Improving Malware Classification: Bridging the Static/Dynamic Gap

Blake Anderson
Los Alamos National Laboratory
banderson@lanl.gov

Curtis Storlie
Los Alamos National Laboratory
storlie@lanl.gov

Terran Lane
The University of New Mexico
terran@cs.unm.edu

ABSTRACT

Malware classification systems have typically used some machine learning algorithm in conjunction with either static or dynamic features collected from the binary. Recently, more advanced malware has introduced mechanisms to avoid detection in these views by using obfuscation techniques to avoid static detection and execution-stalling techniques to avoid dynamic detection. In this paper we construct a classification framework that is able to incorporate both static and dynamic views into a unified framework in the hopes that, while a malicious executable can disguise itself in some views, disguising itself in every view while maintaining malicious intent will prove to be substantially more difficult. Our method uses kernels to place a similarity metric on each distinct view and then employs multiple kernel learning to find a weighted combination of the data sources which yields the best classification accuracy in a support vector machine classifier. Our approach opens up new avenues of malware research which will allow the research community to elegantly look at multiple facets of malware simultaneously, and which can easily be extended to integrate any new data sources that may become popular in the future.

Categories and Subject Descriptors

I.5.2 [**Design Methodology**]: Classifier design and evaluation; K.6.5 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses

General Terms

Security, Algorithms, Experimentation

Keywords

Computer Security, Malware, Machine Learning, Multiple Kernel Learning

1. INTRODUCTION

In 2010, more than 286 million unique variants of malware were detected [47]. Despite the majority of this new malware being created through polymorphism and simple code obfuscation techniques, and thus being very similar to known malware, it will still not be detected by signature-based anti-virus programs [13, 32]. To meet these challenges, machine learning techniques have been developed that are able to learn a generalized description of malware and apply this knowledge to classify new, unseen instances of malware.

The machine learning techniques for malware classification have used a variety of data sources to learn discriminatory functions that are able to differentiate benign and malicious software. Some of the most popular data sources that have been examined include binary files [21, 46], disassembled files [10, 40], entropy measures on the binary [27], dynamic system call traces [9, 18], dynamic instruction traces [4, 14], and control flow graphs [13, 22].

The novel contribution of this paper is to show how to combine different data sources using multiple kernel learning [43] to arrive at a new classification system that integrates all of the available information about a program into a unified framework. The key insight of our approach is that each data source provides complementary information about the true nature of a program, but no one data source contains all of this information. We aim to construct a classification system that does contain all aspects of a program's true intentions. We 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 source. We then use multiple kernel learning [6, 43] to find the weights of each kernel, create a linear combination of the kernels, and finally use a support vector machine [12] to perform classification.

Our framework is particularly appealing for three reasons. First, we are able to elegantly combine both static and dynamic features in a way which allows the learning algorithm to take advantage of both simultaneously. Second, our method is extendable in the sense that future popular data sources could be easily added to the model without complicating the final result. Finally, the method presented in this paper is highly parallelizable: computing the kernel values for testing new malware can all be done in parallel, the implication being that larger datasets can easily be handled.

We present our results on a dataset composed of 776 benign programs and 780 malicious programs. In addition to this dataset, we test our methods on a separate validation dataset composed of 20,936 malicious samples. For each program we collect six data sources: 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 sources. For the binary file, disassembled file, and two

Copyright 2012 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
AISec'12, October 19, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1664-4/12/10 ...$10.00.

dynamic traces, we build kernels based on the Markov chain graphs; for the control flow graph we use a graphlet kernel [41]; and for the file information feature vector we use a standard Gaussian kernel as explained in Section 4.

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

2. RELATED WORK

There has been a large volume of work applying machine learning techniques to the malware classification problem [8, 14, 22, 35]. Most of this work has used a single data source (i.e. static binary or dynamic system call data) and some set of machine learning algorithms (i.e. support vector machines or decision trees) to perform classification and/or clustering. Unlike these methods, we incorporate multiple data sources, both static and dynamic, by using multiple kernel learning [43] to perform classification.

Combining different features of static files has been examined [28]. Here the authors use n-grams and features about the static binary 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. In our work, we also incorporate learning with dynamic trace data, which has been shown to be very important for classifying classes of malware which are packed or obfuscated in other ways [29]. Also, the final combined kernel that we find can be used in a kernel-based clustering algorithm [26] to look at the phylogenetics of the malware as discussed in Section 6.

Some authors have looked into combining static and dynamic features. In [24], the authors build different clusterings based on exploits, payloads, malware, and behavioral features captured by the Anubis framework [5]. Then they explore the relationships between the different clusterings. This is a *results fusion model*, whereas we are concerned with a *data fusion model*. We do not build separate models for each type of static and/or dynamic data source, but rather combines all of the data sources using multiple kernel learning to arrive at a unified view of what it means to be malicious. Another key difference of our approach is that we are concerned with classification and not clustering, with previous efforts not being suitable for the classification domain. We also base our analysis on different types of data. [24] uses 17 static features for the exploit/payload/malware clusters, whereas we compare the Markov chain graphs based on the binary and disassembled information, control flow graphs, and other file information statistics described in Section 3. The behavioral features of Anubis are concerned with modifications to the Windows registry and file system, or interactions with other processes. Our dynamic analysis is based on the Markov chain graphs of the dynamic instructions and system calls performed. It is important to note that the framework presented in this paper is general enough to include the data sources of [24] once an appropriate Kernel is defined. Finally, we combine the information of the data sources using a multiple kernel learning framework and build a single unified classification system, whereas the individual clusterings found in [24] are based solely on their respective data sources.

There has been some work done which combines static and dynamic analysis to locate packed and/or obfuscated code. In [36], the authors first disassemble the code, and then run the code looking for sequences of instructions in the dynamic trace that are not found in the disassembled data. They are only concerned with locating the hidden code which could not be disassembled, whereas we are concerned with classifying instances of malware, whether they have hidden code or not. We also make use of more data sources, such as system calls and control flow graphs.

3. DATA SOURCES

In this work, we take advantage of six different types of data with the aim of covering the most popular data sources that have been used for malware classification in the literature. These data sources 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, disguising itself in every view while maintaining malicious intent will prove to be substantially more difficult. We use three static data sources: the binary file, the disassembled binary, and the control flow graph of the disassembled binary. We use two dynamic data sources: the dynamic instruction trace and the dynamic system call trace. Finally, we use a file information data source which contains seven statistics that provide a summary of the previous data sources.

Binary. We use the raw byte information contained in the binary executable to construct our first data source. There is a long history of using this type of data to classify malware [21, 46]. 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 [21]). In contrast to these methods, we use the 2-grams to condition a Markov chain and then perform classification in graph space as explained in Section 4. 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 [10, 40]. To generate the disassembled code, we use IDA Pro [33]. Once we have the disassembled code, we build a Markov chain similar to the way we built the Markov chain for the binary files. Instead of the byte values being the vertices in the graph, we use the disassembled instructions for the vertices in the graph. Unfortunately, the number of unique instructions found in the disassembled files (∼1200) gave us very large Markov chains that overfitted the data resulting in poor initial performance. This is in part due to the *curse of dimensionality*: the feature space becomes too large and we do not have enough data to sufficiently condition the model. Furthermore, some instructions perform identical or very similar tasks, resulting in many transitions that are identical yet treated as distinct. To combat this, we used several categorizations, each with increasing complexity. The coarsest categorization contained eight categories (math, logic, privileged, branch, memory, stack, nop, and

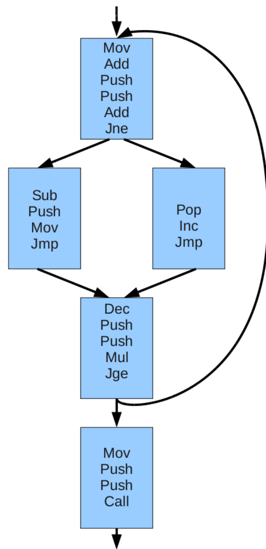


Figure 1: An example of a control flow graph demonstrating jumps.

other). The other categorizations had 34, 68, 77, 86, 154, and 172 categories. We 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 6.

Control Flow Graph. The use of control flow graphs has become a very popular means to perform malware classification [13, 22]. 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 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 [22]. To compute the similarity between different control flow graphs, we use a simplified kernel based on previous work in the literature [22], which works by counting similarly shaped subgraphs of a certain size. This kernel is explained in detail in Section 4.

Dynamic Instruction Traces. Although static analysis techniques for malware classification have been proven to perform quite well, it has been shown that these methods can be evaded by using advanced obfuscation transformations [29]. Because of these limits to static analysis, we chose to include two dynamic data sources, the instruction traces and the system call traces collected over a five minute run using the Xen virtual machine [7] and the Intel Pin program [25]. Dynamic instructions traces are known to produce highly accurate malware classification results [4, 14]. Over the 1556 traces, we recorded 237 unique instructions. We built the Markov chains in the same fashion as the disassembled code, using the seven categorizations mentioned previously and using all 237 unique instructions. We found that mapping each of the 237 unique instructions to 237 unique vertices provided the best results.

Statistic	Malware	Benign
Entropy	7.52	6.34
Binary Size	.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 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.

Dynamic System Call Traces. System call traces have been another popular dynamic data source [9, 18, 38]. Over the 1556 traces, we recorded 2460 unique system calls. We used the Markov chain graph representation, and like the disassembled instruction set, we found that treating each unique system call as a vertex in the Markov chain led to poor initial performance. We 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 source, we collected seven pieces of information about the various data sources described previously. This data is summarized in Table 1. We look at the entropy and the size of the binary file. Similar to previous work on entropy [27, 39], we found the average entropy of the benign files in our dataset to be 6.34 and the average entropy of the malicious files in our dataset to be 7.52. We also have a binary feature to look at whether the binary executable has a recognizable packer such as UPX [49] or Armadillo [48]. To find whether a file was packed or not, we used the PEID signature method [3]. For the disassembled binary feature, we took the number of instructions found in the disassembled file. We also use the number of edges and the number of vertices in the control flow graph. Finally, we took the sum of the number of dynamic instructions and dynamic system calls as the last feature.

4. METHOD

In this section, we first describe how we transform the six data sources of Section 3 into more convenient representations. We then show how it is possible to define kernels, or similarity measures, that are able to accurately compare these data sources in their new representations. Finally, we 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 sources described in Section 3 can be grouped into three sets. The miscellaneous file information that we collect can be represented as a simple feature vector of length seven where each

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
...	...

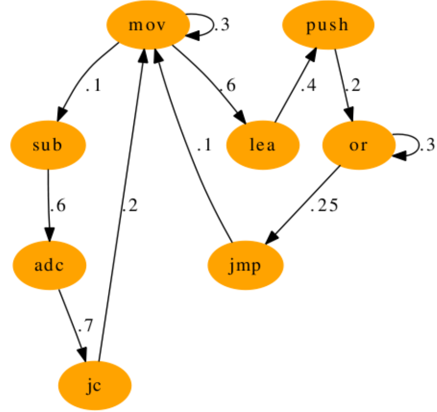


Figure 2: The left table shows an example of the trace data we collect. A hypothetical resulting graph representing a fragment of the Markov chain is shown on the right. In a real Markov chain graph, all of the out-going edges would sum to 1.

of the seven statistics corresponds to a feature. The control flow graphs are represented in the same way as the standard in the literature [13, 22]. For the third set, the raw binary, disassembled binary, dynamic instruction trace, and dynamic system call trace, we use a Markov chain representation.

Given some data source, such as the dynamic instruction trace $\mathcal{P}$, we are interested in finding a new representation, $\mathcal{P}'$, such that we can make unified comparisons in graph space while still capturing the sequential nature of the data. We achieved this by transforming the data into a Markov chain, which is represented 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 vertices and the elements of E are edges. In our representation, the edge weight, e_{ij}, between vertices i and j corresponds to the transition probability from state i to state j in a Markov chain, hence, we require the edge weights for edges originating at v_i to sum to 1, $\sum_{i \rightsquigarrow j} e_{ij} = 1$. We use an $n \times n$ ($n = |V|$) adjacency matrix to represent the graph, where each entry in the matrix, $a_{ij} = e_{ij}$.

As an illustrative example, we will now consider the dynamic instruction trace. We found 237 unique instructions across all of the traces we collected. These instructions are the vertices of the Markov chains. The 237 instructions are irrespective of the operands used with those instructions. By ignoring operands, we remove sensitivity to register allocation and other compiler artifacts. It is important to note that rarely did the instruction trace of a single program make use of all 237 unique instructions, and therefore, the adjacency matrix of that Markov chain graph will contain some rows of zeros. The decision to incorporate unused instructions in the model allowed us to maintain a consistent vertex set between all instruction trace graphs, granting us the ability to make uniform comparisons in graph space.

To find the transition probabilities of the Markov chain, we first scan the dynamic instruction trace, keeping counts for each pair of successive instructions. After filling in the adjacency matrix with these values, we normalize the matrix such that all of the non-zero rows sum to one. This process of estimating the transition probabilities ensures us a well-formed Markov chain. Figure 2 shows a snippet of dynamic instruction trace data with a resulting fragment of the hypothetical Markov chain graph.

Kernels. 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 [37]. 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 this feature space. Let X be any dataset, then a kernel, $K : X \times X \to \mathbb{R}$, is defined as:

$$K(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle \quad (1)$$

where $\mathbf{x}, \mathbf{x}' \in X$, $\langle \cdot, \cdot \rangle$ 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:

1. A kernel must be symmetric: for all $\mathbf{x}$ and $\mathbf{y} \in X$, $K(\mathbf{x}, \mathbf{y}) = K(\mathbf{y}, \mathbf{x})$.
2. A kernel must be 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*, which replaces inner products with kernel evaluations [37]. 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.

For the Markov chain representations and the file information feature vector, we used a standard squared exponential kernel:

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

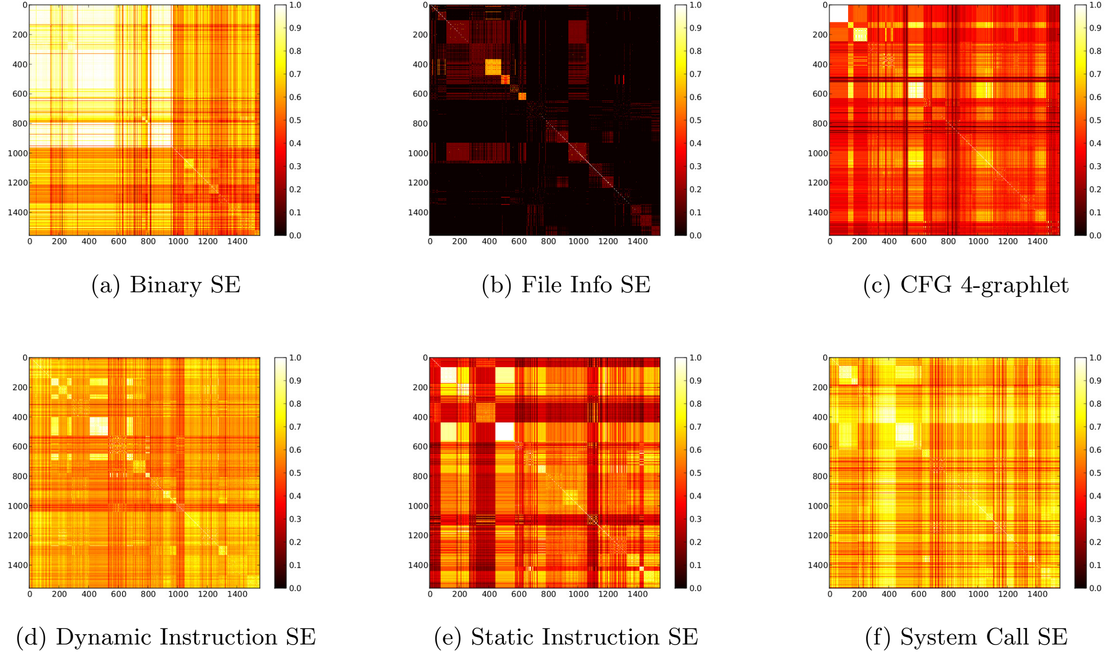
where $\mathbf{x}_i$ represents one of the seven features for the file information data source, or a transition probability for the Markov chain representations. σ and λ are the hyperparameters of the kernel function (estimated through cross-validation), and $\sum_{i,j}$ sums the squared distance between the corresponding features.

For the control flow graph data source, we attempted to find a kernel that closely matched the work done in the literature [22]. Although the approach we chose did not take the instruction information of the basic blocks into account, we settled on a graphlet kernel due to its computational efficiency [41]. A k-graphlet is defined as a subgraph, of a graph G, with the number of nodes of the subgraph equal to k. If we let f_G be a feature vector, where each feature is the number of times a unique graphlet of size k occurs in G, the normalized probability vector is:

$$D_G = \frac{f_G}{\# \text{ of all graphlets of size } k \text{ in } G} \quad (3)$$

and we have the following graphlet kernel:

$$K_g(G, G') = D_G^T D_{G'} \quad (4)$$



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

(d) Dynamic Instruction SE (e) Static Instruction SE (f) System Call SE

Figure 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.

We 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 3 shows the heatmaps for the six individual kernels and Figure 4 shows the heatmap of the combined kernel with the weights of the individual kernels being found using multiple kernel learning (see Equation 10). The block structure observed in these figures is very interesting as it shows that the kernels and data sources we selected are able to discriminate between malware and benign samples. This is very apparent in Figure 4 where we see that the top left block (the similarity between the malware samples) has very high values compared with the rest of the image.

These figures, especially Figure 3 (b), (c), and (e), offer support in using the methodology presented in this paper in a malware phylogenetic setting. The structured blocks along the diagonal lend support to these samples coming from similar families. Ideas to extend the current work for use in malware phylogenetics is examined in Section 6.

If we have some set of M valid kernels, K_1, K_2, ..., K_M, we are assured that

$$K_{comb} = \sum_{1 \leq i \leq M}^{M} K_i \tag{5}$$

is also a valid kernel [11]. This algebra on kernels allows us to elegantly combine kernels that measure very different aspects of the input data, or even different views of the data, and is the object of study in multiple kernel learning [6, 43].

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 [12] and can be found with

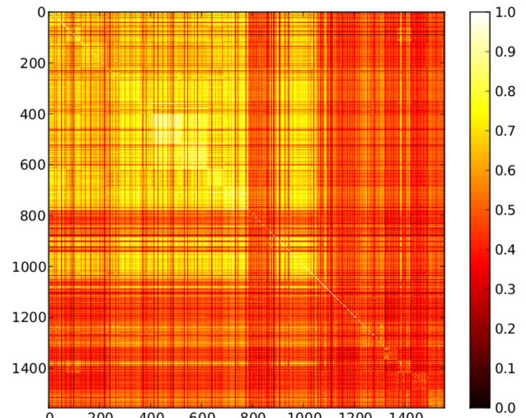


Figure 4: Heatmap for all six kernels combined with the weights found using multiple kernel learning.

the following optimization problem:

$$\min_{\alpha} \underbrace{\left(\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{n} \alpha_i \right)}_{S_k(\alpha)} \tag{6}$$

subject to the constraints:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \quad (7)$$
$$0 \leq \alpha_i \leq C$$

where y_i is the class label of instance x_i. Equation 7 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 to prevent over-fitting the training data and allows for better generalization accuracy.

Given α found in Equation 6, we have the following decision function:

$$f(\mathbf{x}) = \text{sgn}\left(\sum_{i}^{n} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right) \quad (8)$$

which returns class +1 if the summation is ≥ 0, and class -1 if the summation is < 0.

With multiple kernel learning, we are interested in finding β, in addition to the standard α of support vector machines, such that

$$K_{comb}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{M} \beta_k K_k(\mathbf{x}_i, \mathbf{x}_j) \quad (9)$$

is a convex combination of M kernels with $\beta_k \geq 0$, where each kernel, K_k, uses a distinct set of features [43]. In our case, each distinct set of features is a different view of the data given by our different data sources (Section 3). The general outline of the algorithm is to first combine the kernels with $\beta_k = 1/M$, find α, and then iteratively keep optimizing for β and α until convergence.

To solve for β, assuming we have a fixed set of support vectors (α), the following semi-infinite linear program has been proposed [43]:

$$\begin{aligned} \max \quad & \theta \\ \text{w.r.t.} \quad & \theta \in \mathbb{R}, \beta \in \mathbb{R}^K \end{aligned} \quad (10)$$

subject to the constraints:

$$\mathbf{0} \leq \beta \quad (11)$$
$$\sum_k \beta_k = 1$$
$$\sum_{k=1}^{M} \beta_k S_k(\alpha) \geq \theta$$

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 6. M is the number of kernels to be combined. This is a semi-infinite linear program as all of the constraints in Equation 11 are linear, and there are infinitely many of these constraints, one for each $\alpha \in \mathbb{R}^N$ satisfying $\mathbf{0} \leq \alpha \leq \mathbf{1}C$ and $\sum_i y_i \alpha_i = 0$ [17].

To find solutions for both α and β, an iterative algorithm was proposed that first uses a standard support vector machine algorithm to find α (Equation 6), and then fixes α and solves Equation 10 to find β. While this algorithm is known to converge, there are no known convergence rates [17]. Therefore, the following stopping criterion was proposed [43]:

$$\epsilon^{t+1} \geq \epsilon^t := \left| 1 - \frac{\sum_{k=1}^{M} \beta_k^t S_k(\alpha^t)}{\theta^t} \right| \quad (12)$$

Method	Acc (%)	FPs	FNs	AUC
All Six Data Sources	**98.07%**	16	**14**	**.9978**
Three Static Sources	96.14%	36	24	.9934
Two Dynamic Sources	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
AV0	78.46%	4	331	n/a
AV1	75.26%	7	378	n/a
AV2	71.79%	**0**	439	n/a

Table 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.

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 [43]. It is important to note that this optimization problem only needs to be solved once, as the support vectors (α) and kernel weights (β) found can be used to classify all of the newly collected data.

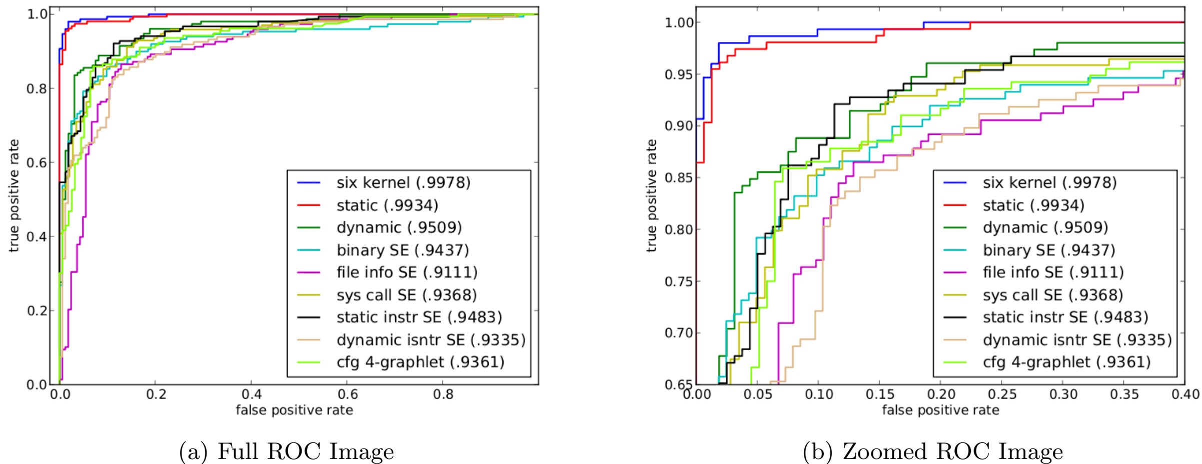
5. RESULTS

In this section, we present our results on a dataset composed of 1,556 samples, 780 malicious programs and 776 benign programs. This dataset was provided by Offensive Computing [1]. In order to get a relevant sample set, the Offensive Computing samples were obtained from a fresh malware feed. This feed takes place between security vendors who harvest malware from a variety of sources such as web spidering and customer uploads. These are distributed in a batched format to researchers on a daily basis. We also present results which demonstrates the generalization accuracy of our methods on a separate dataset composed of 20,936 malicious samples. All of this data was gathered over a six month period, from August 2011 to January 2012.

The metrics we use to quantify our results are classification accuracy, receiver operating characteristic (ROC) curves, area under the ROC curve (AUC), and the average time it takes to classify a new instance. We look at three combination strategies: static sources, dynamic sources, and a combination with all six of our data sources. We compare our combined kernel method against methods based on a single data source. We conclude this section with some interesting observations we found with regard to the performance of the static sources versus the dynamic sources.

Machine/Tools. To perform all of our experiments, we used a machine with quad Xeon X5570s running at 2.93GHz and having 24GB of memory. To perform the multiple kernel learning, we used the modular Python interface of the Shogun Machine Learning Toolbox [44].

Accuracy. Table 2 presents our results for the individual kernels as well as the combined kernels using 10-fold cross validation. The three best performing anti-virus programs (out of 11 considered) are also shown. For the anti-virus program results, it is important to emphasize that our malicious dataset was not composed of zero-day malware, but rather malware that is 9 months to a year old. Despite



(a) Full ROC Image

(b) Zoomed ROC Image

Figure 5: ROC curves for all six individual kernels, a kernel based on the three static sources, a kernel based on the two dynamic sources, and the combined kernel composed of all six data sources. It is easy to see that the kernel based on all six data sources performs significantly better than the other kernels. The full AUC values for the kernels are are listed in legend in parenthesis.

Method	.01	.05	.1	.25	.5	Full AUC
All Six Data Sources	**.9467**	**.9867**	**.9933**	**1.0**	**1.0**	**.9978**
Three Static Sources	.9224	.9634	.9812	**1.0**	**1.0**	.9934
Two Dynamic Sources	.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 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 sources performs the best at all values.

this, the worst performing data source based on the file information feature vector still had ∼6% better classification accuracy than the best anti-virus program. All but one of the false positives found by the anti-virus programs were actually confirmed to be true positives as discussed later in this section.

The best performing method was the combined kernel that used all six data sources and achieved an accuracy of 98.07%. Although using purely static sources performed very well (96.14%), adding dynamic information significantly improved overall performance. All of the single data sources were between 84% to 89% with the single data source winner being the disassembled binary at 89.97%. The disassembled binary most likely has an advantage over the raw binary because it was unpacked before it was disassembled.

ROC Curves / AUC Values. To analyze the different data sources with regard to different false positive thresholds, we looked at the ROC curves and various ROC points representing different false positive rates. Figure 5 plots all the ROC curves together (including the combined kernels) along with a zoomed version of the curve. Figure 5 (b) is particularly informative as we can see that the combined kernel, which includes all six data sources, performs better than any single data source or the two other combined kernels for all false positive rates. If there are certain time and/or other resource constraints, Figure 5 (b) also shows that we are able to achieve reasonably high results by just using the kernel based on the three static sources.

Table 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. We see that by integrating all six data sources, we can achieve an AUC value of .9467 with a .01 false positive rate, significantly higher than any other kernel based on a single data source, which adds more support to the power of our approach and multiple kernel learning in general. The file information data source is the worst performing data source with regard to this metric, and as we explain at the end of this section, we 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 10 are an interesting way to look at how informative different data sources are with regard to the classification accuracy. The weights we found for our data views are shown in Table 5. It is interesting to note that the weights are *not* representative of how well the single data source does at classification. For example, the dynamic instruction view has the highest kernel weight among the weights for all six

Method	Trace	Data Transformation	Classify	Total
All Six Data Sources	300.0s	3.12s	0.21s	303.52s
Three Static Sources	n/a	1.18s	0.13s	2.03s
Two Dynamic Sources	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: The average time it takes from receiving the raw data until a classification decision can be made. This time includes running the program for the dynamic sources (we allow 5 minutes 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 8.

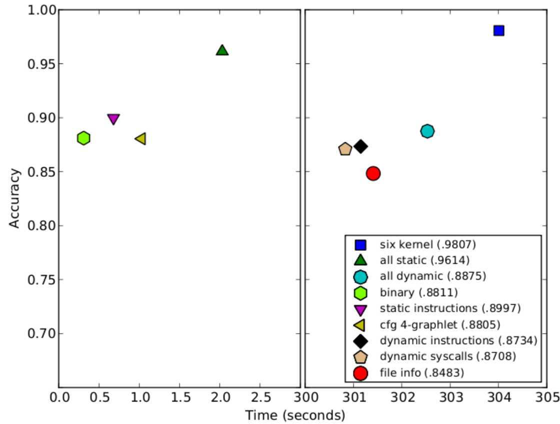


Figure 6: Plot demonstrating the trade-off between accuracy and time to classify. Time is in seconds and the x-axis is the time it takes to first collect the dynamic trace (for the dynamic data sources), transform the data instance into our representation, and finally classify the instance.

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

Table 5: The kernel weights learned from Equation 10 for the different kernel combinations we examine.

views, but in our 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 only need to be done once, offline), we will focus our analysis of the timing results on the average amount of time it takes to classify a new instance. As a note of interest, the time to find the kernel weights and support vectors for the kernel composed of all six data sources, averaged over 10 runs, was only 0.86 seconds!

Given a new instance to classify, there are two or three steps, depending on whether we are using a dynamic data source, that must be performed:

1. Run the instance in a virtual machine keeping a log of the instructions and system calls the program performs.
2. Transform the data source into one of our data representations.
3. Classify the transformed data instance according to Equation 8.

In our timing results, we allow five minutes to collect the dynamic trace data. Transforming the data to our representation could mean several different things for each of the different data sources. For instance, we might have to disassemble the data, find the 2-grams in the disassembled data, and finally build the Markov chain; we might have to build the control flow graph and find the number of graphlets with a specific structure; we might have to collect the statistics of Table 1 in the case of the file information data source; etc. For the classification step, it is important to note that support vector machines are known to find sparse α vectors [12], easing the computational burden of Equation 8.

The timing results, which are broken down into the three stages, are presented in Table 4 and are pictorially featured in Figure 6. These results were averaged over the entire dataset. As the reader can see, 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 sources is especially impressive as we have shown it to have great performance and it only takes 2.03 seconds on average to transform the data and make a classification decision. Using the combined kernel based on the static data sources, assuming 2.03 seconds per data instance, it is possible to classify up to 42,561 new samples each day! Again, the methods presented in this paper are highly parallelizable, and these methods will scale very easily with more computational resources.

Testing on a Large Malware Sample. As explained earlier in this section, obtaining a large benign dataset is difficult. On the other hand, we had an additional 20,936 malware samples at our disposal. We are treating this data as a validation set to test the generalization accuracy of our methods. We 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 20,936 malicious samples as either benign or malicious according to Equation 8.

Method	Accuracy (%)
All Six Data Sources	**97.97**%
Three Static Sources	95.27%
Two Dynamic Sources	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%
AV0	57.55%
AV1	56.01%
AV2	55.32%

Table 6: The classification accuracy on a validation dataset consisting on 20,936 malicious samples.

The results for this experiment are shown in Table 6. The classification accuracies are similar to those of the dataset with 780 malicious and 776 benign samples with the exception that the signature-based anti-virus 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. As with the previous results, we can see that by combining data sources in our multiple kernel learning framework we were able to achieve a large increase in classification accuracy. The results presented in Table 6 suggest that our 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 our classifier, we assumed that the labeled benign samples in our dataset were actually benign. This was reasonable as the executables were taken from clean installations of commercial software. To test this hypothesis, we ran our classifier based on the combined kernel, with all six data sources, using 10-fold cross validation 50 times and counted how many times a data instance was classified incorrectly. We 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 [2]. It is interesting that our 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 much more manageable. Note that we did not correct the dataset for this paper, 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 [29]. Due to these limitations, we chose to include dynamic data sources to improve malware classification despite the time constraints these data collection methods impose. To further analyze some of the pitfalls of our static data sources, we again ran the combined kernel with all six data sources, a kernel with all of the static data sources, a kernel with all of the dynamic data sources, and the six separate kernels, one for each of the six different data sources 50 times, keeping track of the files that were consistently misclassified with respect to each kernel.

Table 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 sources, the percentage of false positives that were packed is the same as the packed percentage of the training data, which would offer support for the kernels based on these data sources not being deceived by the packer. Although the dynamic traces of packed files will 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 [34]. A notable exception of static sources stumbling on programs that were packed is the disassembled data source (and the control flow graph which is based on the disassembled data source), but to get these data sources, we first had to unpack the binary.

Method	Benign	Malicious
All Six Data Sources	**0.00**%	70.00%
Three Static Sources	20.00%	36.84%
Two Dynamic Sources	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%

Table 7: % of files which were packed and consistently misclassified over 50 runs with our 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 Sources	7.43	6.77
Three Static Sources	7.41	6.91
Two Dynamic Sources	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

Table 8: Average entropy of files consistently misclassified over 50 runs with our different kernels. Note that the average entropies over the entire dataset are 6.34 and 7.52 for benign and malicious files respectively.

Table 8 shows the average entropy of files which were consistently misclassified. The link between entropy and files being packed is well-known [27], therefore we see similar results to Table 7. Again, the two dynamic sources' 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 source had problems with classifying instances whose entropies differ significantly from the norm. As we would expect, the file information data source had the most problems with entropy (remember that entropy is one of the seven features for this source), 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.

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 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 2.

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, there are malware that are able to detect if they are being run in a sandboxed environment and being traced [9]. We choose the Intel Pin program [25] because it allowed us to collect both instructions and system calls simultaneously, but it does not make an effort to be a transparent tracing tool like the Ether framework [15]. The dynamic instruction data source's classification accuracy in this paper is lower than that reported in the literature [14] and we suspect that this is in part due to Intel Pin not being transparent and the malware altering its behavior.

To test this hypothesis, we looked at 8 malicious data instances that were consistently misclassified (over 50 runs) by the kernel based on the dynamic instruction data source. We 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 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 we used the same tracing tool for both samples (a kernel value of 0 is also highly unlikely for obvious reasons). Although this is a coarse measure as to whether the program alters its behavior, it does give us 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 do suggest that their dynamic instruction traces are substantially different under the different tracing tools.

6. LIMITATIONS AND FUTURE WORK

There has been a rising interest in learning how to cluster malware so that researchers can gain insight into the phylogenetic structure of current viruses [8, 19, 23, 35]. Because the majority of new viruses are derived from, or are composites of, established viruses, this information would allow for more immediate responses and allow researchers to understand the new virus much more quickly. Given our kernel matrix, we can easily use spectral clustering [26] to partition our dataset into groups with similar structure with regard to the data sources we have chosen. We would first construct the weighted graph Laplacian, a $|V| \times |V|$ matrix:

$$\mathcal{L} = \begin{cases} 1 - \frac{e_{vv}}{d_v} & \text{if } u = v, \text{ and } d_v \neq 0, \\ -\frac{e_{uv}}{\sqrt{d_u d_v}} & \text{if } u \text{ and } v \text{ are adjacent}, \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

where e_{vv} is the edge weight, in our case the entry in the kernel matrix, and d_v is the degree of the vertex, which would be the sum of the corresponding row in the kernel matrix. Then we would perform an eigen-decomposition on the Laplacian and take the l-smallest eigenvectors. Finally, we would use standard k-means clustering with the l-smallest eigenvectors as the features [16].

With our method, we are not restricted to only using different data sources, but we also have the flexibility to incorporate different data representations for each data source assuming a suitable kernel can be defined. For instance, it is common in the literature to use n-gram analysis [14, 45] on the dynamic trace or static data. But these approaches have the drawback of choosing the appropriate n. In our approach, we could use several values for n and then let the multiple kernel learning optimization weight the choices which it finds to be the most informative. This would be an interesting avenue for future research as we can view the same dataset in a variety of ways which would allow us to extract even more information for classification.

In this paper, we made the decision to use a standard squared exponential kernel (Equation 2) for the majority of the data sources. With our framework, we have the ability to incorporate more advanced kernels that have the ability to measure different aspects of similarity of the different data sources. These kernels can be based on random walks over the Markov chains, the eigen-structure of the graph Laplacians, the number of shortest paths in the graphs, etc. [20]. And like using different n-gram values, the multiple kernel learning optimization problem would find the kernel weights it deems most appropriate.

Although we have shown that acceptable performance can be achieved if certain time constraints are imposed by using solely static data sources, the full multiple kernel learning infrastructure, which includes the time it takes to collect the dynamic trace data, would be too resource intensive to be deployed on a normal user's system. There has been a rising interest in performing virus classification in the cloud [30, 31, 50] due to the constraints of users' resources. Our work would be perfect for a cloud-based solution as we could dedicate machines to collect the dynamic trace data and wouldn't have to burden the user's system with these expensive tracing tools.

As we explained in Section 3, we choose to use instruction/system call categorizations to reduce the size of the vertex set of the resulting Markov chains to avoid the *curse of dimensionality* with these models. Choosing an optimal instruction categorization is a non-trivial task. For example, using the coarse categorization with 8 categories on the disassembled data gave us ∼10% less classification accuracy than the categorization with 86 categories. There is also the possibility that different categorizations could prove to be better suited for different tasks. Clustering could be easier with a categorization that uses categories based on different instructions that are more likely to be used by different compilers. Or similarly, categories based on different instruction sets, such as SSE, MMX, AVX, and FMA, could be useful.

A unified data gathering infrastructure that would supply us with both our static and dynamic data sources would be an invaluable tool for our analysis. BitBlaze [42] aims to pro-

vide this need, incorporating static analysis techniques such as disassembled code generation, control flow graph generation, and other data flow analyses using the static analysis component of BitBlaze, Vine. The dynamic analysis component of BitBlaze, TEMU, allows for dynamic instruction traces and memory taint analysis, among other things. BitBlaze has been shown to provide reliable data that has been used to produce very accurate malware classification results [51]. It would be interesting to take advantage of all of the data sources provided by BitBlaze in our multiple kernel learning framework.

7. CONCLUSIONS

In this paper we 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 restricting malware classification to a single data source. We were able to achieve an accuracy of 98.07% on a dataset of 780 malware and 776 benign instances. We showed that while we had 16 false positives in this dataset, several of these were confirmed to be true positives. Our ROC curve analysis showed significant increases in performance when the data sources were combined, and also that acceptable performance can be achieved with just static sources in a resource constrained environment.

We demonstrated several interesting observations about our results, illustrating some of the pitfalls of both static analysis and dynamic analysis techniques. Namely that static data sources have problems classifying instances which were packed and/or had abnormal entropy values. We also showed the importance of having a dynamic tracing tool that is capable of evading detection from the malware so that we can get a truly representative sample of how the malware actually behaves in the wild.

8. REFERENCES

[1] Offensive Computing. http://www.offensivecomputing.net/, Accessed June 2011.

[2] Virus Total. http://www.virustotal.com/, Accessed October 2011.

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

[4] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-Based Malware Detection using Dynamic Analysis. *Journal in Computer Virology*, 7:247–258, 2011.

[5] Anubis. http://anubis.iseclab.org/, 2009.

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

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM, 2003.

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

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

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

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

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

[13] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[14] Jianyong Dai, Ratan Guha, and Joohan Lee. Efficient Virus Detection Using Dynamic Instruction Sequences. *Journal of Computers*, 4(5), 2009.

[15] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis Via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[16] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[17] R. Hettich and K. O. Kortanek. Semi-Infinite Programming: Theory, Methods, and Applications. *SIAM Review*, 35:380–429, September 1993.

[18] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, January 1998.

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

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

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

[22] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer Berlin / Heidelberg, 2006.

[23] G. Jacob L. Nataraj, S. Karthikeyan and B. Manjunath. Malware Images: Visualization and Automatic ClassiïňĄcation. In *Proceedings of VizSec*, 2011.

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

[25] 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*, pages 190–200, June 2005.

[26] Ulrike Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[27] Robert Lyda and James Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[28] Eitan Menahem, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Improving Malware Detection by Applying Multi-Inducer Ensemble. *Computational Statistics and Data Analysis*, 53(4):1483–1494, 2009.

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

[30] Jon Oberheide, Evan Cooke, and Farnam Jahanian. CloudAV: N-version Antivirus in the Network Cloud. In *Proceedings of the 17th Conference on Security Symposium*, pages 91–106, 2008.

[31] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized In-Cloud Security Services for Mobile Devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing, MobiVirt*, pages 31–35. ACM, 2008.

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

[33] IDA Pro. http://www.hex-rays.com/products/ida/index.shtml, 2012.

[34] Daniel Quist, Lorie Liebrock, and Joshua Neil. Improving Antivirus Accuracy with Hypervisor Assisted Analysis. *Journal in Computer Virology*, pages 1–11, 2010.

[35] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick DÃijssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 108–125. Springer Berlin / Heidelberg, 2008.

[36] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *22nd Annual Computer Security Applications Conference (ACSAC)*, pages 289–300, 2006.

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

[38] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[39] M. Shafiq, Syed Khayam, and Muddassar Farooq. Embedded Malware Detection Using Markov n-Grams. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 88–107. Springer Berlin / Heidelberg, 2008.

[40] Madhu Shankarapani, Subbu Ramamoorthy, Ram Movva, and Srinivas Mukkamala. Malware Detection Using Assembly and API Call Sequences. *Journal in Computer Virology*, 7(2):1–13, 2010.

[41] Nino Shervashidze, S. V. N. Vishwanathan, Tobias H. Petri, Kurt Mehlhorn, and Karsten M. Borgwardt. Efficient Graphlet Kernels for Large Graph Comparison. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 5, pages 488–495. CSAIL, 2009.

[42] 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. In *Information Systems Security*, volume 5352 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin / Heidelberg, 2008.

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

[44] Sören Sonnenburg, Gunnar Rätsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtěch Franc. The SHOGUN Machine Learning Toolbox. *The Journal of Machine Learning Research*, 99:1799–1802, August 2010.

[45] Salvatore Stolfo, Ke Wang, and Wei-Jen Li. Towards Stealthy Malware Detection. In *Malware Detection*, volume 27 of *Advances in Information Security*, pages 231–249. Springer US, 2007.

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

[47] Symantec. Internet Security Threat Report, Volume 16. White Paper, April 2011.

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

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

[50] Yanfang Ye, Tao Li, Shenghuo Zhu, Weiwei Zhuang, Egmen Tas, Umesh Gupta, and Melih Abdulhayoglu. Combining File Content and File Relations for Cloud Based Malware Detection. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011.

[51] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127. ACM, 2007.