Detecting Worm Variants using Machine Learning

Oliver Sharma, Mark Girolami, Joseph Sventek
Department of Computing Science
University of Glasgow, United Kingdom
{oliver, girolami, joe}@dcs.gla.ac.uk

ABSTRACT

Network intrusion detection systems typically detect worms by examining packet or flow logs for known signatures. Not only does this approach mean worms cannot be detected until the signatures are created, but that variants of known worms will remain undetected since they will have different signatures. The intuitive solution is to write more generic signatures. This solution, however, would increase the false alarm rate and is therefore practically not feasible. This paper reports on the feasibility of using a machine learning technique to detect variants of known worms in real-time.

Support vector machines (SVMs) are a machine learning technique known to perform well at various pattern recognition tasks, such as text categorization and handwritten digit recognition. Given the efficacy of SVMs in standard pattern recognition problems this work applies SVMs to the worm detection problem. Specifically, we investigate the optimal configuration of SVMs and associated kernel functions to classify various types of synthetically generated worms. We demonstrate that the optimal configuration for real time detection of variants of known worms is to use a linear kernel, and unnormalized bi-gram frequency counts as input.

1. INTRODUCTION

Worms are malicious programs that spread over the Internet without human intervention. The first Internet worm was unleashed in 1988 and brought down hundreds of machines across America[32]. Several other worms were unleashed in the following decade, many of which wreaked havoc and caused considerable financial damage[11].

None of the damage, however, came close to the $2.6 billion caused by *Code Red*[19] and its variants. Code Red exploited a vulnerability in Microsoft's Internet Information Services (IIS) web server through which it infected the host. Once infected, Code Red generated a set of random IP addresses to which it then tried to spread. Fortuitously, however, there was a fatal flaw in this worm: it used a static seed to generate the IP addresses, which meant that all infected hosts generated the same set of IP addresses. This flaw prevented the worm from spreading far.

Several days after Code Red's arrival, a change in its behavior was observed: it began to probe new hosts. The change in behavior was due to an updated version of Code Red, identical in all aspects except for the random number generator – which now used a dynamic random seed. This mutation enabled it to infect $359,000$ hosts in less than 14 hours.

A worm that spread even faster was the Slammer[18] worm, which infected most of its $75,000$ victims within 10 minutes. This worm was the first Warhol[34] worm observed in the wild. These worms get their name from Andy Warhol's remark that "in the future, everybody will have 15 minutes of fame", since they are capable of spreading to most vulnerable machines within 15 minutes.

Two major observations may be made about worms:

- they tend to be released in variants of each other, as demonstrated by Code Red, and
- they spread much faster than humans can respond.

These two observations suggest that it is imperative to find a way of detecting and stopping worms automatically. In particular, we focus on the problem of being able to detect variants of known worms in real time. Generally, there are two categories of network intrusion detection systems[20]: anomaly detection and misuse detection.

In *anomaly detection*, systems are equipped with a model of *normal* traffic. The idea is to detect intrusions by comparing traffic to this model, looking for deviations. Due to the diversity of network traffic, however, it is difficult to model normal traffic. Email relaying and peer-to-peer queries, for example, show worm-like traffic characteristics. Moreover, abnormal traffic does

*CoNEXT'*07, December 10-13, 2007, New York, NY, U.S.A.

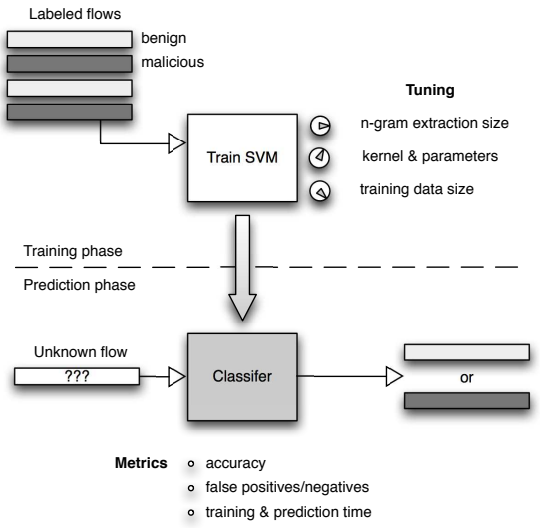


Figure 1: Proposed solution: use machine learning based pattern recognition, such as Support Vector Machines (SVMs), to classify flows as malicious or benign.

not necessarily constitute an intrusion. Consequently, anomaly detection has a high false alarm rate and is hardly used in practice.

In *misuse detection*, on the other hand, systems are equipped with models of known intrusions. These models, known as signatures, are used to identify intrusions by looking for matches in the network traffic. A problem with misuse detection is that only intrusions whose signatures are known can be detected.

A signature is a type of fingerprint that can be used to identify intrusions. In its simplest form, it consists of a string of characters (or bytes). Nowadays many intrusion detection systems[23, 28] also support regular expressions and even behavioral fingerprints.

Although there is a lot of ongoing research into anomaly systems, misuse detection systems have become the *de facto* standard for intrusion detection, since they are simple and scale well. When it comes to rapidly spreading intrusions such as worms, however, misuse detection systems have a serious bottleneck: generating the signature itself.

Signatures are typically created by security experts who analyze network and host logs after intrusions have occurred. As you can imagine, sifting through thousands of lines of log files, looking for characteristics that uniquely identify an intrusion is a vast and error prone undertaking.

We propose to leverage the flexibility of machine learning based pattern recognition techniques to overcome this shortcoming. The general idea is to train a classifier to distinguish between malicious and benign flows, and hence implicitly generate signatures to use for classifying unknown flows, as depicted in Figure 1. In particular, we investigate the feasibility of using Support Vector Machines[6] (SVMs) to detect variants of known worms in real time.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 discusses how we generated synthetic worm variants to test our system. Section 4 describes SVMs and how we preprocessed data for them. Section 6 presents our results. Section 7 discusses parallels to existing work. And finally, Section 8 presents conclusions and explores future work.

2. RELATED WORK

2.1 Signature generation

There are a number of research systems that focus on automatically generating signatures for known intrusions. Three prominent examples are Earlybird[31], Autograph[12], and Polygraph[22]. Earlybird taps into the network watching passing traffic and builds a histogram of all observed strings from which it constructs signatures for the most frequent ones. Its basic assumption is that since worms spread to many hosts in a short period of time, strings that occur both frequently and are widely dispersed, must belong to a worm.

Autograph improves over Earlybird, by (i) using a heuristic filter rather than looking at all traffic, and (ii) observing entire flows rather than just single packets. Using a heuristic filter greatly reduces the amount of work that needs to be done by the signature generator. The filter does not need to be very accurate, since it must simply flag potentially malicious flows for later examination. Autograph derives signatures from flagged flows by using greedy induction algorithms such as longest common substring.

Polygraph is based on Autograph's framework, but focuses on fingerprinting polymorphic worms, which are worms that mutate from hop to hop when traversing a network. It generates signatures using sophisticated string matching techniques, such as, finding the longest common subsequence.

Finding polymorphic worms is a similar problem to our problem – finding worm variants. Since a polymorphic worm mutates from hop to hop, each instance is a variant of the previous instance. It should, therefore, be fairly straightforward to adapt polymorphic worm detection techniques to find worm variants and vice versa.

Although all three systems demonstrate great accuracy in generating signatures, there are some shortcomings. Assuming that frequently occurring strings are malicious is inaccurate. `HTTP GET` requests, for instance, are extremely common and do not constitute a worm. Blocking all packets with an `HTTP GET` request

is unacceptable. The proposed solution is to use whitelists to keep track of frequently occurring, yet benign, strings. This solution, however, only inverts the problem of finding malicious signatures.

Two systems that approach this problem from slightly different angles are Honeycomb[14] and Vigilante[7]. Honeycomb is a system that is deployed on an idle host containing untreated vulnerabilities (known as honeypots[33]). It works with the fundamental assumption that any communication with the host is inherently malicious, thereby building a database of signatures. A drawback with this approach is that only intrusions that infect the honeypot can be detected.

Vigilante, on the other hand, is a host-based intrusion detection systems that monitors executing code for malicious behavior, such as illegal memory accesses. Any code that attempts such malicious behavior is immediately traced back to the flow that caused it, thereby creating a pool of suspicious traffic from which signatures can be generated.

2.2 Machine learning for intrusion detection

We intend to tackle the problem of detecting variants of known worms by leveraging machine learning techniques. There have been a number of investigations into the feasibility of machine learning for intrusion detection [8, 30, 24], most of which have inherently been some form of anomaly detection.

Three approaches that are of particular relevance to this work are fileprints[16], classifying malicious executables[13], and detecting attacks using language models[26]. All three of which investigate the feasibility of being able to accurately identify the true type of an arbitrary file [16, 13], or flow [26] using statistical analysis of their binary contents without parsing. In particular they apply n-gram analysis, upon which we will elaborate in Section 4.2.

Our approach essentially fuses machine learning techniques, n-gram analysis of malicious executables, and automatic signature generation, with particular emphasis on the detection of variants of known worms. Note, however, that we intend to detect worms by classifying network traffic contents directly, rather than generating signatures explicitly.

3. SYNTHETIC WORMS

Machine learning algorithms train classifiers by looking for common patterns in a set of sample data. The more ground this set covers, the better the quality of the resulting classifiers. Due to the difficulty in obtaining large numbers of real-life worm variants, we created synthetic variants for this study.

The data set we created consists of malicious and benign flows[1] as depicted in Figure 2. Malicious flows are made up of random data, a signature, and more random data, whereas benign flows are made up of only random data. A fundamental assumption we are making here is that all variants of the same worm have at least a very small amount of data in common. This common data is represented by the signature. Each malicious flow is different from the next, the only thing they all have in common is a small string of text – the signature.

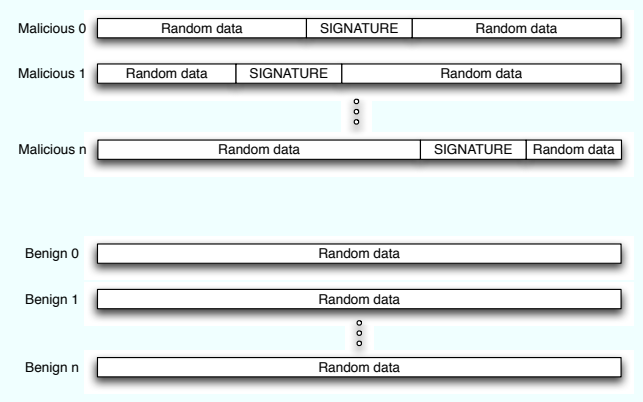


Figure 2: Synthetic flows. Malicious flows consist of random data but contain the worms signature at an arbitrary position, whereas benign flows are just random data.

A shortcoming of using random padding is the problem of choosing a random distribution that realistically resembles a flow's real application data (padding). To overcome this problem, we pad our synthetic flows with randomly selected chunks of benign flow traces. We compare the classifiers performance using padding generated with a uniform random number generator and padding generated from benign flow traces.

Note that although we know the signature, since we created the synthetic flows, we do not make this information available to the SVM. All we are telling the SVM is whether a flow is malicious or benign. We intend for the SVM to deduce the signature implicitly from only the information provided, thereby alleviating the burden of having to produce signatures manually.

3.1 Synthetic Slapper

In an effort to make flows used more realistic we created malicious flows that are loosely based on the Slapper[1] worm. The Slapper worm was first discovered in 2002 and exploits an OpenSSL vulnerability in Apache, the Internet's most popular web server. It allows the attacker to run arbitrary and potentially malicious code, and builds a botnet of infected machines. For the point

[1]Flows in communication networks are typically defined as data exchanges between a single source (IP, port) and destination (IP, port). For the purpose of this work it suffices to think of a flow as a stream of binary data that was exchanged between two hosts.

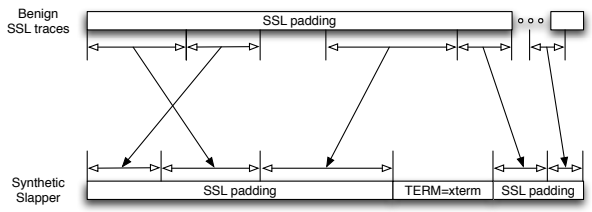


Figure 3: Synthetic worms loosely based on the Slapper worm consist of random-sized chunks of SSL traces used as padding.

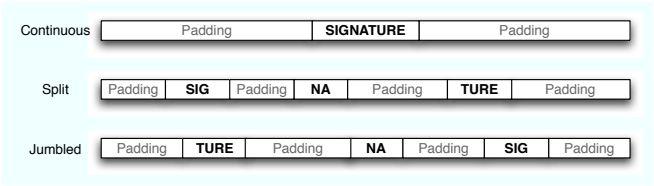


Figure 4: Synthetic worm models. Continuous signatures, split signatures, and jumbled signatures.

of this work it suffices to know that the Slapper worm exploits a vulnerability in OpenSSL and has a signature that contains the string `TERM=xterm`[2]. We created these by first capturing a benign SSL trace and then using randomly selected chunks of this trace as the padding for malicious and benign flows. Also, we used the string `TERM=xterm` as the signature, as shown in Figure 3.

3.2 Synthetic signature models

Thus far we have assumed that a worm's signature is a continuous sequence of characters, as is the case with the synthetic Slapper worm. Often, however, it is not this simple. Many worms are described by signatures containing wild-cards and regular-expressions. Consider, as an example, the Lion worm[10], which exploits a vulnerability in the bind DNS server. This worm's payload essentially consists of a regular DNS request that contains an invariant value used to overwrite a return address. Since this invariant value alone does not uniquely identify the attack, a regular expression that contains further disambiguations is used.

We consider three different types of signatures: continuous signatures, split signatures, and jumbled signatures as shown in Figure 4. Continuous signatures are signatures that consist of a continuous sequence of characters, as we have described thus far. Split signatures are signatures that are split at arbitrary locations and spread apart with padding. And jumbled signatures are signatures that are split and then jumbled into arbitrary order. Note that these signature classes correspond to Polygraph's[22] conjunction, ordered token, and unordered token signatures.

[2]Note that this is an oversimplified version of the signature that is used by Snort and Bro. Both of which additionally consider some of Slapper's other actions, such as probing servers and exhausting their connection pools.

3.3 Corrupted signatures

Thus far we have assumed that all variants of the same worm share at least a very small amount of code. We emulated this sharing by creating variants that all contained the same signature but did not inform the SVM of this signature in the training data. All we told the SVM was whether a flow was considered malicious or benign. Based on this training data, the SVM generated a model to distinguish between malicious or benign flows. Essentially this means that the SVM implicitly found the signature that we had embedded into malicious flows.

To further test the flexibility of support vector machines for the detection of variants of known worms, we created a test set containing corrupted signatures. That is, we trained the classifier on synthetically created worms that all contained the same signature, as discussed previously. However, instead of testing these classifiers on worms with the same signature, we tested them on corrupted (or dirty) versions of the signatures. To create this test set we randomly (to various degrees) corrupted parts of the signatures in the test set.

4. SUPPORT VECTOR MACHINES

Support Vector Machines (SVMs)[6] were first introduced in the mid 1990s, and have since been established as one of the standard tools for machine learning and data mining. SVMs are based on recent advances in statistical learning theory and have been successfully applied in real-world problems such as text categorization[9], image classification[5], and hand-written character recognition [6].

Support Vector Machines are typically used for two types of problems: regression and classification. Regression finds curves of *best fit* for data, while classification categorizes data into two or more classes. In this work, we are interested in classification, specifically classifying flows as malicious (worm) or benign (non-worm).

The simplest case of classification is binary classification. Consider, as an example, determining a person's gender given only their height and weight. Using a set of example heights and weights, we can draw up a hypothesis that allows us to establish a person's gender based solely on their height and weight. We are not guaranteed that this hypothesis is correct, but if the set of examples used is large enough (and indeed there exists a correlation between genders, heights, and weights) we can obtain a good estimate.

Visually, finding the hypothesis can be described as plotting the given heights and weights in a two dimen-

sional coordinate system and drawing a line (separating hyperplane) that divides the points into regions *male* and *female*. Clearly, there are many possible separating lines – the best would be the one which is the furthest distance from any training point. For a more formal discussion on SVMs see [3].

Given a set of points and their classification, SVMs determine the maximum margin separating hyperplane. Unfortunately, it is not always that straightforward. Sometimes data classes are not linearly separable. For simple cases it is sufficient to say that data points must lie within a certain distance of the margin to be classified as a certain type. It is often the case, however, that data is simply not linearly separable. This is where the so-called *kernel trick* comes to the rescue.

4.1 Kernels

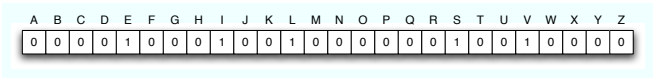
Kernels provide support vector machines with the capability of implicitly mapping non-linearly separable data points into a different dimension, where they are linearly separable. Consider, as an example, a cluster of data points forming an elliptical structure. If surrounded by data points of a different class, they would clearly not be linearly separable. Yet looking at the data, it is easy to imagine an elliptical boundary that clearly distinguishes between the two types. The trick is to map the data points into a dimension where they are linearly separable.

Mapping data points to a higher dimension, comes at a cost. More dimensions, mean larger vectors which mean larger memory requirements and longer calculation times. Fortunately, however, SVMs do not need to store these high dimensional vectors explicitly. They map the input data into the higher dimension and then are only required to store inner products[3].

Different kernel functions provide different mappings. Unfortunately, there is no silver bullet choice of kernel. Each kernel has its advantages and disadvantages for the data in question. The choice of kernel greatly affects the SVM's ability to classify data points accurately. It is also arguably the most important parameter to be tuned in the support vector machine.

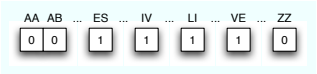
In this work we investigate the optimal configuration of support vector machines for the classification of worms. We have selected three candidate kernels: (i) a linear kernel, (ii) a radial basis function (RBF) kernel, and (iii) a string kernel[17].

We chose the linear kernel as a candidate kernel for our investigation since it is the simplest kernel, which, as the name suggest simply finds a linearly separating hyperplane. We chose the RBF kernel as another candidate kernel since it has good generic performance and is therefore often recommended as a starting point. We chose the string kernel as a third candidate kernel, since it is a kernel that was specifically designed to work for string matching problems, as in our case.



(a) Uni-gram vector for *ELVIS* (or *LIVES*)

(b) Bi-gram vector for *ELVIS*



(c) Bi-gram vector for *LIVES*

Figure 5: Feature vectors representing the words *ELVIS* and *LIVES*.

4.2 Feature extraction

Support vector machines and indeed many other machine learning techniques perform statistical analysis on input data that can be readily described by feature vectors. A feature vector is a row of data where each column represents a specific character in a fixed alphabet.

Often, however, data, such as images and text, cannot readily be described as feature vectors, and features must be extracted explicitly. A case-insensitive feature vector that represents words, for example, could consist of 26 columns, where columns 1 to 26 represent the characters A to Z respectively. The value held in each column denotes the presence (1) or absence (0) of it's respective character in the given word.

Consider, as an example, the word *ELVIS* respresented by the feature vector in Figure 5(a). The columns representing characters `E`, `L`, `V`, `I`, and `S` contain a 1 and all others contain 0s.

A shortcoming of this simple representation is that any combination of the letters in the word *ELVIS*, would produce the same feature vector. The word *LIVES*, for example, is represented by an identical feature vector. To overcome this problem, we can use a feature vector that represents all possible combinations of two characters, starting with AA all the way up to ZZ. Figures 5(b) and 5(c) show the feature vectors for *ELVIS* and *LIVES* using bi-grams. As you can see, they are now represented by different feature vectors.

The more characters we combine per entry in the feature vector, the more precisely the word is represented, but the greater the number of elements. In the case of tri-grams, for example, *ELVIS* would be represented by entries at `ELV`, `LVI`, `VIS`. Using 5-grams, we could represent the entire word in one column of the feature vector. In addition to marking the presence or absence of character combinations, one can keep frequency counts of

these combinations. This technique is known as *n-gram extraction*.

In n-gram extraction the number of columns in the feature vector increase exponentially with the size of n. An array of integers for 4-grams, for example, would consume 16 gigabytes of memory. One might think that, given infinite memory and processing time, the higher the value of n, the more precise the information we obtain. This, however, is not the case. Essentially, we are estimating the probability at which individual n-grams occur, by measuring their relative frequencies. Nevertheless, there will almost always be situations where we encounter an n-gram that has not been seen before. Using frequency counts, this n-gram would be estimated at a nonzero probability, even though its relative probability is zero. This is known as the *zero frequency problem*[27]. We investigate the tradeoffs involved in selecting the value of n and discuss its implications for the real-time detection of worm mutations.

Since worms are often transmitted in the form of binary executables, we are using the full byte range as our alphabet. That is, rather than using just letters `A` to `Z` or even just displayable characters, such as ASCII characters, we are using byte values 0 to 255, and hence do not distinguish between displayable and non-displayable characters.

Since we are interested in detecting worms in real-time, we limit ourselves to 1-grams, 2-grams and 3-grams for this work. These sizes are easily implemented as integer arrays in memory, and require 1 KB, 256 KB, 64 MB respectively[3]. Larger values of `n` would have to be represented using hashtables or even bloom filters. Alternatively, features could be extracted into rabin fingerprints[25]. We leave higher values of `n` and alternative representations for future work.

Note that string kernels deal with feature extraction implicitly for text-like data, as in our case, and hence do not require preprocessing with n-gram extraction. The basic idea behind string kernels is to compare documents by all subsequences of a given length. These subsequences are ordered, but not necessarily continuous, strings that are weighted by their continuity. See [17] for a detailed discussion.

5. EXPERIMENTAL APPROACH

All experiments were performed on an Intel Xeon with a 2.40GHz CPU and 1GB of RAM. We used libsvm[4] version 1.8 as our support vector machine implementation for linear and RBF kernels, and a libs[15] version 1.3, a libsvm modification, for string kernels.

As a baseline for each experiment, we used a training set of 100 flows, half of which were malicious and half of which were benign. We investigate the implications of this training data size in Section 6.2.3 below. We used a test set of size 1000, again half of which were malicious and half benign flows. And we repeated each experiment, with different sets of data, 50 times to obtain averages and standard deviations.

[3]These size values are calculated assuming 4-byte integers

Worm	Payload (bytes)	Signature (bytes)	Ratio
Code Red	4039	396	9:1
Slammer	404	16	24:1
Witty	1184	64	18:1

Table 1: Approximate data-to-signature ratios for some exemplar worms.

Note that we mention data-to-signature ratios in most of the results. As the name implies, the data-to-signature ratio shows the amount of data (padding) relative to the amount of signature. Table 1 shows exemplar data-to-signature ratios for the Code Red[19], Slammer[18], and Witty worm[29].

6. RESULTS

The discussion is organized as follows. First, we investigate the choice of kernel and associated parameters. Second, we explore other factors that potentially influence the classifier's performance, such as the optimal n-gram size. And third, we investigate the classifiers resilience to change in signature model and corruption.

6.1 Choosing the kernel

As discussed above, the choice of kernel is arguably the most important SVM parameter to be tuned. This section compares the performance of the linear, RBF, and string kernels, and investigates their optimal parameter values.

6.1.1 Configuring the linear kernel

The linear kernel has only a single parameter that can be tuned – the C value. This parameter determines the softness of the margin used to distinguish between classes. The softer the margin, the more erroneously placed data points it tolerates. Essentially it is the trade-off between fitting the training data and maximizing the margin.

We investigated the affect of altering the C parameter by performing a 10-fold cross validation. That is, we took a dataset of 1000 entries and split it into 10 equal parts. We then used 1 out of these 10 parts to train the classifier and the other 9 to test it. We repeated this with a range of C values.

Altering the C value did not affect the classifiers performance. This is true for both scaled and unscaled data.

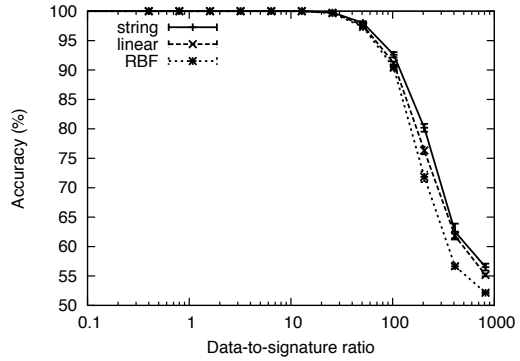


Figure 6: Prediction accuracy at various data-to-signature ratios for the linear, RBF, and string kernel.

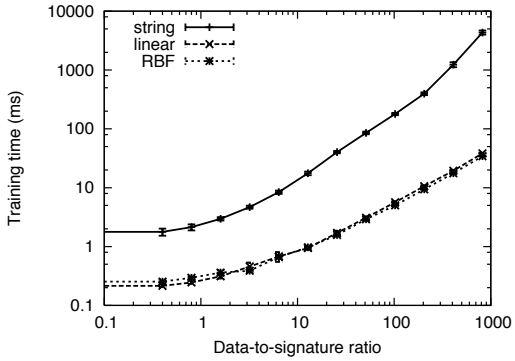


Figure 7: Training time for a single flow using the linear, RBF, and string kernel.

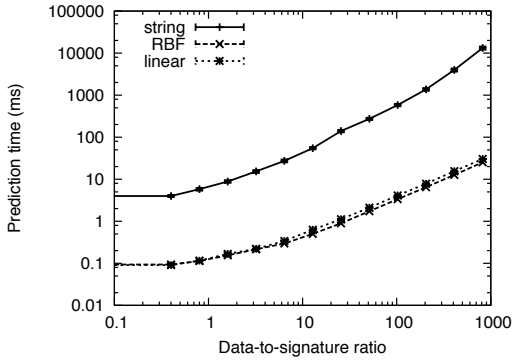


Figure 8: Time taken to predict a flows class using the linear, RBF, and string kernel.

6.1.2 *Configuring the RBF kernel*

There are two parameters that can be tuned in RBF kernels: C and *gamma*. The RBF's C parameter serves the same purpose as the linear kernel's. The *gamma* value is used to control the width of the RBF kernel.

We performed a grid-based cross validation to determine the optimal parameter values for our training data. This is a similar cross validation as used for determining the linear kernel's C value, except that we investigated combinations of the two parameters. We used libsvm's[4] *grid.py* script to perform this cross validation and found that the RBF performs well at C value of $lg(4)$ and a *gamma* value of $lg(-15)$.

6.1.3 *Configuring the string kernel*

There are two important parameters that can be tuned in the string kernel: the substring length, and whether to consider all lengths up to this or just the given length itself. We investigated the affect of varying the substring length relative to the signature length for both fixed and variable-length strings, and found that the performance peaks at a string-length to signature-length value of 0.5 using fixed-length strings. Using Slapper's 10 character signature, this corresponds to a substring length of 5.

It should also be noted that the processing time using variable-length strings is orders of magnitude greater than using fixed length strings. This is because the number of substrings to consider increases exponentially when considering all possible substrings up to a certain length rather than just a single fixed length.

6.1.4 *Prediction accuracy*

Now that we have investigated the optimal parameter settings for our kernels, we can compare their classification accuracy. The graph in Figure 6 shows the accuracy at various data-to-signature ratios for the linear, RBF, and string kernel. As can be seen in the graph, they all show similar accuracies. The string kernel shows the best accuracy, then the linear, and finally the RBF. Given that all three kernels show similar accuracies, we next investigate their performance in terms of training and prediction times.

Note that all accuracies trail off to 50%. This simply indicates that the classifier is guessing. It is as accurate as flipping a coin and cannot really get worse than this.

6.1.5 *Training time*

The graph in Figure 7 shows the time it takes (in milliseconds) to train the classifier's portrayed in the previous graph. As expected, the time taken increases with data-to-signature ratio due to the increase in total data. Note that it shows the time to train a training set containing a single flow. To obtain an estimate of how long it will take to do larger training sets, multiply this number by the number of entries in your training data.

As you can see the time taken to train an SVM using string kernels is far greater than the time taken for linear and RBF kernels.

6.1.6 *Prediction time*

Arguably, the training time is not too important,

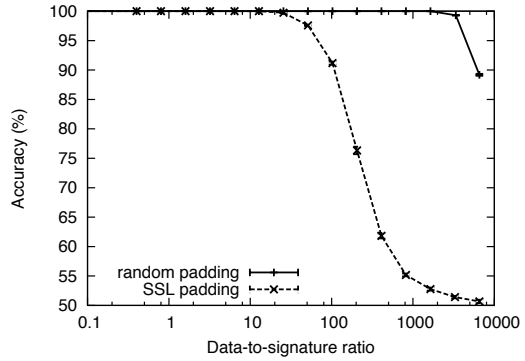


Figure 9: Prediction accuracy at various data-to-signature ratios for flows padded with random and SSL data.

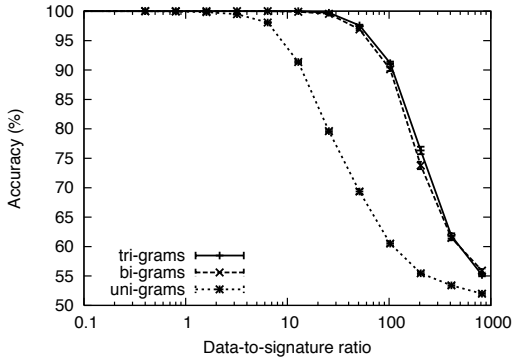


Figure 10: Prediction accuracy at various data-to-signature ratios for uni-grams, bi-grams and tri-grams.

since we can train our classifiers offline, as long as they are able to perform well at classification time. The graph in Figure 8 shows the time (in milliseconds) taken to predict a single flow's class. As was the case with training time, the string kernel's prediction time is between one and two orders of magnitude higher than for the linear and RBF kernels.

6.1.7 Summary

So what is the best choice? The classifier with the best accuracy used a string kernel. However, since this accuracy was not significantly higher than with the other two kernels, we are basing our choice on processing time. This takes the string kernel out of the race since it is much slower in both training and classifying. Leaving us with the choice between the linear and RBF kernel. Since the RBF kernel is sensitive to it's parameter values, we opted for the linear kernel as the best choice for our scenario.

6.2 Preparing the data

Having established that the linear kernel gives encouraging results, the natural next step is to consider tuning the other factors that affect a classifier's performance. In this section we present results comparing various configurations that were considered. In particular we explored:

- random versus SSL padding,
- the optimal n-gram extraction size,
- the affect of varying the training data size, and
- whether normalizing the data improves performance.

6.2.1 Random vs SSL padding

Figure 9 shows the prediction accuracy at various data-to-signature ratios for flows padded with random and SSL data. As expected, it is much harder to distinguish between malicious and benign worms using SSL padded data. This is most probably due to the fact that the SSL padding is likely to have large chunks repeated among all flows, thereby diluting the signature. Using uniform random data as padding, on the other hand, highlights the signature's information content.

Since SSL padded flows are both more realistic and constitute a harder problem, we will hence forth only show results for SSL padded data.

6.2.2 Optimal n-gram size

The graph in Figure 10 shows the prediction accuracy at various data-to-signature ratios for uni-grams, bi-grams and tri-grams. The results shown were obtained using a linear kernel. They illustrate that the prediction accuracy increases with higher values of n. Using tri-grams, we are able to correctly classify all flows at a data-to-signature ratio of approximately $12:1$ and over 90% at a ratio of $100:1$.

As you can see from the results, uni-grams perform considerably worse than bi-grams and tri-grams. The performance improvement of tri-grams over bi-grams is negligible, which means the best choice for our scenario is to use the smaller of the two. As discussed above, a bi-gram feature vector consists of 256^2 features, whereas a tri-gram feature vector consists of 256^3. In terms of memory, a feature vector made up of an array of 4-byte integers for bi-grams is allocated 256 kilobytes, whereas an array for tri-grams is allocated 64 megabytes.

6.2.3 Training data size

As mentioned above, we used a training data size of 100 as a baseline for our experiments. In this section we investigate the training data size's affect on the classifiers performance. The graph in Figure 11 shows how the accuracy varies with training data size for data-to-signature ratios of $25:1$, $100:1$ and $200:1$. As you can see the accuracy increases with training data size. The larger the training data size, the better the accuracy. This is probably because the larger the training

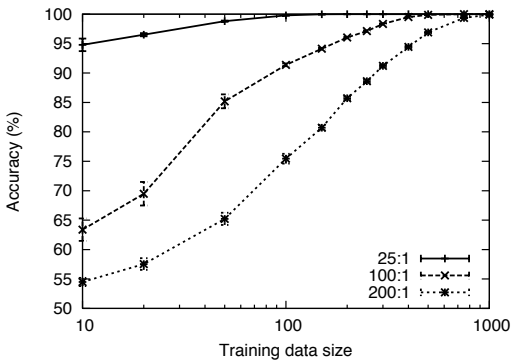


Figure 11: The effect of training data size on prediction accuracy for various data-to-signature ratios.

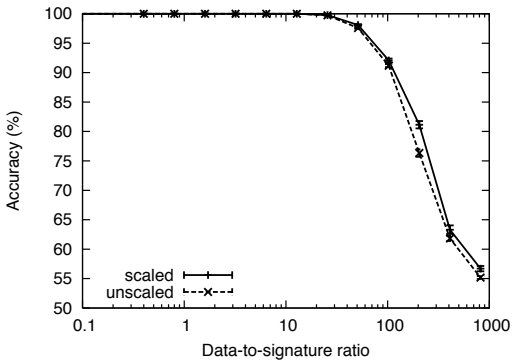


Figure 12: Prediction accuracy at various data-to-signature ratios for scaled and unscaled data.

data size the larger the chance of the signature being the only repeated string amongst all malicious flows.

Unfortunately we cannot simply pick an optimal training data size from these results. In a real situation we would be constrained by the number of available traces. We can, however, use this information to guarantee certain levels of accuracy. Consider as an example, that we need to achieve an accuracy of at least 90% and we are dealing with flows with a data-to-signature ratio of 100. We can then use this graph to determine that we require a training data size of 100.

6.2.4 Scaling results

As discussed, we are using n-gram frequency counts as feature vectors. These counts range from 0 to the maximum value for 32-bit integers in our implementation. As you can imagine, the actual value varies greatly depending on the flow size and hence data-to-signature ratio. We compared these raw feature vectors to normalized (or scaled) versions bound between 0 and 1. The graph in Figure 12 shows the affect of scaling the data before passing it to the SVM.

Scaling only improves the accuracy by a small amount.

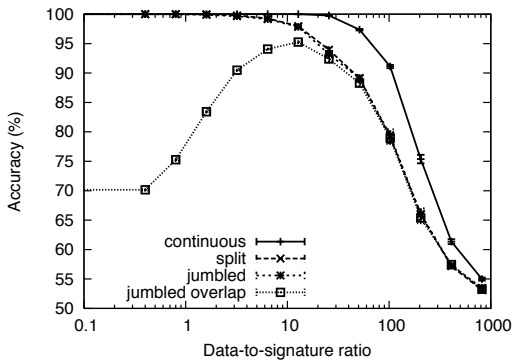


Figure 13: Prediction accuracy at various data-to-signature ratios for different worm models.

This may be due to the fact that we have not experimented using training and test sets of mixed data-to-signature ratios.

Since scaling the data comes at great computational cost, and apparently does not improve our performance by large amounts, we have decided to use unscaled data hence forth. Determining whether scaling boosts performance when mixing flow sizes, remains open for future investigation.

6.3 Resilience to worm variations

Resilience to variations of the original worm is a key advantage when using machine learning based classifiers. Thus far we have discussed variations in the application specific data part of flows only – leaving the signature unchanged. In this section we explore the effect of altering the signature itself in addition to altering the data.

6.3.1 Synthetic signature models

As discussed previously, we are concerned with three different synthetic signature models. Models that contain continuous signatures, split signatures, and jumbled signatures. In particular we are interested in observing the capability of our classifier to train using one model only and still be able to correctly classify worms with a different signature model. Note that we are not changing the characters in the signature, merely their location in the flow.

The graph in Figure 13 shows the prediction accuracy at various data-to-signature ratios for the different signature classes. We trained this classifier using continuous signatures. As you can see it is still able to detect worm variants where the signature has mutated into split or jumbled models, although at a slight cost of accuracy.

There is no significant difference in detecting split and jumbled signatures. We did, however, introduce a fourth model, jumbled with overlapping, which showed

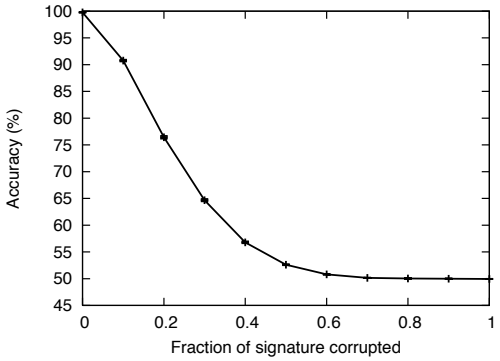


Figure 14: Prediction accuracy at various levels of signature corruption.

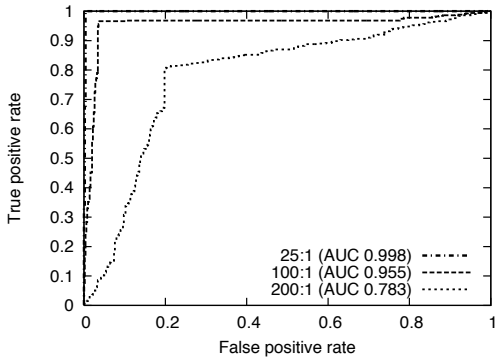


Figure 15: ROC curve showing the trade-off between true and false positive rates for various data-to-signature ratios.

a different behavior. Unlike jumbled signatures, which consist of split signature fragments that have been reordered, jumbled with overlapping signatures do not take other signature fragments' locations into consideration, and hence overlapping may occur. This explains the bell shape of its graph. For small data-to-signature ratios, the chances of overlapping fragments is significantly higher, hence corrupting the signature's characters. This is shown by the relatively poor performance for data-to-signature ratios of less than 10. After that size, the accuracy curve eventually matches those of split and jumbled signatures.

6.3.2 *Corrupted signatures*

In this section we present results demonstrating the classifiers capability of coping with corrupted signatures. Given, continuous signatures and using a uniform random number generator, we randomly corrupted a certain number of characters. The graph in Figure 14 shows the classifiers accuracy at various levels of corruption. As you can see the accuracy decreases gracefully with increasing signature corruptness.

6.3.3 *False alarms*

Thus far we have used overall accuracy as a metric for determining how effectively we can classify flows. Accuracy is an intuitive measure, but unfortunately does not reveal all problems. In intrusion detection, the false alarm rate is of paramount importance. Falsely classifying flows as malicious (false positive) means that we may block legitimate traffic. Conversely, falsely classifying flows as benign (false negative) means that worms get by undetected.

Receiver operator characteristics (ROC) [36] curve analysis presents a way of quantifying the trade-off between the detection rate (true positives) and the false alarm rate (false positive). ROC curves have their roots in signal detection and medical decision making, and have recently become a popular way of analyzing machine learning classifiers.

The ROC curve for a classifier is generated by plotting the true positive rate versus the false positive rate at various confidence intervals. The true positive rate can be determined by dividing the number of true positives by the total number of positives. And the false positive rate can be determined by dividing the number of false positives by the total number of negatives.

There are certain regions of interest in an ROC graph. The diagonal line joining the bottom left and top right corner indicates random guessing by the classifier. The region above this line is further divided with a perpendicular line that connects to the top left corner. Classifiers in the lower region produce fewer false positives than true positives, whereas classifiers in the upper region produce fewer true positives than false positives. As a general rule, classifiers that are closer to the top left corner perform better than ones that are further.

Note that a single classification will produce only a single point in the ROC space. We obtain curves by first sorting the test sets individual classifications by their confidence values[4], and then iterating over these values, computing true and false positive rates for all classifications up to and including the current value.

A single metric that can be obtained from these curves is the total area under the curve (AUC) [2]. The larger this number, the better the classifier. The graph in Figure 15 shows how the AUC decreases for increasing data-to-signature ratios.

7. DISCUSSION

In this section we explicitly compare our work with two studies [22, 24] that have taken fundamentally different approaches to the worm detection problem, yet are both related to our work.

[4]Prediction confidence in SVMs is determined by calculating the distance to the separating hyperplane.

Polygraph [22], a misuse detection system, generates signatures for polymorphic worms by employing various string algorithms, such as longest common subsequence, on pools of suspicious and innocuous flows. This is comparable to our approach, where we train classifiers using labeled training data sets consisting of malicious and benign flows.

Polygraph suggests three different algorithms, each optimized for a different worm model (or signature class). If the worm model is unknown, they suggest experimenting with each algorithm and selecting the one that yields the least false alarms. In contrast to this, we suggest the use of a single algorithm, based on machine learning techniques, that we have shown to be flexible enough to deal with different types of worm models.

The cost of this added flexibility lies in run-time performance. Since Polygraph explicitly generates signatures that can be used by fast string matching algorithms, such as employed by Snort and Bro, it will almost certainly yield a higher throughput rate. We leave a thorough comparison of run-time performance for future work.

As is the case on our work, Perdisci et al [24] investigate the use of support vector machines in intrusion detection. The fundamental difference is that [24] investigates anomaly detection, whereas we investigate misuse detection. [24] proposes the use of an ensemble of one-class SVMs using gaussian kernels and determines associated parameters using several trial runs. We, on the other hand, investigated the use of binary-class SVMs using various kernels and determined associated parameters using cross-validation.

Perdisci et al [24] investigate a wide range of n-gram values by approximating n-gram values higher than 2 using *2v-grams*[5]. They show that 2-grams yield good results, however propose the use of multiple classifiers, each operating in a different feature space, to give optimal results. In contrast to this, we investigated only n-gram values between 1 and 3, and have shown that 2-grams yield similar results to 3-grams.

8. CONCLUSION AND FUTURE WORK

Machine learning based pattern recognition techniques show encouraging results in the worm detection problem. We have investigated the optimal configuration for support vector machines and associated kernel functions. We have demonstrated that tuning these parameters greatly affects the classifiers performance. For this particular application the optimal configuration is to use a linear kernel, bi-grams and unnormalized data. We have also demonstrated that SVMs are robust to mutations in signature model and, to a certain degree, signature corruption.

It is important to mention how the use of approximate string matching [21] for detecting signature corruption relates to our approach. Armed with a worm's signature, it is evident that approximate string matching algorithms would outperform SVMs at detecting variants. Note, however, that the approximate string matching algorithm must be informed of the worm's signature, whereas SVMs are able to deduce it on their own. Since the two approaches are not supplied with the same amount of information, a direct comparison cannot be made.

Confidence values associated with an SVM must be obtained in an ad-hoc manner by, for example, calculating the distance to the separating hyperplane. Gaussian processes[35], on the other hand, return a confidence value as part of their result automatically. Given that gaussian processes have been shown to produce similar accuracies to SVMs, and confidence values could be used to reduce the number of false positives, it is a natural next step to investigate their use in our scenario.

We have shown that our classifier's accuracy is greatly affected by the number of entries in the training data. Due to the difficulty in obtaining large numbers of real worm variants, we are investigating a way of automatically generating variants from a single worm. This variant generator consists of three machines connected in series: an attacker, a forwarding proxy, and a target. The basic idea is to launch a worm from the first machine, reassemble its flow and randomly mutate it in the second machine, and then forward it to the third. As a final step we verify whether the target has been infected and label the generated mutation as malicious or benign accordingly.

[5]This technique uses a sliding window size of 2 and approximates greater values of n by incrementing the window by v steps, when traversing the data.

9. REFERENCES

[1] I. Arce and E. Levy. An analysis of the Slapper worm. *Security & Privacy Magazine, IEEE*, 1(1):82–87, 2003.

[2] A. Bradley. Use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

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

[4] C. Chang and C. Lin. LIBSVM: a library for support vector machines. *Software available at http://www. csie. ntu. edu. tw/cjlin/libsvm*, 2001.

[5] O. Chapelle, P. Haffner, and V. Vapnik. Support vector machines for histogram-based image classification. *Neural Networks, IEEE Transactions on*, 10(5):1055–1064, 1999.

[6] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron,

L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. *Proc. ACM symposium on Operating systems principles*, pages 133–147, 2005.

[8] W. Hu, Y. Liao, and V. Vemuri. Robust Support Vector Machines for Anomaly Detection in Computer Security. *Proc. International Conference on Machine Learning and Applications*, pages 23–24, 2003.

[9] T. Joachims. Text categorization with support vector machines: learning with many relevant features. *Proc. European Conference on Machine Learning*, (1398):137–142, 1998.

[10] A. Kasarda. The Lion Worm: King of the Jungle. *SANS reading room, http://www. sans. org/rr*.

[11] D. Kienzle and M. Elder. Recent worms: a survey and trends. *Proc. ACM Workshop on Rapid Malcode*, pages 1–10, 2003.

[12] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[13] J. Kolter and M. Maloof. Learning to detect malicious executables in the wild. *Proc. ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478, 2004.

[14] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1):51–56, 2004.

[15] C. Kruengkrai, V. Sornlertlamvanich, and H. Isahara. Language, Script, and Encoding Identification with String Kernel Classifiers. *Proc. Conference on Knowledge, Information and Creativity Support Systems*, 2006.

[16] W. Li, K. Wang, S. Stolfo, and B. Herzog. Fileprints: identifying file types by n-gram analysis. *Proc. IEEE Systems, Man and Cybernetics Information Assurance Workshop*, pages 64–71, 2005.

[17] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444, 2002.

[18] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 2003.

[19] D. Moore, C. Shannon, and k claffy. Code-red: a case study on the spread and victims of an internet worm. *Proc. ACM SIGCOMM Workshop on Internet measurment*, pages 273–284, 2002.

[20] B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *Network, IEEE*, 8(3):26–41, 1994.

[21] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1), 2001.

[22] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. *IEEE Symposium on Security and Privacy*, 2005.

[23] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[24] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. *Proc. International Conference on Data Mining*, pages 488–498, 2006.

[25] M. Rabin. Fingerprinting by random polynomials. Technical report, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[26] K. Rieck and P. Laskov. Detecting unknown network attacks using language models. *Proc. DIMVA*, pages 74–90, 2006.

[27] M. Roberts. Local-order-estimating Markovian analysis for noiseless source coding and authorship identification. Technical report, UCRL-53310, Lawrence Livermore National Lab., CA (USA), 1982.

[28] M. Roesch. Snort - lightweight intrusion detection for networks. *Proc. USENIX System administration*, pages 229–238, 1999.

[29] C. Shannon and D. Moore. The spread of the Witty worm. *Security & Privacy Magazine, IEEE*, 2(4):46–50, 2004.

[30] C. Sinclair, L. Pierce, and S. Matzner. An Application of Machine Learning to Network Intrusion Detection. *Proc. Computer Security Applications Conference*, page 371, 1999.

[31] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *Proc. Symposium on Operating Systems Design and Implementation*, 2004.

[32] E. Spafford. The internet worm program: an analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[33] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Professional, 2002.

[34] N. Weaver. Warhol Worms: The Potential for Very Fast Internet Plagues. *UC Berkeley, February*, 2002.

[35] C. Williams and D. Barber. Bayesian classification with Gaussian processes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(12):1342–1351, 1998.

[36] M. Zweig and G. Campbell. Receiver-operating characteristic plots: a fundamental evaluation tool in clinical medicine. *Clinical Chemistry*, 39(4):561–577, 1993.