Leveraging String Kernels for Malware Detection

Jonas Pfoh, Christian Schneider, and Claudia Eckert

Technische Universität München
Computer Science Department
Munich, Germany
{pfoh,schneidc,eckertc}@in.tum.de

Abstract. Signature-based malware detection will always be a step behind as novel malware cannot be detected. On the other hand, machine learning-based methods are capable of detecting novel malware but classification is frequently done in an offline or batched manner and is often associated with time overheads that make it impractical. We propose an approach that bridges this gap. This approach makes use of a support vector machine (SVM) to classify system call traces. In contrast to other methods that use system call traces for malware detection, our approach makes use of a string kernel to make better use of the sequential information inherent in a system call trace. By classifying system call traces in small sections and keeping a moving average over the probability estimates produced by the SVM, our approach is capable of detecting malicious behavior online and achieves great accuracy.

Keywords: Security, Machine Learning, Malware Detection, System Calls

1 Introduction

Detecting malware is an ever present challenge in the field of security. Traditionally, malware detection makes use of signature-based methods. That is, known malware samples are analyzed to create a repository of signatures which are then matched against a static object to determine whether the particular object is infected with malware. While this approach is straightforward, it has two fundamental issues. The first stems from the static nature of the analysis. A static analysis indicates that it is performed on an inert object, that is, an object that is not being executed or in any other way active. Malware authors take advantage of this fact by obfuscating the inert object in such a way that it no longer matches any of the the signatures in the repository. However, when executed, the actions of the active process prove malicious. This may be achieved by simple packing and unpacking of the malicious portions of the object or by more advanced polymorphism techniques.

The second issue with such an approach is a result of its reliance on signatures. These signatures must be generated prior to a successful match, which makes such an approach disadvantageous in situations where no prior sample existed for signature generation. For example, novel malware that makes use of so-called "0-day" exploits (exploits which have not yet been seen in the wild) are difficult to detect with a signature-based method. To address these issues, dynamic machine learning-based analysis has often been considered in various forms [1–5].

In a dynamic analysis, the behavior of the malware is analyzed rather than the inert object. This circumvents traditional code obfuscation as the behavior remains malicious and it is this behavior that is analyzed. Obfuscating behavior becomes much more difficult as the malicious act must be carried out in some form. That is, one can attempt to conceal their intentions, but once the malicious act is carried out, this behavior is ideally observable and can be acted upon. Furthermore, machine learning techniques lend themselves well to malware detection as such techniques make an attempt to generalize and learn the features of malware that differentiate them from benign software. This can then also be applied to novel malware, thus countering the threat of 0-day exploits.

While dynamic approaches show much promise they are not immune to shortcomings of their own. While obfuscating behavior is more difficult than obfuscating code, it is not impossible. Depending on how the behavior of a process is modeled, dynamic analysis is generally vulnerable to a class of attacks called mimicry attacks [6, 7]. This class of attack attempts to "act benign" while secretly carrying out some malicious action. Additionally, the large time complexity combined with the massive amount of data that needs to be classified often makes a practical solution difficult.

In this paper we model process behavior though system call traces and present a practical machine learning-based method for malware detection. Specifically, we make use of a support vector machine (SVM) in combination with a string kernel function called a string subsequence kernel (SSK) [8]. This kernel function has properties that lend themselves well to malware detection in spite of mimicry attacks. Additionally, we present a novel method for classifying the behavior of processes in an online manner. Finally, we present an evaluation of our approach which includes several comparisons with other machine learning methods for malware detection.

2 Background

For the classification of system call traces, we make use of support vector machines (SVMs) [9]. SVMs are a maximal margin hyperplane classifiers. That is, given a training set $X = \{(\mathbf{x_m}, y_m)\}_{m=1}^{M}$, where $\mathbf{x_m}$ is a training vector and y_m is the associated class $+1$ or -1, the SVM identifies the hyperplane for which the separation between the most relevant training vectors (i. e., the support vectors) and the hyperplane is maximized, then classifies new vectors based on their relation to this hyperplane. The hyperplane is represented by a weight vector $\mathbf{w} \in \mathbb{R}^D$ and a variable $b \in \mathbb{R}$ and is formally defined for some $C > 0$ in the following optimization problem:

$$\begin{aligned} \underset{\mathbf{w},\xi,b}{\text{minimize}} \quad & \frac{\|\mathbf{w}\|^2}{2} + \frac{C}{M}\sum_{m=1}^{M}\xi_m \\ \text{subject to} \quad & y_m(\langle \mathbf{w}, \mathbf{x}_m \rangle + b) \geq 1 - \xi_m, m = 1, \cdots, M \end{aligned} \tag{1}$$

where ξ_i represents slack variables that are responsible for preventing an overfitting of the model.

By introducing a Lagrangian with multipliers $\alpha_m \geq 0$, the training phase determines which training vectors will become support vectors. Then, the classification occurs by comparing the test vector to each support vector and measuring the similarity. The decision function f is formally defined as:

$$f(\mathbf{x}) = \text{sgn}(g(\mathbf{x})) \tag{2}$$

where

$$g(\mathbf{x}) = \sum_{m=1}^{M} y_m \alpha_m \langle \mathbf{x}, \mathbf{x}_m \rangle + b \tag{3}$$

Here the dot product ($\langle \mathbf{a}, \mathbf{b} \rangle$) plays the role of the kernel function, which measures the similarity between the two vectors. For simple geometric classification, a dot product may suffice as a measure of similarity. However, for detecting malware through system call traces, a more complex kernel function is necessary. This kernel function must be carefully chosen for a given domain and is discussed in further detail in Section 2.1.

While SVMs produce a binary result as seen in (2), it is often beneficial to work with a posterior probability $P(y = 1|g(x))$ based on $g(x)$ defined in (3). Such a posterior probability is especially helpful when the output is to be combined with other factors to reach a final decision.

Several methods for probability estimation have been proposed. We make use of a method proposed by Platt [10]. Platt's method estimates the posterior probability by using the following sigmoid function:

$$P(y = 1|g(x)) = \frac{1}{1 + e^{Ag(x)+B}} \tag{4}$$

where A and B are found by minimizing the negative logarithmic likelihood of the training data.

2.1 Kernel Function

In looking for a kernel function, we begin by examining the nature of the input itself. The input consists of a string (i. e., sequence) of system call numbers. For the language processing domain, string kernels were introduced to classify texts or strings [8]. In essence, our input is very similar, though instead of classifying strings over the roman alphabet, for example, we are interested in classifying strings over the alphabet of all system calls. That is, we define our alphabet,

Σ, as all possible system calls and a string is a sequence $s \in \Sigma^*$ of letters (i. e., system calls). Based on this similarity, we choose a string kernel for our method.

Specifically, we choose to use the string subsequence kernel (SSK) [8]. This kernel measures the similarity between inputs by considering the number of common *subsequences*. A subsequence allows for non-matching, interior letters between its elements, though the kernel penalizes the similarity as this number of interior letters increases. For example, the string ABC would clearly match on the string ABC, but it would also match on the string $AaaaBbbbCccc$, though with a lower similarity measure due to the interior aaa and bbb. This property of the kernel is especially attractive as a sequence of system calls may contain interior system calls that might be irrelevant to the malicious nature of the sequence.

The SSK is formally defined as:

$$k(s,t) = \sum_{u \in \Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})} \tag{5}$$

where n is the size of the subsequence and $\lambda \in (0,1)$ is the decay factor used to weight the contribution of the match based on the number of interior letters. The notation $u = s[\mathbf{i}]$ denotes that u is a subsequence of s for which there exist indices $\mathbf{i} = (i_1, \ldots, i_{|u|})$, with $1 \leq i_1 < \cdots < i_{|u|} \leq |s|$, such that $u_j = s_{i_j}$, for $j = 1, \ldots, |u|$. Finally, $l(\mathbf{i})$ represents the length of the subsequence including interior letters.

3 Method

We begin this section by arguing for system call traces as a model for process behavior. We present the observation that a process in complete isolation cannot perform any malicious action on the rest of the system. Hence, in order for a process to act maliciously it must interact in some manner with the rest of the system and if the isolation mechanism in place is sound, this interaction must take place through the interface provided by the operating system (OS) (i. e., system calls). System calls are necessary to perform actions such as file operation, network communication, inter-process communication, etc. As a result of the above observation, system call traces are often used to model process behavior [1, 3, 4].

However, previous approaches often make use of polynomial kernels or other methods that do not fully consider the sequence of the system calls [1, 3, 5]. That is, in the most trivial case, the number of times that a system call occurs in the trace is taken into account without considering the order of the system calls. This is most likely due to the fact that string kernels incur a massive time overhead when used with large amounts of data. However, if one can mitigate the increased time overhead, an approach that considers sequential data has the potential to produce very high accuracy rates. Intuitively, considering sequential data is logical. If one were to manually analyze a system call trace, one would consider the order of the system calls in addition to which system call is being executed. In an effort to baseline the time overhead, we began training the SSK with our raw data and broke the test off after two months of running with no result in sight. So with practical analysis as a goal, clearly this time overhead must be addressed.

We address the time overhead of the SSK with the observation that if we are able to classify a process by updating an interim classification value and making a decision before the process has finished, we inherently address online classification while reducing the time overhead by not having to analyze the entire system call trace.

Training. To prepare the training data, we iterate over each individual system call trace and extract contiguous sub-traces of size S starting at random points within the traces. We iterate over all the training traces several times in order to get several sub-traces from each original trace. These size S sub-traces become our training set. We do this for two reasons. First, training the SSK with circa 2000 full-length traces, some of which may contain hundreds of thousands of system calls, takes months even on modern hardware. Second, classifying against a support vector with hundreds of thousands of system calls is equally time consuming. The clear concern is that some of these sub-traces may not be indicative of the class they belong to because they represent a relatively small fraction of the entire trace. However, with enough sub-traces we will eventually collect some that are indicative of the class they belong to. The beauty of a SVM is that it will decide which of the sub-traces to use as support vectors (hopefully those indicative of the training class) and which to disregard.

Classification. The classification works by sliding a window of size S over the system call trace that is to be classified. This sliding window moves forward by $S/2$ elements in the trace for each iteration. Then, for each iteration, probability estimates are taken using Platt's method [10] as described in Section 2 and factored into a cumulative moving average for each class. If we let $p_i = P(y = 1|x_i)$ represent the probability estimate as approximated by (4) for an iteration i, we represent the cumulative moving average after iteration i as:

$$U_i = \frac{p_1 + \cdots + p_i}{i} \tag{6}$$

In addition to calculating the cumulative moving average, we also experimented with a simple moving average of the probability estimates. This is a similar method, though instead of considering all previous window iterations, a simple moving average only considers the last y window iterations in the average (where y may be arbitrarily set). Formally,

$$S_i = \frac{p_{i-y} + \cdots + p_i}{y} \tag{7}$$

where $i \geq y$.

We continue our classification by defining two thresholds $T_1 \in [0.5, 1]$ and $T_{-1} \in [0.5, 1]$. These thresholds are compared to U_i and $1 - U_i$, respectively and if either threshold is exceeded, the classification ends by predicting the class represented by the exceeded threshold. Formally, the decision function is represented as follows:

$$D_i = \begin{cases} 1 & \text{if } U_i > T_1, \\ -1 & \text{if } 1 - U_i > T_{-1}, \\ D_{i+1} & \text{else} \end{cases} \tag{8}$$

Clearly, $U_i > 0.5 \wedge (1 - U_i) > 0.5$ can never be true if $U_i \in [0, 1]$, therefore if $T_1 \in [0.5, 1]$ and $T_{-1} \in [0.5, 1]$, only one single case of the decision function will ever be true for a given iteration. For practicality, if the cumulative moving average never exceeds either threshold and there are no more system calls in the trace, the decision function simply predicts 1 if $U_i > 0.5$ or it predicts -1 otherwise.

4 Evaluation

In this section we present the results of our experiments when testing our SVM-based method for malware detection on real-world data

4.1 Data Collection

We ran this experiment on two sets of sample traces collected from Windows XP SP3. We chose Windows XP as it is a popular commercial OS and numerous malware samples are available for this platform.

The first set of system call traces was collected using Nitro [11], a VMI-based system for system call tracing and trapping. This dataset includes 1943 system call traces of malicious samples taken from VX Heavens[1] and 285 system call traces of benign samples taken from a default Windows XP installation and selected installations of well-known, trusted applications.

The second set of traces is taken from a level slightly above system calls. Windows XP wraps its system calls in APIs that it provides to programmers through system libraries. While these traces are technically at a level slightly above the system calls themselves, they serve the same purpose and demonstrate that our method works at both levels. This dataset was collected by hooking these API functions and was first used by Xiao and Stibor [4]. It consists of 2176 API call traces of malicious samples and 161 API call traces of benign samples.

We chose to introduce the second independent data set for two reasons. First, the second data set makes use of API call traces rather than system call traces directly. This gives us a chance to observe the accuracy of our approach for system calls as well as API calls. The second and perhaps more important reason

[1] `http://www.vxheavens.com`

for including a second data set is to confirm that our method also works on an independent data set that was not collected by us. This strengthens the credibility of our approach as it allows us to present results based on data that others have previously used in similar experiments. In fact, we directly compare the results of our method with that of Xiao and Stibor in Section 5.

One might notice that the amount of benign and malicious samples are somewhat imbalanced. We address this by making use of cross-validation as described in Section 4.2 and by reporting the false positive rate in addition to the recall as seen in Section 4.3.

4.2 Setup

We begin by preparing the training set as described in Section 3. That is, we iterate over the full system call traces and extract random contiguous subsequence of size S (in our experiments $S = 100$). We iterate a number of times as to have 2000 random contiguous subsequences in each training set. We take care that the training samples can be traced back to their original trace as to make sure we properly perform a two-fold cross-validation. That is we are careful that, when testing, we train the SVM with samples that do not come from traces in the testing set. Making use of cross-validation allows us to "simulate" the detection of 0-day malware as the classification is performed on data that was not seen during the training phase.

With the data collected and prepared, we make use of LIBSVM [12] along with a provided string kernel extension to perform both training and classification as described in Section 3. Since the SSK is not implemented in LIBSVM or the string kernel extension, we incorporated the SSK implementation proposed by Herbrich [13]. This implementation had to be further modified such that it accepts an input over an integer alphabet as opposed to a roman letter alphabet used in text classification. It is also important to note that LIBSVM calculates probability estimates by making use of an improved algorithm for minimizing the negative logarithmic likelihood proposed by Lin et al. [14].

With these tools and the data prepared, we set up the experiment as described in Section 3. We also found that it was necessary to factor several values into the moving average before checking either threshold. This allows the moving average to factor in the first several iterations before a decision is made. For this reason we always factor the probability estimates for the first 10 iterations in our moving average before we begin considering the thresholds.

4.3 Results

For each experiment, P represents the number of positive (malicious) samples and N represents the number of negative (benign) samples. The variables associated with tuning our detection mechanism (n, λ, T_{-1}, T_1) are experimentally optimized. A discussion of each of these variables can be found below. Then, as each experiment runs, we collect the number of correctly and incorrectly classified results as true positives (TP), true negatives (TN), false positives (FP),

Test Num.	n	λ	T_{-1}	T_1	Avg.	TP	FP	TN	FN	Average Iterations
1	**3**	**0.50**	**0.50**	**0.75**	**CMA**	**1929**	**11**	**274**	**14**	**13.2608**
2	4	0.50	0.50	0.75	CMA	1910	15	270	33	29.4753
3	5	0.50	0.50	0.75	CMA	1903	20	265	40	19.7244
4	3	0.25	0.50	0.75	CMA	1702	17	268	241	209.4165
5	3	0.75	0.50	0.75	CMA	1902	14	271	41	120.6194
6	3	0.40	0.50	0.75	CMA	1919	12	273	24	14.1831
7	3	0.60	0.50	0.75	CMA	1929	14	271	14	22.4475
8	3	0.50	0.50	0.50	CMA	1941	35	250	2	10.8406
9	3	0.50	0.50	0.75	SMA	1869	7	278	74	11.3321
10	3	0.50	1.00	0.90	SMA	1906	48	237	37	722.3012

Test Num.	FP Rate $\frac{FP}{N}$	Recall $\frac{TP}{P}$	Precision $\frac{TP}{TP+FP}$	Accuracy $\frac{TP+TN}{P+N}$	F-Measure $\frac{2}{\frac{1}{\text{Precision}}+\frac{1}{\text{Recall}}}$
1	**0.0386**	**0.9928**	**0.9943**	**0.9888**	**0.9936**
2	0.0526	0.9830	0.9922	0.9785	0.9876
3	0.0702	0.9794	0.9896	0.9731	0.9845
4	0.0596	0.8760	0.9901	0.8842	0.9295
5	0.0491	0.9789	0.9927	0.9753	0.9857
6	0.0421	0.9876	0.9938	0.9838	0.9907
7	0.0491	0.9928	0.9928	0.9874	0.9928
8	0.1228	0.9990	0.9823	0.9834	0.9906
9	0.0246	0.9619	0.9963	0.9636	0.9788
10	0.1684	0.9810	0.9754	0.9618	0.9782

Table 1: Experimental results for *system call trace dataset* ($P = 1943$, $N = 285$)

and false negatives (FN). With this information we calculate the classification measures presented in Table 1 and Table 2 and discussed below. Finally, we also present the number of average iterations (the average number of times the SVM classifier had to be called until either threshold was met) and the type of moving average (CMA = cumulative moving average, SMA = simple moving average) used in each experiment.

We began by considering the threshold values. These values are quite important as they most directly affect the average number of iterations it takes to make a decision. As the thresholds rise, so does the average number of iterations in general. However, if a threshold is too low, the number of false positives and/or negatives rises. In addition, we noticed that our SVM was much more sensitive to malicious samples than it was to benign samples. That is, the average of the malicious probability estimates rose much more quickly to 1 for malicious samples than the average of the benign probability estimates for benign samples. We speculate that this is due to the fact that the traces from the malicious samples are more similar to one another than the traces from the benign samples. That is, the diversity among the benign samples is higher due to the fact that while malicious behavior is generally easier to define, benign behavior is simply "everything else".

Test Num.	n	λ	T_{-1}	T_1	Avg.	TP	FP	TN	FN	Average Iterations
1	**3**	**0.6**	**0.5**	**0.75**	**CMA**	**2029**	**19**	**142**	**147**	**37.2174**
2	3	0.5	0.5	0.75	CMA	1971	19	142	205	31.3479
3	3	0.4	0.5	0.75	CMA	1699	15	146	477	32.8015
4	4	0.5	0.5	0.75	CMA	1660	20	141	516	29.1656
5	5	0.5	0.5	0.75	CMA	1697	18	143	479	32.2657
6	3	0.5	0.5	0.75	SMA	1456	16	145	720	14.1694
7	3	0.5	0.7	0.9	SMA	1961	21	140	215	51.3697

Test Num.	FP Rate $\frac{FP}{N}$	Recall $\frac{TP}{P}$	Precision $\frac{TP}{TP+FP}$	Accuracy $\frac{TP+TN}{P+N}$	F-Measure $\frac{2}{\frac{1}{\text{Precision}}+\frac{1}{\text{Recall}}}$
1	**0.1180**	**0.9324**	**0.9907**	**0.9290**	**0.9607**
2	0.1180	0.9057	0.9905	0.9042	0.9462
3	0.0932	0.7808	0.9912	0.7895	0.8735
4	0.1242	0.7629	0.9881	0.7706	0.8610
5	0.1118	0.7799	0.9895	0.7873	0.8723
6	0.0993	0.6691	0.9891	0.6851	0.7982
7	0.1304	0.9012	0.9894	0.8990	0.9432

Table 2: Experiment results for *API call trace dataset* ($P = 2176$, $N = 161$)

We next considered the size n of the subsequence that the kernel function looks for in the traces being compared. What we observed is that as we increased n from the initial value of 3, the classification measures for both datasets became worse. This may seem somewhat counterintuitive. However, n is very dependent on S (the size of the window). Since the SSK function does not compute distance between matched subsequences, it must look for exact subsequence matches and as n approaches S, the probability that two subsequences of size n exist in two separate traces of relatively small size decreases.

We then began to experiment with various values for $\lambda \in (0, 1)$. λ is the decay factor used to weight the contribution of the match based on the number of interior letters. That is, as λ approaches 1, interior letters are increasingly penalized. We were surprised by the drastic increase in the average number of iterations it took for a decision to be reached as λ moved away from 0.5. This can most dramatically be seen for values $\lambda = 0.25$ and $\lambda = 0.75$ in Table 1. We found that these values caused the probability estimates to remain closer to 0.5, this caused the decision function to take longer when the probability estimates where favoring the malicious (i. e., "+1") class, as this threshold is set to 0.75.

Finally, we considered using simple moving averages as opposed to cumulative moving averages to make a classification. We found that using a cumulative moving average performed slightly better than a simple moving average. We reasoned that because the average number of iterations is so low when using the cumulative moving average, the success of two methods would not differ greatly if all other factors remained the same. One would expect to see a greater difference in the performance of the two methods if the average number of iterations is much

	Author	Approach	FP Rate	Recall	Accuracy
1	Pfoh et al.	SVM+SSK (syscalls)	0.0386	0.9928	0.9888
2	Pfoh et al.	SVM+SSK (API)	0.1180	0.9324	0.9290
3	Rieck et al. [3]	SVM+Poly	☐	☐	0.88
4	Rieck et al. [3]	SVM+Poly (extended)	☐	☐	0.76
5	Liao and Vemuri [15]	kNN (total)	0.0	0.917	☐
6	Liao and Vemuri [15]	kNN (novel)	0.0	0.75	☐
7	Xiao and Stibor [4]	STT	0.4286	0.9955	0.9721
8	Xiao and Stibor [4]	STT+SVM	0.3748	0.9997	0.9790

Table 3: A comparison of results from various machine learning approaches to malware detection using system call traces. The ☐ symbol indicates that the information is not available.

higher. This is supported by the API call dataset in which the average number of iterations is higher and the success of the two methods differ more greatly. In both cases, however, the experiments that made use of a cumulative moving average performed better.

After having experimentally optimized the various variables, we see that $n = 3$, $\lambda = 0.5$, $T_{-1} = 0.5$, $T_1 = 0.75$, and using a cumulative moving average produces the best results for the system call datasets. We show that these values contribute to a 99.28% recall, a 99.43% precision, a 98.88% accuracy, and a 99.36% F-measure, with only a 3.86% false positive rate. We performed more thorough testing on the system call data set as it is the data we collected and it is the system call traces that our system focuses on rather than API call traces. We tested our method on the second dataset (i. e., the API call dataset) to strengthen our claim that our approach performs well. For this dataset, we produced the best results with $n = 3$, $\lambda = 0.6$, $T_{-1} = 0.5$, $T_1 = 0.75$. With these inputs, our approach produced a 93.24% recall, a 99.07% precision, a 92.90% accuracy, and a 96.07% F-measure, with a 11.80% false positive rate.

5 Related Work

In this section, we compare the results of our approach with those of other approaches. To our knowledge, there are no other approaches that make use of string kernels with SVMs, however we compare our approach with another SVM-based approach, a k-nearest neighbor approach, and an approach that makes use of probabilistic topic models.

5.1 SVM/Polynomial Kernel Function

The first approach we will compare our results with is the work of Rieck et al. [3]. This approach models the system trace by counting the frequency of each system call. The frequency of a system call becomes the weight of that particular system call and this information is stored in a separate vector for each trace.

These vectors can then be introduced as arguments to a kernel function. In this case, Rieck et al. make use of a polynomial kernel.

For their testing, they made use of a corpus of 10,072 malware samples divided into 14 malware families. The results of this approach can be seen in Table 3, lines 3 and 4. Line 3 represents a round of testing the authors did using normal cross-validation as is the case in our testing, while line 4 represents testing that took place with an extended dataset that included malware that belonged to none of the malware families along with benign processes. We see that, comparatively, our approach is more accurate. This is not surprising as the approach used by Rieck et al. does not consider any sequential information at all.

5.2 *k*-nearest Neighbor Classifier

Liao and Vermuri [15] present an approach that makes use of a k-nearest neighbor (kNN) classifier. A kNN classifier makes use of frequencies by storing the frequency of a single system call on a per-trace basis. That is, to train such a classifier each trace is processed and the frequency with which each system call is used is stored per trace. In order to classify an unknown trace, the classifier computes the k most similar traces from the training set and classifies the unknown trace based on the labels associated with the k most similar traces. In this instance, the authors make use of the cosine similarity.

For their experimentation, the authors made use of 5,285 benign traces and 24 malicious traces. When training, they used 16 of the 24 malicious traces. This leads to a situation in which the results in line 5 of Table 3 include the same 16 of 24 traces when testing as when training. Clearly, the classifier classified these 16 traces 100% correctly. Therefore, the results on line 6 of Table 3 represent results that are a better measure of the approach. Despite this, our approach achieves a higher recall than both approaches and the 0% false positive rate for each test can be attributed to the fact that there are far more benign traces than malicious traces.

5.3 Probabilistic Topic Model

Finally, Xiao and Stibor present an interesting approach that makes use of the supervised topic transition (STT) model. This approach assigns system calls to topics. That is, the algorithm groups the system calls based on co-occurrence. The model is then built by modeling the topic transitions rather than the system call transitions that one might expect.

This approach makes use of an algorithm that iteratively alternates between a Gibbs sampling approach and a gradient descent approach to update the topic assignment and the topic transition model in parallel to train the algorithm. The classification then takes place by generating a topic transition model for the unknown trace and probabilistically predicting a label.

In addition to a pure STT approach, the authors also considered a classifier that makes use of a SVM. In this instance, the same training method is used, however the topic transitions are fed into a SVM. This SVM makes use of a Radial Basis Function (RBF) kernel.

In their experimentation, the authors made use of the same API call dataset we used and tested several methods. The two most successful are described here and the results are depicted in Table 3. Line 7 represents the pure STT approach while line 8 represents the approach in which the authors combined their STT model with a SVM classifier. While this approach performs slightly better than our approach when considering the recall, the fact that they report a 37% and 43% false positive rate favors our approach in this regard.

6 Discussion

In this section we discuss the applicability of our approach to online scenarios and discuss the impact of mimicry attacks on our approach.

6.1 Online Classification

As mentioned in Section 3, our method inherently lends itself to online classification due to the fact that it considers additional system calls as they are produced by the process (i. e., while the process is still running). However, we must also consider the time overhead. The issue with classifying an entire system call trace using the SSK is that a single trace may be hundreds of thousands of system calls long and examining two traces of this length for matching subsequences will clearly lead to a large time overhead. We solve this problem by keeping the lengths (S) of the traces that we input into the SVM relatively small (100 system calls).

By setting S to a relatively small value we make the use of the SSK feasible. However, in order for our classification to be accurate, we need to iterate over some number of windows before a final decision can be made. That is, we must still consider the number of iterations that it takes our method to make a decision. As is shown in Table 1, the average number of iterations for the experiment with the highest accuracy is 13.26, while in Table 2 the average number of iterations for the experiment with the highest accuracy is 37.22. That is, our method of classification can make a decision after only considering a relatively small number of system calls, which significantly reduces the time overhead and allows for online classification.

One may criticize the point that our approach does not consider the entire trace, however all such approaches must address this practical problem somehow. The problem is that one may have to wait an indefinite amount of time for a process to finish. For example, a permanently resident process will only end execution once the system is shut down. That is, practically, one will always have to set a maximum trace length to address this and other approaches do this arbitrarily [16] while our approach makes use of the given thresholds to determine when to stop.

6.2 Mitigating Mimicry Attacks

Mimicry attacks [6, 7] are a class of attacks in which either an adversary drowns the individual steps necessary for delivering the malicious payload in "benign steps" or an adversary "acts benign" for a certain amount of time before delivering a malicious payload.

While this class of attacks is certainly a concern for any system that models program behavior through system call traces, the use of the SSK significantly raises the bar against this type of attack. As mentioned in Section 2.1, the SSK matches on subsequences, where the definition of a subsequence allows for interior system calls. In a simple case, if we consider the system call sequence "12,19,39" to be indicative of malicious behavior, a mimicry attack might try to fool the security mechanism by introducing interior "benign" system calls. For example, the attacker might augment the malicious program such that the system call trace was as follows: "**12**,17,13,**19**,32,**39**". This may be enough to fool signature-based or simple "bag of words" approaches to malware detection, but the beauty of the SSK is that it, by design, will still match on these traces.

On the other hand, if an adversary decides to "act benign" for a time before delivering a payload, our approach may miss the payload if either threshold has been met. The solution for this is simply to raise the threshold. In the most extreme case, one could raise the threshold for the benign class to 1.0. This will result in a system that continuously scans a trace and will only exit if a malicious classification is made. Such an approach would also be applicable in detecting injected code (e. g., shellcode). Due to the fact that any process will be scanned until the threshold for malicious activity is reached, any benign process that is injected with malicious code will also be potentially detected. In order for such an approach to be successful one would most likely have to consider the simple moving average of the probability estimates as described in Section 3. We performed such a test on our system call data and were able to produce 96% accuracy as can be seen in Table 1.

7 Conclusion

This paper proposes a novel method for practical malware detection with system calls using the SSK. We address the large time overhead generally associated with such an approach by considering the moving average of probability estimates over a sliding window. This moving average is then compared to a threshold to predict a class.

Our experimentation shows that this method is both accurate and considerably reduces the time overhead associated with using the SSK for this domain. We test our method on two separate datasets and the fact that our method shows promising results for both datasets makes us confident that this method is universally applicable. Additionally, we compare our approach with other machine learning-based approaches and could show that our approach performs very well in comparison. Finally, we argue that our approach raises the bar against mimicry attacks through the use of the SSK and our threshold mechanism.

References

1. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Technical report, Berlin Institute of Technology (2009)
2. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. Journal of Machine Learning Research **7** (December 2006) 2721–2744
3. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment, Berlin, Heidelberg, Springer (2008) 108–125
4. Xiao, H., Stibor, T.: A supervised topic transition model for detecting malicious system call sequences. In: Proceedings of the Workshop on Knowledge Discovery, Modeling and Simulation, New York, NY, USA, ACM (2011)
5. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Proceedings of the IEEE Symposium on Security and Privacy, Washington DC, USA, IEEE (2001) 38–49
6. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE (2001) 156–168
7. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the ACM Conference on Computer and Communications Security, New York, NY, USA, ACM (2002) 255–264
8. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. Journal of Machine Learning Research **2** (March 2002) 419–444
9. Schölkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge, MA, USA (2001)
10. Platt, J.C.: 5. In: Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. MIT Press, Cambridge, MA, USA (2000) 61–74
11. Pfoh, J., Schneider, C., Eckert, C.: Nitro: Hardware-based system call tracing for virtual machines. In: Advances in Information and Computer Security. Volume 7038 of Lecture Notes in Computer Science. Springer (November 2011) 96–112
12. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology **2** (2011) 27:1–27:27 Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.
13. Herbrich, R.: Learning Kernel Classifiers: Theory and Algorithms. MIT Press, Cambridge, MA, USA (2001)
14. Lin, H.T., Lin, C.J., Weng, R.C.: A note on platt's probabilistic outputs for support vector machines. Machine Learning **68**(3) (October 2007) 267–276
15. Liao, Y., Vemuri, V.R.: Using text categorization techniques for intrusion detection. In: Proceedings of the USENIX Security Symposium, Berkeley, CA, USA, USENIX (2002) 51–59
16. Wang, X., Yu, W., Champion, A., Fu, X., Xuan, D.: Detecting worms via mining dynamic program execution. In: Proceedings of the International Conference on Security and Privacy in Communications Networks. (2007)