–200.

[74] Ulrike Luxburg, *A Tutorial on Spectral Clustering*, Statistics and Computing **17** (2007), no. 4, 395–416.

[75] Robert Lyda and James Hamrock, *Using Entropy Analysis to Find Encrypted and Packed Malware*, IEEE Security & Privacy **5** (2007), no. 2, 40–45.

[76] McAfee, *McAfee Threat Report, Second Quarter*, Accessed 17 September 2013. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2013-summary.pdf`.

[77] *McAfee*, Accessed 27 February 2014. `http://www.mcafee.com/us/`.

[78] *Mineserver*, Accessed 17 September 2013. `https://github.com/fador/mineserver`.

REFERENCES

[79] Andreas Moser, Christopher Kruegel, and Engin Kirda, *Limits of Static Analysis for Malware Detection*, Computer Security Applications Conference, Annual **0** (2007), 421–430.

[80] *NetworkMiner*, Accessed 17 September 2013. `http://sourceforge.net/projects/networkminer/`.

[81] Huazhong Ning, Wei Xu, Yun Chi, Yihong Gong, and Thomas Huang, *Incremental Spectral Clustering with Application to Monitoring of Evolving Blog Communities*, In SIAM International Conference on Data Mining, 2007.

[82] *objdump*, Accessed 28 February 2014. `http://www.gnu.org/software/binutils/`.

[83] *Offensive Computing*, Accessed 17 September 2013. `http://www.offensivecomputing.com/`.

[84] Roberto Perdisci, David Dagon, Prahlad Fogla, and Monirul Sharif, *Misleading Worm Signature Generators Using Deliberate Noise Injection*, Proceedings of the IEEE Symposium on Security and Privacy, 2006, pp. 17–31.

[85] John Platt, *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*, Technical Teport MSR-TR-98-14, Microsoft Research (1998).

[86] *Portable Executable iDentifier*, Accessed 6 October 2011. `http://peid.info/`.

[87] Daniel Quist, Lorie Liebrock, and Joshua Neil, *Improving Antivirus Accuracy with Hypervisor Assisted Analysis*, Journal of Computer Virology (2010), 1–11.

[88] Alain Rakotomamonjy, Francis Bach, Stéphane Canu, and Yves Grandvalet, *More Efficiency in Multiple Kernel Learning*, Proceedings of the Twenty-Fourth International Conference on Machine Learning, 2007, pp. 775–782.

[89] Dubba Reddy, Subrat Dash, and Arun Pujari, *New Malicious Code Detection Using Variable Length N-grams*, Information Systems Security, 2006, pp. 276–288.

[90] Dubba Reddy and Arun Pujari, *N-gram Analysis for Computer Virus Detection*, Journal of Computer Virology **2** (2006), 231–239.

REFERENCES

[91] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dssel, and Pavel Laskov, *Learning and Classification of Malware Behavior*, Detection of Intrusions and Malware, and Vulnerability Assessment, 2008, pp. 108–125.

[92] Bernhard Schölkopf and Alexander Johannes Smola, *Learning with Kernels*, MIT Press, 2002.

[93] R. Sekar, Michael Bender, Dinakar Dhurjati, and Pradeep Bollineni, *A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors*, IEEE Symposium on Security and Privacy, 2001, pp. 144–155.

[94] Yang seo Choi, Ik kyun Kim, Jin tae Oh, and Jae cheol Ryou, *PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)*, International Symposium on Computer Science and Its Applications, 2008, pp. 28–31.

[95] M. Shafiq, Syed Khayam, and Muddassar Farooq, *Embedded Malware Detection Using Markov N-grams*, Detection of Intrusions and Malware and Vulnerability Assessment, 2008, pp. 88–107.

[96] Madhu Shankarapani, Subbu Ramamoorthy, Ram Movva, and Srinivas Mukkamala, *Malware Detection Using Assembly and API Call Sequences*, Journal of Computer Virology **7** (2010), no. 2, 1–13.

[97] Nino Shervashidze, S. V. N. Vishwanathan, Tobias H. Petri, Kurt Mehlhorn, and Karsten M. Borgwardt, *Efficient Graphlet Kernels for Large Graph Comparison*, Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS), 2009, pp. 488–495.

[98] Robert Sokal and James Rohlf, *The Comparison of Dendrograms by Objective Methods*, Taxon **11** (1962), no. 2, 33–40.

[99] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena, *BitBlaze: A New Approach to Computer Security via Binary Analysis*, Information Systems Security, 2008, pp. 1–25.

REFERENCES

[100] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo, *On the Infeasibility of Modeling Polymorphic Shellcode*, Proceedings of the Fourteenth ACM Conference on Computer and Communications Security, 2007, pp. 541–551.

[101] Sören Sonnenburg, Gunnar Raetsch, and Christin Schaefer, *A General and Efficient Multiple Kernel Learning Algorithm*, Nineteenth Annual Conference on Neural Information Processing Systems (2005).

[102] Sören Sonnenburg, Gunnar Rätsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtech Franc, *The SHOGUN Machine Learning Toolbox*, Journal of Machine Learning Research **11** (2010), 1799–1802.

[103] Sören Sonnenburg, Gunnar Rätsch, Christin Schäfer, and Bernhard Schölkopf, *Large Scale Multiple Kernel Learning*, Journal of Machine Learning Research **7** (2006), 1531–1565.

[104] *Sophos*, Accessed 27 February 2014. `http://www.sophos.com/en-us.aspx`.

[105] Salvatore Stolfo, Ke Wang, and Wei-Jen Li, *Towards Stealthy Malware Detection*, Malware Detection, 2007, pp. 231–249.

[106] Salvatore J. Stolfo, Ke Wang, and Wei-Jen Li, *Fileprint Analysis for Malware Detection*, ACM Workshop on Recurring/Rapid Malcode, 2005.

[107] Orathai Sukwong, Hyong S Kim, and James C Hoe, *Commercial Antivirus Software Effectiveness: An Empirical Study*, Computer (2011), 63–70.

[108] *Symantec*, Accessed 27 February 2014. `https://www.symantec.com/index.jsp`.

[109] *Symantec Bagle Security Report*, Accessed 17 September 2013. `http://www.symantec.com/security_response/writeup.jsp?docid=2004-011815-3332-99`.

[110] *Symantec Koobface Security Report*, Accessed 17 September 2013. `http://www.symantec.com/security_response/writeup.jsp?docid=2008-080315-0217-99`.

[111] *Symantec Mytob Security Report*, Accessed 17 September 2013. `http://www.symantec.com/security_response/writeup.jsp?docid=2005-022614-4627-99`.

REFERENCES

[112] The Silicon Realms Toolworks, *Armadillo Software Protection System*, Accessed 6 October 2011. `http://www.siliconrealms.com/`.

[113] *UPX: The Ultimate Packer for eXecutables*, Accessed 6 October 2011. `http://upx.sourceforge.net/`.

[114] *Virus Total*, Accessed 17 September 2013. `http://www.virustotal.com/`.

[115] *VX Heaven*, Accessed 27 February 2014. `http://vxheaven.org/`.

[116] Grard Wagener, Radu State, and Alexandre Dulaunoy, *Malware Behaviour Analysis*, Journal of Computer Virology **4** (2008), no. 4, 279–287.

[117] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, *Detecting Intrusions using System Calls: Alternative Data Models*, Proceedings of the IEEE Symposium on Security and Privacy, 1999, pp. 133–145.

[118] Scott White and Padhraic Smyth, *A Spectral Clustering Approach to Finding Communities in Graphs*, Proceedings of the Fifth SIAM International Conference on Data Mining, 2005, pp. 76–84.

[119] Daniela M Witten, Jerome H Friedman, and Noah Simon, *New Insights and Faster Computations for the Graphical Lasso*, Journal of Computational and Graphical Statistics **20** (2011), no. 4, 892–900.

[120] Stephen J Wright, *Primal-Dual Interior-Point Methods*, Vol. 54, SIAM, 1997.

[121] Ilsun You and Kangbin Yim, *Malware Obfuscation Techniques: A Brief Survey*, International Conference on Broadband, Wireless Computing, Communication and Applications, 2010, pp. 297–300.

[122] Stella X Yu and Jianbo Shi, *Multiclass Spectral Clustering*, Proceedings of the Ninth IEEE International Conference on Computer Vision, 2003, pp. 313–319.

[123] Ming Yuan and Yi Lin, *Model Selection and Estimation in the Gaussian Graphical Model*, Biometrika **94** (2007), no. 1, 19–35.

REFERENCES

[124] Qinghua Zhang and D.S. Reeves, *MetaAware: Identifying Metamorphic Malware*, Twenty-third Annual Computer Security Applications Conference, 2007.

[125] Ying Zhao and George Karypis, *Evaluation of Hierarchical Clustering Algorithms for Document Datasets*, Proceedings of the Eleventh International Conference on Information and Knowledge Management, 2002, pp. 515–524.

[126] Dengyong Zhou and Christopher J. C. Burges, *Spectral Clustering and Transductive Learning with Multiple Views*, Proceedings of the Twenty-Fourth International Conference on Machine Learning, 2007, pp. 1159–1166.