# Automating Reverse Engineering with Machine Learning Techniques

Blake Anderson
Los Alamos National Laboratory
banderson@lanl.gov

Curtis Storlie
Los Alamos National Laboratory
storlie@lanl.gov

Micah Yates
Los Alamos National Laboratory
micah@lanl.gov

Aaron McPhall
Los Alamos National Laboratory
mcphall@lanl.gov

## ABSTRACT

Malware continues to be an ongoing threat, with millions of unique variants created every year. Unlike the majority of this malware, *Advanced Persistent Threat* (APT) malware is created to target a specific network or set of networks and has a precise objective, e.g. exfiltrating sensitive data. While **0**-day malware detectors are a good start, they do not help the reverse engineers better understand the threats attacking their networks. Understanding the behavior of malware is often a time sensitive task, and can take anywhere between several hours to several weeks. Our goal is to automate the task of identifying the general function of the subroutines in the function call graph of the program to aid the reverse engineers. Two approaches to model the subroutine labels are investigated, a multiclass Gaussian process and a multiclass support vector machine. The output of these methods is the probability that the subroutine belongs to a certain class of functionality (e.g., file I/O, exploit, etc.). Promising initial results, illustrating the efficacy of this method, are presented on a sample of 201 subroutines taken from two malicious families.

## 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; Gaussian Processes; Support Vector Machines


*AISec'14,* November 7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2666652.2666665.

## 1. INTRODUCTION

Millions of unique variants of malware are created every year, with McAfee labs cataloging more than 30 million new samples in the first quarter of 2014 [14]. The majority of this malware is created through simple modifications of known malware and is not intended to subvert sophisticated security procedures. On the other hand, *Advanced Persistent Threat* (APT) malware is created with the intention to attack a specific network or set of networks and has a precise objective, e.g. setting up a persistent beaconing mechanism or exfiltrating sensitive data. Because APT malware is much more alarming, most incident response teams of large networks have several reverse engineers on hand to deal with these threats.

A reverse engineer has the task of classifying the hundreds-to-thousands of individual subroutines of a program into the appropriate classes of functionality. With this information, they can then begin to decipher the intent of the program. But this is a very time consuming process, and can take anywhere from several hours to several weeks depending on the complexity of the program. At the same time, reversing APT is a time critical process, and understanding the extent of an attack is of paramount importance. And while **0**-day malware detectors are a good start, they do not help the reverse engineers better understand the threats attacking their networks.

In this paper, we have developed methods to aid the reverse engineer, specifically in the process of classifying individual subroutines. Figure 1 illustrates a function call graph visualized by the popular reverse engineering program, IDA Pro [10]. The program in Figure 1 is a relatively small and straightforward one.

The novel contribution this line of research makes is to automatically label each subroutine in the function call graph. The subroutine label is modeled using a multiclass Gaussian process or multiclass support vector machine giving the probability that the subroutine belongs to a certain class of functionality (e.g., file I/O, exploit, etc.). A multiview approach is used to construct the subroutine kernel (or similarity) matrix for use in the classification method. The different views include the instructions contained within each subroutine, the API calls contained within each subroutine, and the subroutine's neighbor information.

The process begins with a skilled reverse engineer labeling subroutines into general, predefined categories. The cate-

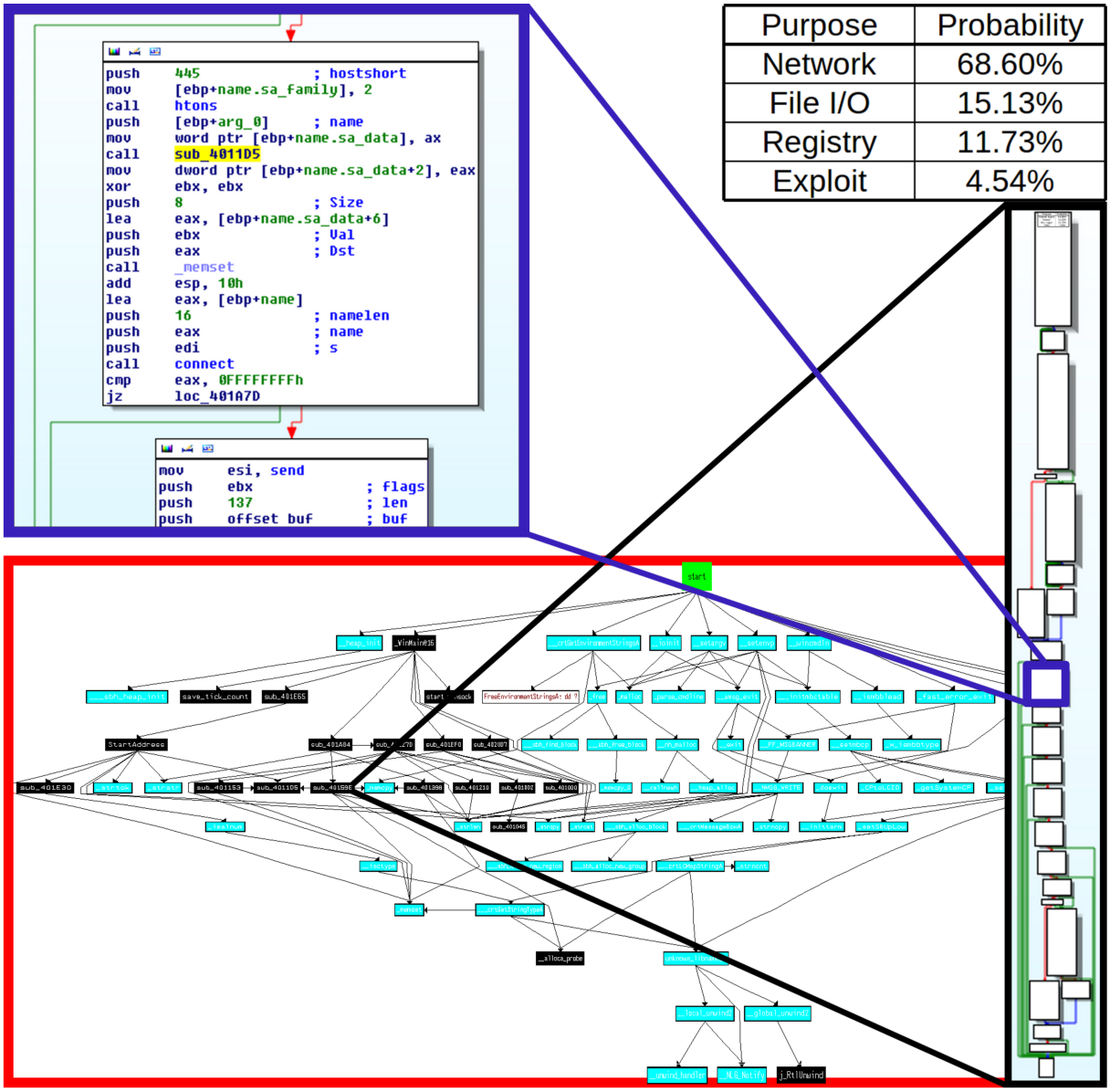



Figure 1: The function call graph as depicted by IDA. The novel contribution that our paper makes is to label each subroutine with the probability of its functionality.

gories currently being used include file I/O, process/thread, network, GUI, registry, and exploit. This information is used to train a classifier, and finally this classifier can be used to label new subroutines. Promising initial results, illustrating the efficacy of this method, are presented on a sample of 201 subroutines taken from two malicious families.

A web-based interface for this research is also presented. This interface allowed the reverse engineers to help guide the methodological development by offering an intuitive view into the classification results of this paper.

The rest of this paper is organized as follows. Section 2 gives an overview of the data and the views of the data used for classification. In Section 3, we show how to construct kernels for the different views and briefly describe the multiclass support vector machine and multiclass Gaussian process. The results are described in Section 4. In Section 5 we give related work, Section 6 outlines plans for future work, and finally, we conclude in Section 7.

## 2. DATA

The three views of the data, and their corresponding representations, are described below. The first two views, assembly instructions and API calls, have been studied extensively in the literature and have been shown to have strong discriminatory power in the malware versus benign classification problem. The neighbor information view has had less exposition, mainly due to subroutine classification being a novel problem. All of this data is collected from the IDA Pro [10] disassembly of the programs.

201 subroutines were collected and classified as one of six possible categories: file I/O, process/thread, network, GUI,

| | |
|---|---|
| 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 |
| push | edi |
| or | ebp, 0xFF |
| ... | ... |

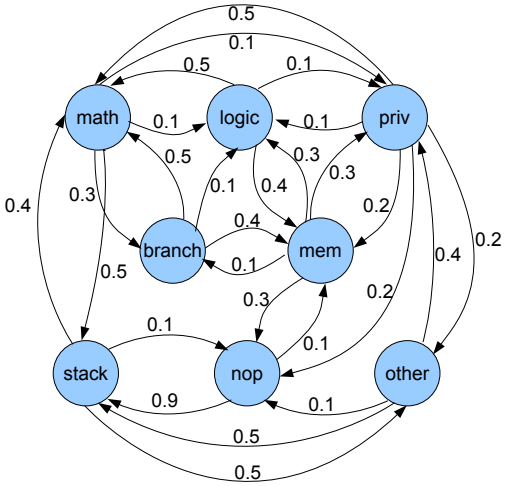



Figure 2: The left table shows an example of the assembly instructions contained within a subroutine. A hypothetical resulting Markov chain is shown on the right.

| Type | # examples |
|---|---|
| File I/O | 44 |
| Process/Thread | 42 |
| Network | 70 |
| GUI | 21 |
| Registry | 18 |
| Exploit | 6 |

Table 1: The number of each class of subroutines. The dataset contained 201 subroutines in total.

registry, or exploit. These subroutines came from 2 APT malware families and some random benign programs. The benign programs were mainly used to get more examples of the GUI category. There were 32 programs in total. The number of each class of subroutines is given in Table 1.

## 2.1 Instructions

Assembly instructions have had a lot of exposure in the literature [1, 4, 19, 23]. This is a fundamental view of subroutines, and we make use of it in this work. The assembly instructions are first categorized, i.e., there is a set number of classes of instructions and all instructions seen fall into one of the categories. In this work, 86 classes of instructions are used, which are based on the pydasm instruction types. Categorizations are used because there are a large number of semantically similar instructions (e.g. add and fadd), and this helps to limit the feature space to a manageable size.

There are several methods that can be used to represent the assembly instructions. The first method we experimented with was simply as sequences and then use a sequence alignment algorithm to compare the subroutines. This seems to be the most intuitive method, but yielded poor results and was orders of magnitude slower than the chosen method.

Because the sequence alignment method did not work as well as hoped, the instructions were modeled as a Markov chain. The instruction categories are the nodes of the Markov chain graph. In the Markov chain representation, the edge weight, $e_{ij}$, between vertices $i$ and $j$ corresponds to the transition probability from state $i$ to state $j$, therefore, the edge weights for edges originating at $v_i$ are required to sum to 1, $\sum_{i \rightsquigarrow j} e_{ij} = 1$. An $n \times n$ ($n = |V|$) adjacency matrix is used to represent the graph, where each entry in the matrix, $a_{ij} = e_{ij}$. An example is shown in Figure 2 on a simple eight category representation for ease of illustration.

## 2.2 API Calls

When a reverse engineer begins the process of understanding the functionality of a program, the API calls performed within the subroutine are highly informative. For instance, "wininet.dll" contains API calls that are exclusively used for network activity, and is a good indicator that the subroutine containing those calls is related to network functionality. The efficacy of API calls for the program classification problem has been shown in other work [8, 19]. Our dataset contains 791 unique API calls from 22 unique DLLs.

We tried several methods to encode the information from the API calls, notably using a feature vector of length 791 for each unique API call and a feature vector of length 22 for each unique DLL. Based on early results, we settled on using the feature vector of length 22, where each entry in the vector corresponds to the count of calls to that specific DLL within the subroutine.

## 2.3 Neighbor Information

Although API calls are clearly very informative, there exists a large number of subroutines that do not contain any API calls. This prompted the use of neighborhood information, with the assumption that the neighboring subroutines of subroutine $x$ will be likely to perform a similar function to the neighboring subroutines of subroutine $y$, given that $x$ and $y$ have the same label.

Two views are constructed with the neighbor information, the incoming and outgoing neighbor views. Similar to the API calls, a feature vector of length 22 (for the 22 unique DLLs) is used for each view. The incoming view is constructed by counting all unique DLLs in every incoming subroutine and setting the appropriate entry in the feature vector. For example, in Figure 3, the counts of the blue subroutines' DLLs would be used to construct the feature vector. The outgoing neighbor view is constructed analogously.

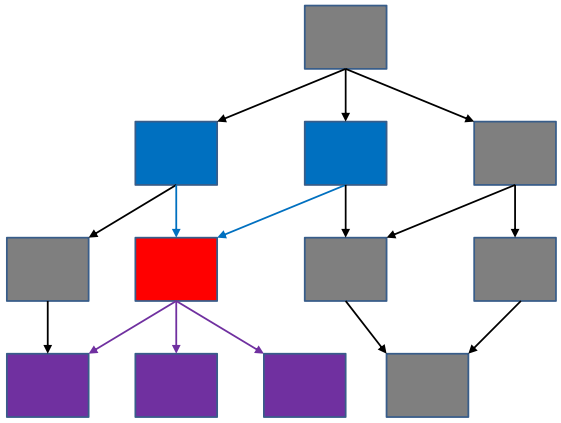


Figure 3: Illustration of the neighbor information used. For a given red node, the incoming blue nodes and the outgoing purple nodes are used.

# 3. METHODS

Kernel-based classifiers have been shown to perform well on a wide variety of tasks [16, 18]. For this work, support vector machines and Gaussian processes are used to classify the subroutines. These methods are related [16], and both rely on kernel matrices to perform their respective optimizations.

## 3.1 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 [18]. 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:

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

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 $\mathbf{x}' \in X$: $K(\mathbf{x}, \mathbf{x}') = K(\mathbf{x}', \mathbf{x})$) 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* [18], 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.

If each view from Section 2 is treated as a feature vector, a Gaussian kernel can be defined:

$$K(\mathbf{x}, \mathbf{x}') = \sigma^2 e^{-\lambda d(x, x')^2} \tag{2}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are the feature vectors for a specific view, $\sigma$ and $\lambda$ are the hyperparameters of the kernel function (determined through cross-validation or MCMC), and $d(\cdot, \cdot)$ is the distance between two examples. The Euclidean distance is used for $d(\cdot, \cdot)$.

## 3.2 Classification

**Support Vector Machine.**

A support vector machine searches for a hyperplane in the feature space that separates the points of the two classes with a maximal margin [6]. The hyperplane that is found by the SVM is a linear combination of the data instances, $x_i$, with weights, $\alpha_i$. It is important to note that only points close to the hyperplane will have non-zero $\alpha$'s. These points are called support vectors. Therefore, the goal in learning SVMs is to find the weight vector, $\alpha$, describing each data instance's contribution to the hyperplane. Using quadratic programming, the following optimization problem can be efficiently solved:

$$\max_{\alpha} \left( \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \right) \tag{3}$$

subject to the constraints:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{4}$$

$$0 \leq \alpha_i \leq C \tag{5}$$

Given $\alpha$ found in Equation 3, the decision function is defined as:

$$f(\mathbf{x}) = \operatorname{sgn} \left( \sum_{i}^{n} \alpha_i y_i K(\mathbf{x}, \mathbf{x}_i) \right) \tag{6}$$

which returns class +1 if the summation is $\geq 0$, and class -1 if the summation is $< 0$. The number of kernel computations in Equation 6 is decreased because many of the $\alpha$'s are zero.

To perform multiclass classification with the support vector machine, a one-versus-all strategy is used [9]. A classifier is trained for each class resulting in $l$ scores, where $l$ is the number of classes (in our case, 6). This list of scores can then be transformed into a multiclass probabilty estimate by standard methods [25].

**Gaussian Process.**

Gaussian processes are a popular probabilistic alternative to support vector machines for kernel learning. A Gaussian process can be completely specified by a mean function, $m$, and covariance (kernel) function, $K$, although the mean function is often taken to be zero without loss of generality [16]. They can be thought of as an infinite For multiclass classification, we use multinomial logistic Gaussian process regression [11]. For each class label, $l$, define

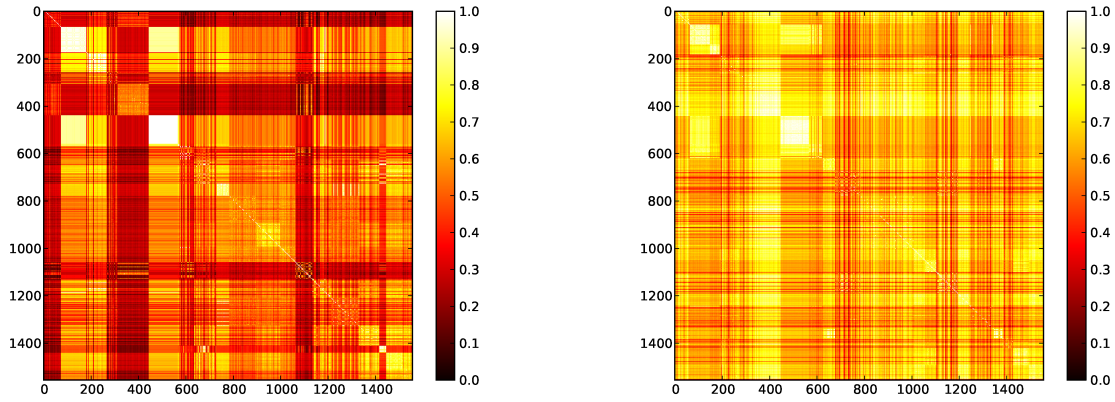
$$f_l \sim \mathrm{GP}(0, K) \tag{7}$$

to be an independent Gaussian process with covariance matrix $K$ and positive training examples belonging to class $l$. Let $p_l(x)$ be the probability of $x$ belonging to the $l$th class, and be defined as:

$$p_l(x) = \begin{cases} \frac{\exp f_l(x)}{1 + \sum_{l=1}^{L-1} \exp f_l(x)} & \text{for } l = 1, \ldots, L-1 \\ \frac{1}{1 + \sum_{l=1}^{L-1} \exp f_l(x)} & \text{for } l = L \end{cases} \tag{8}$$

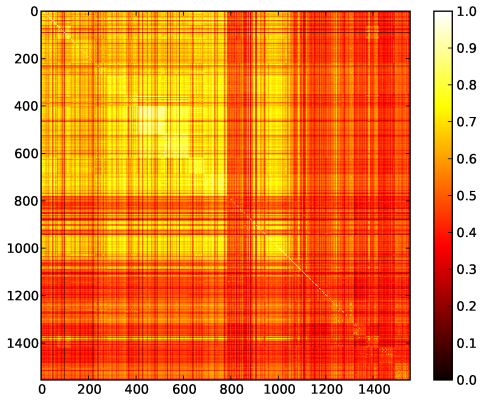
$p(x)$ is now a probability vector containing the probabilities of belonging to each of the $L$ classes.

The $f_l(x)$ are then conditioned on the training lables, $y$, and a posterior distribution is obtained for $f_l(x)$, and




(a) Instructions

(b) API Calls




(c) Combined Kernel

Figure 4: Kernel heatmaps showing the efficacy of combining multiple data views for cybersecurity. Combining views helps to "smooth" the space and results in better classification accuracy.

thus $p(x)$, at the training points. This is accomplished via Markov Chain Monte Carlo (MCMC) methods [20]. Prediction of new observations $x_*$ is then conducted by obtaining the predictive $f_l(x_*)$ by conditioning on the estimated $f_l$ corresponding to the training data.

$$f_l(x_*) = K_*(K + \sigma_n^2 I)^{-1} f_l \tag{9}$$

## 3.3 Combining Information

Combining multiple views has been shown to be advantageous for the malware versus benign classification problem [2]. For this reason, and the intuitive reasons outlined in Section 2, multiple views of subroutines are included in the models. For support vector machines, this can be accomplished with multiple kernel learning [21]. For the Gaussian processes approach used here, a new kernel can be defined over multiple views via product correlation (i.e., taking a product of the kernels for the individual views). Figure 4 gives an intuitive example of the benefits gained by using multiple views. Figure 4 (a) and Figure 4 (b) are distinct views of the instructions and API calls. Combining these views in a sense "smooths" the kernel space, allowing for better predictive accuracy.

**Support Vector Machine.**

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

$$K(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{M} \beta_i K_i(\mathbf{x}, \mathbf{x}') \tag{10}$$

is a convex combination of $M$ kernels with $\beta_i \geq 0$, where each kernel, $K_i$, uses a distinct set of features [21]. In the current case, each distinct set of features is a different view of the data (Section 2). The general outline of the algorithm is to first combine the kernels with $\beta_i = 1/M$, find $\alpha$, and then iteratively continue optimizing for $\beta$ and $\alpha$ until convergence. $\beta$ can be solved for efficiently using a semi-infinite linear program [21].

**Gaussian Process.**

Learning with multiple views in the Gaussian process is conceptually simpler in some respects, although it can be

| Method | Views | Accuracy | Average Probability of True |
|---|---|---|---|
| SVM | Instructions | .9403 | .8903 |
| GP | Instructions | .9701 | .8075 |
| SVM | API Calls | .8159 | .7857 |
| GP | API Calls | .8159 | .7609 |
| SVM | API Calls, Neighbor Information | .9403 | .8703 |
| GP | API Calls, Neighbor Information | .9154 | .8443 |
| SVM | Instructions, API Calls | **.9851** | **.9169** |
| GP | Instructions, API Calls | **.9851** | .8988 |

Table 2: 10-fold CV results on a dataset with 201 subroutines. These results demonstrate the benefit of including API calls within the classification model.

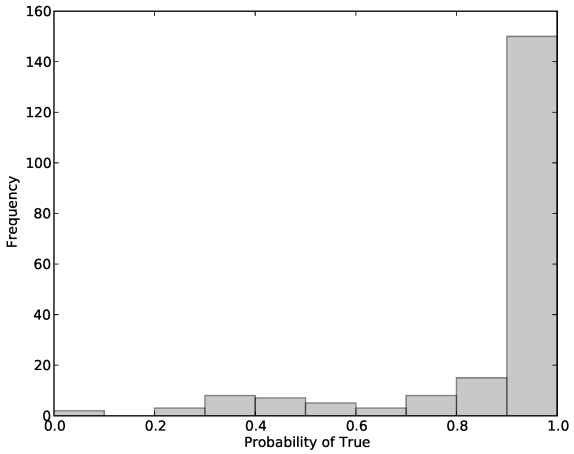



Figure 5: Histogram of the predicted probability of the true class with using only the instruction view.

more computationally demanding. To begin, modify Equation 2 to take the multiple views into account. This involves defining a distance function on each view, e.g. $d_j(x, x')^2$ for the $j$th view where in our case $d_j$ is the Euclidean distance. If there are $M$ views, the new multiview kernel is defined as:

$$K(x, x') = \sigma^2 e^{-\sum_j^M \lambda_j d_j(x,x')^2} \tag{11}$$

The $\lambda_j$'s now act as a way to combine the different metric spaces of the subroutines, similar to how the $\beta$ weight vector works in the multiple kernel learning method. The $\lambda_j$'s are now also optimized over within the same framework as the other parameters of the Gaussian process model using MCMC sampling [20]. Equation 10 could be used in place of Equation 11 within the same framework, but Equation 11 was found to produce better results.

# 4. RESULTS

In this section we will describe several experiments to test how well the methods described in Section 3 perform on the multiclass subroutine classification problem. 10-fold cross validation is used for all experiments unless stated otherwise. Within each fold, the parameters of the models are adjusted using 10-fold CV on the training data while the original hold-out is used for validation. We collected a dataset of 201 subroutines that are assigned one of the six labels from Table 1. Subroutines that perform multiple functions are excluded, and the problem of estimating subroutines belonging to multiple classes is left for future work. For the support vector machine, the Shogun machine learning toolbox [22] is used. The Bayesian multiclass logistic GP was custom coded for use here.

## 4.1 Classifying Subroutines

The first set of experiments examines the plausibility of our approach to classifying subroutines. Using just the instructions, we are able to achieve an accuracy of 94-97% with 10-fold cross validation. Table 2 shows the full results. The average probability of true is even more impressive than the raw accuracy. To reiterate, the methods of Section 3 return a probability vector of that subroutine belonging to each of the six classes. The "average probability of true" in Table 2 refers to the predicted probability of the true class averaged over all predictions. The average probability of true, using only the instructions, is .8075 for the Gaussian process and .8903 for the support vector machine. A histogram of these probability is given in Figure 5.

While we are not able to classify all subroutines correctly, the probability of the class can act as a pseudo-confidence for the reverse engineer looking at the results. And as Figure 5 demonstrates, the subroutine's true class is predicted at 90-100% roughly 75% of the time. In a sense, false predictions can be more harmful in this setting than the malware versus benign setting as a false prediction can give false leads to the reverse engineer, potentially wasting days of their time. By giving the probability vector of belonging to the different types of functionality, the reverse engineer can have some confidence of the predictions by focusing on the subroutines with 90-100% probability.

As mentioned in Section 2, API calls are very informative for a reverse engineer trying to understand a subroutine. API calls often clearly encode the type of functionality that a subroutine performs as the DLL from which the API call is imported from is usually homogeneous, i.e., contains functions that perform one type of functionality such as network. Unfortunately, API calls are not guaranteed to be in subroutines. In our dataset, only 163 out of the 201 subroutines contained API calls. Table 2 demonstrates the pitfall in only using API calls, as instruction-only classifiers are easily able to outperform API-only classifiers. But, including the subroutine's neighbors' API information significantly improved performance, .8159 → .9403 for the SVM.

Although API-only classifiers are outperformed by instruction-only classifiers, including API calls significantly improves performance, giving 98.51% classification accuracy. Furthermore, the average probability of true is increased for

| Method | Views | Accuracy | Average Probability of True |
|---|---|---|---|
| SVM | Instructions | .8000 | .7716 |
| GP | Instructions | .8800 | .5750 |
| SVM | API Calls | .5000 | .5067 |
| GP | API Calls | .4200 | .4132 |
| SVM | Instructions, API Calls | **.9400** | .7955 |
| GP | Instructions, API Calls | .9000 | .6552 |
| SVM | API Calls, Neighbor Information | .8400 | .6804 |
| GP | API Calls, Neighbor Information | .8400 | 7228 |
| SVM | Instructions, API Calls, Neighbor Information | **.9400** | **.8112** |
| GP | Instructions, API Calls, Neighbor Information | **.9400** | .7382 |

Table 3: In this problem, there are 151 training subroutines taken from one family and one member of another family. There are 50 subroutines in the test set, which are taken from the remaining members of the second family. These results indicate that including neighbor information may help the classification methods.

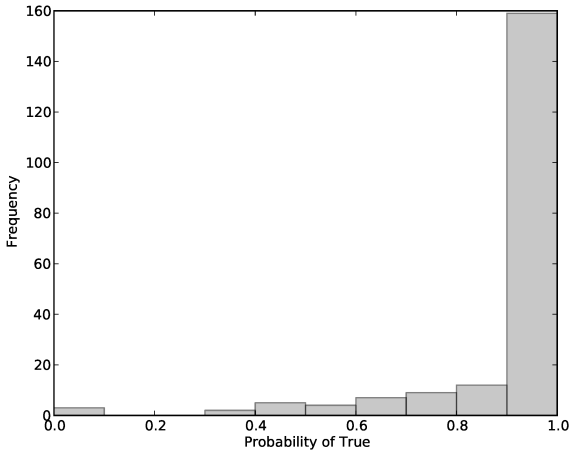



Figure 6: Histogram of the predicted probability of the true class with using the instruction view and the API call view.

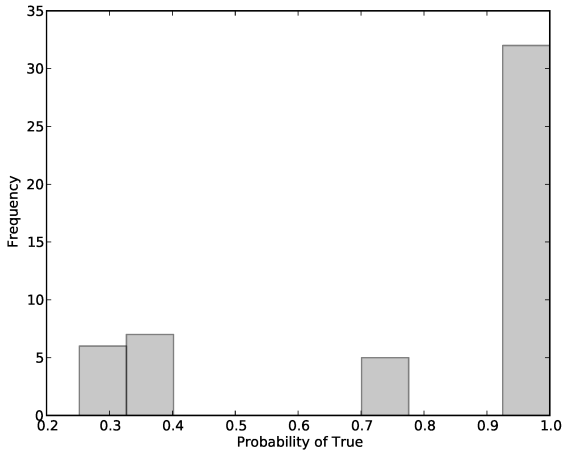



Figure 7: Histogram of the predicted probability of the true class with using the instruction view, the API call view, and the neighbor view. This is for the problem of classifying a family based on one example.

both the support vector machine and the Gaussian process, the increase for the Gaussian process being quite substantial (.8075 → .8988). The histogram for the predicted probabilities of the true class for the support vector machine classifier is given in Figure 6.

## 4.2 Testing on a New Family

One of the problems with developing methods with a limited dataset is that it is difficult to know if the improvements seen on the current dataset will generalize to much larger datasets. This is especially true for our problem, as we currently only have 201 labeled subroutines and have achieved a relatively high accuracy (98.51%) with 10-fold cross validation. To make the problem more challenging, we created a new experiment where the training data includes all of the subroutines from the first family of malware, the random benign files, and *one* sample from the second family. The testing set is composed of the subroutines from the remaining samples of the second family of malware.

In addition to allowing for new methodological developments, this is a more realistic test. APT malware is usually developed in campaigns. When a new malware sample attacks a network, a reverse engineer has most likely spent time on another sample from that family. Therefore, at least one member of that family's subroutines would be in the training dataset.

Table 3 lists the results for this new experiment. Because this is a more difficult experiment, both accuracy and the average probability of the true class suffer compared to the results of Table 2. With this harder experiment, it is clear that including the neighbor information helps the results. For the Gaussian process, including neighbor information pushes the accuracy from 90% to 94%, and the average probability of the true class from .6552 to .7382. The histogram for the predicted probability of the true class is shown in Figure 7. As our dataset is still relatively small, it is difficult to know whether the neighbor's DLLs will continue to be informative as new data is collected, but the results of Table 3 indicate that it should be helpful and is worth further investigation.

## 4.3 Prototype System

Developing methods for cybersecurity applications, or any application for that matter, should not be done in a vac-

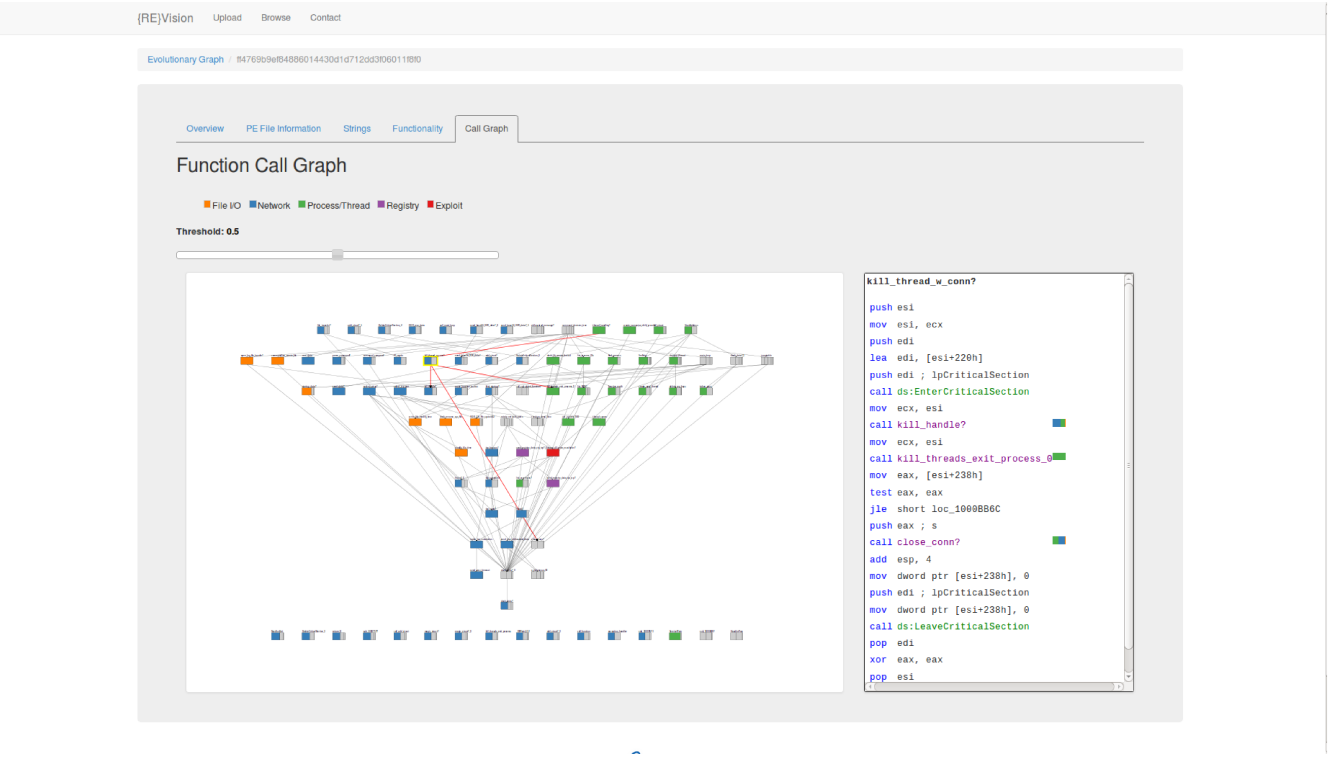



Figure 8: A screenshot from a live prototype developed in tandem with the methods of Section 3.

uum. To make sure the developed methods stayed relevant to the reverse engineer, a prototype user interface was simultaneously created to display the results. This system is a web-based application using d3.js for the graph layout and a python backend for the classification. This allowed for constant feedback from the reverse engineer. A screenshot of the system is given in Figure 8.

The reverse engineer was able to check the results and make sure the algorithms developed were performing in a consistent way. All the subroutines of a typical program ($\sim$ 400-500) can be classified in around 5-10 seconds, making this tool very useful in a online setting. This tool also allowed for some interesting real-world results. The algorithm labeled one subroutine, not in any training set, as being .55 network and .4 process/thread. After investigating the subroutine, the reverse engineer came to the conclusion that this subroutine was looking for threads with an active internet connection and killing them. Observations such as these make it clear that not all subroutines are "pure", and although it is left for future work, having the ability for subroutines to be placed into multiple classes in necessary for a robust system. These types of developments would not have happened unless we were closely interacting with the incident responders.

## 5. RELATED WORK

Because subroutine classification is a novel problem, there is no direct related work. But the methods presented in this paper have been heavily influenced by other papers in the literature using machine learning techniques on the malware versus benign classification problem [3, 7, 12, 17]. Notably, the features used in our classification methods, the instructions and API calls of each subroutine, have had a lot of exposure in the literature [1, 4, 8, 19, 23]. The main difference between this work and what has been published being that a data instance is not an entire program, but rather a single subroutine.

There has been some investigation into using the control flow and function call graphs to help in detecting polymorphic malware [5, 12]. But these papers use the structural information of these graphs as features to then classify programs as either malicious or benign. On the other hand, we are less interested in the program as a whole, and more interested with labeling the subroutines themselves with their functionality.

Finally, as shown in Section 4, combining the different views of the subroutine (instructions, API calls, and neighbor information) significantly helps in the multiclass classification problem of this paper. Combining multiple views of malware has been examined before in the literature [2, 13, 15]. Just as before, these papers looked at different views of the entire program, but we examine different views of individual subroutines.

## 6. FUTURE WORK

As stated previously, acquiring data for this research is very expensive and time consuming, and this is the biggest factor hindering further methodological developments. This is one reason why the user interface of Figure 8 was developed. We hope that once there is a critical mass of labeled subroutine data, the tools of this paper will begin to significantly speed up the reverse engineering process. This speed up will allow us to collect more data from the reverse engineers, making our methods more effective. This is the

synergy we are seeking between our methods/tools and the reverse engineers who use them.

The first attempt at constructing the similarity matrices for use in the kernel classifiers was based on sequence alignment [19, 24]. While sequence alignment between two programs may be easily confused by simple reordering of basic blocks and subroutines, we believed that the subroutines would be homogeneous enough to avoid these problems. In our initial tests, we found that the Markov chain representation performed slightly better with respect to accuracy, much better in terms of the predicted probabilities of the true class, and was orders of magnitude faster. Along these lines, it would be foolish not to continue investigating the subroutine metric space to find better, more reliable kernels. The most fruitful direction is most likely going to be finding new ways of incorporating the graph structure and neighborhood information, a direction we are pursuing.

As mentioned in Section 4, subroutines are not always functionally "pure", i.e., a subroutine can perform multiple functions. We have shown that we can classify pure subroutines with high accuracy. It would be interesting to design methods that can robustly classify subroutines into more than one class.

Along these lines, classifying subroutines into general categories can be seen as a first step to classifying groups of subroutines, or a sub-graph of the function call graph, into more specific tasks. These tasks could include things such as data exfiltration or keylogging. These complex tasks are often comprised of more than one subroutine. We are looking at ways to cluster the function call graph using graph structure and the general labels found in this paper to find the more specific task labels.

Assuming that we can accurately identify the specific tasks of a program, such as data exfiltration, keylogging, etc., building classifiers based on this information for the malware/benign problem for an overall program would seem like a natural next step. One would expect malware to perform several malicious tasks, but benign programs should, for the most part, be free of these tasks.

To get the methods of this paper adopted for mainstream use, a new user interface will need to be developed that can be easily integrated into the workflow of a reverse engineer. The reverse engineers of this project have been very willing to test our prototype system, but it is highly unlikely that all reverse engineers will be so willing to learn new tools. A future plan is to integrate this line of research with the highly used program, IDA Pro [10]. Creating a plugin for IDA Pro that can automatically label the subroutines would be far less disruptive and much more likely to be adopted than a new web-based application.

# 7. CONCLUSIONS

Classifying programs as either benign or malicious is an important first step to stopping advanced APT malware, but a simple binary decision does not give the analysts the information they need to properly assess the threat. In this paper, we presented a first step in helping reverse engineers understand a malicious program more quickly by classifying the subroutines of the function call graph into six general categories: file I/O, process/thread, network, GUI, registry, and exploit. Support vector machines and Gaussian processes were used for the classification process. We showed that we can achieve high accuracy (98.51%) on a set of 201 labeled subroutines.

# 8. REFERENCES


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

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

[3] 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.

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

[5] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting Self-Mutating Malware using Control-Flow Graph Matching. In *Detection of Intrusions and Malware and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 129–143. Springer Berlin / Heidelberg, 2006.

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

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

[8] 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.

[9] Chih-Wei Hsu and Chih-Jen Lin. A Comparison of Methods for Multiclass Support Vector Machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, 2002.

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

[11] Balaji Krishnapuram, Lawrence Carin, Mario AT Figueiredo, and Alexander J Hartemink. Sparse Multinomial Logistic Regression: Fast Algorithms and Generalization Bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6):957–968, 2005.

[12] 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.

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

[14] McAfee. McAfee Threat Report, First Quarter, June Accessed 15 July 2014.

http://www.mcafee.com/sg/resources/reports/rp-quarterly-threat-q1-2014.pdf.

[15] 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.

[16] Carl Edward Rasmussen and Christopher K.I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[17] 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.

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

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

[20] Qi-Man Shao and Joseph G Ibrahim. *Monte Carlo Methods in Bayesian Computation*. Springer Series in Statistics, New York, 2000.

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

[22] 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*, 11:1799–1802, 2010.

[23] Curtis Storlie, Blake Anderson, Scott Vander Wiel, Daniel Quist, Curtis Hash, and Nathan Brown. Stochastic Identification of Malware with Dynamic Traces. *The Annals of Applied Statistics*, 8(1):1–18, 2014.

[24] J-Y Xu, Andrew H Sung, Patrick Chavez, and Srinivas Mukkamala. Polymorphic Malicious Executable Scanner by API Sequence Analysis. In *Fourth International Conference on Hybrid Intelligent Systems (HIS)*, pages 378–383. IEEE, 2004.

[25] Bianca Zadrozny and Charles Elkan. Transforming Classifier Scores into Accurate Multiclass Probability Estimates. *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 694–699, 2002.