Detecting Worms via Mining Dynamic Program Execution

Xun Wang, Wei Yu, Adam Champion, Xinwen Fu and Dong Xuan

***Abstract*— Worm attacks have been major security threats to the Internet. Detecting worms, especially new, unseen worms, is still a challenging problem. In this paper, we propose a new worm detection approach based on mining dynamic program executions. This approach captures dynamic program behavior to provide accurate and efficient detection against both seen and unseen worms. In particular, we execute a large number of real-world worms and benign programs (executables), and trace their system calls. We apply two classifier-learning algorithms (Naive Bayes and Support Vector Machine) to obtain classifiers from a large number of features extracted from the system call traces. The learned classifiers are further used to carry out rapid worm detection with low overhead on the end-host. Our experimental results clearly demonstrate the effectiveness of our approach to detect new worms in terms of a very high detection rate and a low false positive rate.**

***Index Terms*— Worm detection, system call tracing, dynamic program analysis, data mining**

I. Introduction

In this paper, we address issues related to detecting worms, especially new, unseen worms. Worms are malicious programs that propagate themselves on the Internet to infect computers by remotely exploiting vulnerabilities in those computers. Worm attacks have always been considered major threats to the Internet. There have been many cases of Internet worm attacks that caused significant damage, such as the "Code Red" worm in 2001 [1], the "Slammer" worm in 2003 [2], and the "Witty/Sasser" worms in 2004 [3]. For example, in November 2001, the Code Red worm infected more than $350,000$ computers in less than 14 hours by exploiting the buffer-overflow vulnerability in version 4.0 or 5.0 of Microsoft's Internet Information Services (IIS) web server, resulting in over $1.2 billion in damage.

After infecting a number of computers without being detected, the worm attacker can remotely control thems and use them as "stepping stones" to launch additional attacks. Consequently, as the first line of defense against worm attacks, worm detection research has become vital to the field of Internet security.

In general, there are two types of worm detection systems: network-based detection and host-based detection. Network-based detection systems detect worms primarily by monitoring, collecting, and analyzing the scan traffic (messages to identify vulnerable computers) generated by worm attacks. Many detection schemes fall into this category [4]–[7]. Nevertheless, because of their reliance on scan traffic, these schemes are not very effective in detecting worms that spread via email systems, instant messenger (IM) or peer-to-peer (P2P) applications.

On the other hand, host-based detection systems detect worms by monitoring, collecting, and analyzing worm behaviors on end-hosts. Since worms are malicious programs that execute on these machines, analyzing the behavior of worm executables[1] plays an important role in host-based detection systems. Mmany detection schemes fall into this category [8], [9]. Considering that a large number of real-world worm executables are accessible over the Internet, they provide an opportunity for researchers to directly analyze them to understand their behavior and, consequently, develop more effective detection schemes. Therefore, the focus of this paper is to use this large number of real-world worm executables to develop a host-based detection scheme which can *efficiently* and *accurately* detect new worms.

Within this category, most existing schemes have been focusing on *static properties* of executables [8], [9]. In particular, the list of called Dynamic Link Libraries (DLLs), functions and specific ASCII strings extracted from the executable headers, hexadecimal sequences extracted from the executable bodies, and other static properties are used to distinguish malicious and benign executables. However, using static properties without program execution might not accurately distinguish between these exectuables due to the following two reasons.

- First, two different executables (*e.g.*, one worm and one benign) can have same static properties, *i.e.*, they can call the same set of DLLs and even call the same set of functions.
- Second, these static properties can be changed by the worm writers by inserting "dummy" functions in the worm executable that will not be called during program execution, or by inserting *benign-looking* strings [10].

Hence, the *static properties* of programs, or *how they look*, are not the keys to distinguish worm and benign executables. Instead, we believe the keys are *what programs do*, *i.e.*, their *run-time behaviors* or *dynamic properties*. Therefore, our study adopts dynamic program analysis to profile the run-time behavior of executables to *efficiently* and *accurately* detect

Xun Wang, Adam Champion and Dong Xuan are with the Department of Computer Science and Engineering, The Ohio-State University, Columbus, OH 43210. E-mail: {wangxu, champion, xuan}@cse.ohio-state.edu. Wei Yu is with the Department of Computer Science, Texas A&M University, College Station, TX 77843. E-mail: {weiyu}@cs.tamu.edu. Xinwen Fu is with the College of Business and Information Systems, Dakota State University, Madison, SD 57042. E-mail: xinwen.fu@dsu.edu.

[1] In this paper, an *executable* means a binary that can be executed, which is different from program source code.

new worm executables. However, dynamic program analysis poses three challenges. First, in order to capture the *run-time* behavior of executables (both worm and benign ones), we have to execute a large number of malicious worms, which might damages our host and network systems. Second, given the large number of executables, manually executing and analyzing them is infeasible in practice. Hence, we need to find an *efficient* way to automatically capture programs' run-time behavior from their execution. Third, from the execution of a large set of various worms and benign executables, we need to find some constant and fundamental behavior differences between the worms and the benign executables in order to *accurately* determine whether an unseen executable is a worm or a benign executable.

In order to address the above issues, we propose an effective worm detection approach based on mining system-call traces of a large amount of real-world worms and benign executables. In particular, we set up virtual machines as the testbed on which we execute both worms and benign executables and analyze their execution. Since system-call tracing is an efficient and low-overhead method for studying of program execution dynamics, we adopt it to automatically record the program execution, then use the segments in traced system-call sequences as the detection features to capture executables' run-time behavior. Considering the large volume of system-call sequence segments traced from the large number of executables, we apply two data-mining classification algorithms to learn classifiers for distinguishing worm executables from benign ones. Then the learned classifiers are used to carry out run-time worm detection on the end-host.

Our extensive experiments with real executables demonstrate that our approach can achieve a very high detection rate and very low false positive rate in detecting new worms. Our approach is practical and has low overhead during both classifier learning and run-time detection. It does not require individual investigation for each executable, as it can automatically generate the worm detector (a classifier) based on existing worm sets for detection of new worms. Furthermore, we consider preventing worm writers from abusing our worm detection by generating a "black-box" classifier that is difficult to interpret. Our approach is an effective and efficient complement to other efforts in the worm detection field.

The remainder of the paper is organized as follows. In Section II, we discuss the background of worm detection. In Section III, we introduce the basic workflow of our approach. In Section IV, we introduce the dataset collection process. In Section V, we detail the extraction process of our detection features. In Section VI, we present the detail to classify and detect worms. In Section VII, we demonstrate the experimental results of our detection scheme. Section VIII further discuss some related issues. We present related work in Section IX. Finally, we conclude this paper in Section X.

II. Background

In this section, we give an overview of worm detection and then introduce program analysis and data mining techniques.

A. Worm Detection

Generally, worm detection can be classified into network-based and host-based schemes. Network-based schemes detect worm attacks by monitoring, collecting, and analyzing worm-generated traffic. For this purpose, Internet Threat Monitoring (ITM) systems have now been developed and deployed [11]. An ITM system usually consists of a number of monitors and a data center. Each monitor of an ITM system is responsible for monitoring traffic targeted to a range of unused, yet routable, IP address space and periodically reports the collected traffic logs to the data center. The data center analyzes the logs and posts summarized reports for alarming Internet worm attacks. Based on data reported by ITM systems, many detection schemes have been proposed [4], [5].

Host-based schemes detect worm attacks by monitoring, collecting, and analyzing the worm behavior on end-hosts. In particular, when a worm executes on an infected computer, it may take control of the system with high privileges, modify the system as needed, and continue to infect other computers. These acts expose some anomalies on the infected computers, such as writing or modifying registry keys and system binaries or opening network connections to transfer worm executables to other vulnerable computers. For example, the "Blaster" worm changes a registry entry, downloads a file named "`msblast.exe`", and executes it [12].

Traditional host-based detection focuses primarily on detecting worms by *signature matching*. In particular, these detection systems have a database of distinctive patterns (*signatures*) of malicious code for which they scan in possibly-infected systems. This approach is fast and, until recently, quite effective to detect known worms. However, it is not effective to detect *new* worms, as they have new signatures unknown to these detection systems during the worms' early propagation stage. Furthermore, worm writers can use the clear worm signatures generated or used by these detection systems to change the signatures in order to evade detection. For example, worms such as MetaPHOR [13] and Zmist [14] intensively metamorphose to hide themselves from detection, thereby illustrating the feasibility and the efficiency of mutation techniques.

Since attackers always want to hide their malicious actions, they do not make their attack source code publicly available. However, the attack executables are publicly available after the attacks are captured. Unlike classical host-based detection, our intention is to use a large number of real-world worm executables and further develop a generic detection scheme to detect new worms. For this purpose, dynamic program analysis plays an important role and is introduced in the following subsection.

B. Program Analysis

Unlike static program analysis, dynamic program analysis *does not* require the executable's source code, but dynamic analysis must be performed by executing the program [15], [16]. Most dynamic program analysis methods, such as debugging, simulation, binary instrumentation, execution tracing, stack status tracking, etc. are primarily used for software-

engineering and compiler-optimization purposes. Recently, interest in dynamic program analysis has arisen for vulnerability and "security hole"-detection purposes. However, some dynamic-analysis approaches are only suitable for analysis of individual executables with human expertise, such as debugging, or are only fit for specific attacks [17]. For our work, we need an appropriate dynamic program analysis method to investigate the run-time behavior of worm and benign executables to detect worms. The method we adopt here is to trace system calls during the program execution, which is a type of execution tracing. In particular, we trace the operating system calls invoked by the programs during their execution. This method can be used to automatically record interesting information during execution to further investigate executables' behavior in the course of worm detection.

C. Data Mining

Data mining refers to the process of extracting "knowledge," or meaningful and useful information, from large volumes of data [18], [19]. This is achieved by analyzing data from different perspectives to find inherent hidden patterns, models, relationships, or any other information that can be applied to new datasets. It includes algorithms for classification, clustering, association-rule mining, pattern recognition, regression, and prediction, among others. Data-mining algorithms and tools are widely adopted in a range of applications as well as in the computer-security field. In particular, various data-mining technologies are adopted in different threat-detection approaches as described in Section IX. In our work, we use classification algorithms to differentiate between worm and benign program execution in order to provide accurate worm detection against both seen and unseen worms.

III. Framework

A. Overview

Recall that the focus of this paper is to use a large number of real-world worm executables and subsequently develop an approach to detect new worms. In this section, we introduce the framework of our system for dynamic program analysis that detects worm executables based on mining system-call traces of a large amount of real-world worm and benign executables. In general, this mining process is referred to as the *off-line classifier learning process*. Its purpose is to *learn* (or *train*) a generic classifier that can be used to distinguish worm executables from benign ones based on system call traces. Then we use the learned *classifier* with appropriate classification algorithms to determine, with high accuracy, whether unknown executables belong to the worm class or the benign class. This process is referred to as the *on-line worm detection process*. The basic workflow is illustrated in Fig. 1 and Fig. 2, which is subsequently explained.

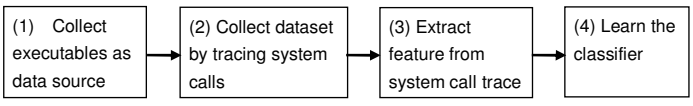

Fig. 1. Workflow of the off-line classifier learning

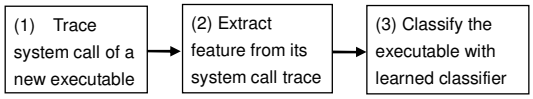

Fig. 2. Workflow of the on-line worm detection

B. Off-line Classifier Learning

1) Data Source Preparation: Before we can begin dynamic program analysis and profile the behavior of worm and benign executables, we need to collect a large number of such executables as the data source for our study. These executables are labeled into two classes: worm executables and benign executables. The worms are obtained from the Web site *VX Heavens* (`http://vx.netlux.org`).

2) Dataset Collection—Dynamic Properties of Executables: With the prepared data source, we discuss how to collect the dataset, which we refer to as the *dynamic properties* of executables. Recall that in order to accurately distinguish worm executables from benign ones, we need to collect data that can capture the fundamental behavior differences between them—the dynamic properties. One feasible and efficient method we choose is to run the executables and trace the run-time system-call sequences during their execution. However, executing worms might damage the host operating systems or even the computer hardware. In order to solve this problem in our experiments, we set up virtual machines as the testbed. Then we launch each executable in our data source and record its system-call trace during the execution on the virtual machine. We refer to the collection of the system-call traces for each executable in our data source as the *dataset*. We split the dataset into two parts: the *training set* and the *test set*. With the training set, we will apply classification learning algorithms to learn the classifier. The concrete format and content of the classifier is determined by the learning algorithms adopted. With the test set, we will further evaluate the accuracy of the learned classifier with respect to the classification of new and unidentified executables.

3) Feature Extraction: With the collection dataset comprising system-call traces of different executables, we extract all the system-call sequence segments with a certain length. These segments are referred as *n-grams*, where n is the length of the sequence, *i.e.*, the number of system calls in one segment. These n-grams can map to relatively independent and meaningful *actions* taken during the program execution, or the executables' *program blocks*. We intend to use these n-grams to capture the behaviors of common worms and benign executables. Hence these n-grams are the *features* for classifying worms and benign executables, and each distinct n-gram represents a particular feature in our classification.

4) Classifier Learning: From the features we extract from the training dataset, we need to learn a classifier to distinguish between worms and benign executables. When we select the classification algorithm, we need to consider the learned classifier's accuracy as well as its interpretability. Some classifiers are easy to interpret and the classification (*i.e.*, the *decision rule* of worm detection) can be easily extracted from the classifier [9]. Then worm writers can use the *rules* to change their

worms' behavior and consequently evade detection, similar to self-mutating worms that metamorphose to defeat signature-based detection [10]. Thus, we need classifiers with very low interpretability. In our case, we consider two algorithms, the Naive Bayes-based algorithm and the Support Vector Machine (SVM) algorithm, and compare their performance. While the Naive Bayes-based algorithm is simple and efficient in classifier learning, SVM is more accurate. More importantly, SVM learns a *black-box* classifier that is hard for worm writers to interpret.

C. On-line Worm Detection

Having learned the classifier in the off-line process, we now describe how it is used to carry out on-line worm detection. In this process, we intend to automatically detect a new and unseen executable. In particular, we follow the same procedure as in the off-line process, in which system-call traces of an unknown executable are recorded and classification features (*i.e.*, system-call sequence segments with certain lengths) are extracted during its execution. Then the classification algorithm is applied with the learned classifier to classify the new executable as a worm or a benign program.

In fact, the aforementioned worm detection actually depends on the accuracy of the classifier. In order to evaluate it, we use it to classify the executables in the *test set*. Since we already know the class label of these executables, we can simply compare the classification results from the learned classifier with the pre-known labels. As such, the accuracy of our classifier can be measured.

In the following sections, we will present the major steps above, *i.e.*, dataset collection, feature extraction, classifier learning and on-line worm detection in detail, followed by experiment results.

IV. Dataset Collection

In this section, we present the details on how we obtain the dataset, *i.e.*, the dynamic program properties of executables in the form of system call traces.

A. Worm Execution with Virtual Machine

In order to study the run-time behavior of worms and benign executables for worm detection, we need to execute the benign executables as well as the worms. However, worms might damage the operating system and even the hardware of the hosts. In order to solve this problem, we set up virtual machines (VMs) [20], [21] as the testbed. The VM we choose is *VMware* [20].

Even with VMs, two difficulties can still arise during data collection because of the worm execution. First, since worms can crash the operating system (OS) in the VM, we might have to repeatedly re-install the OS. In order to avoid these tedious re-installations, we install all necessary software for our experiments and store all our worm executables on the VM, and then save the image file for that VM. Whenever the VM OS crashes, we can clone the identical VM from the image file to continue our experiment. Second, it is difficult to obtain the system-call traces from the VM after it crashes. In order to solve this problem, we set the *physical machine* on which a VM is installed as the network neighbor of the VM through the virtual network. Thus, during worm execution, the VM automatically outputs the system-call trace to the physical machine. Although the physical machine can be attacked by the worms on the VM because of this virtual network, we protect the physical machine with anti-virus and other security software and impose very restrictive access controls.

B. System-Call Trace

Recall that we choose dynamic properties of executables to capture their behavior and, more accurately, distinguish worms from benign executables. There are multiple dynamic program analysis methods [15], [16] that can be used to investigate the dynamic properties of executables. The most popular methods are debugging and simulation. However, they must be used manually with human expertise to study program behavior. In our case, their human-intervention requirement makes them unsuitable for automatic analysis. Still, execution tracing is a good method for automatic analysis, as it can automatically record run-time behavior of executables. In addition, it is easy to analyze the system-call trace using automatic analysis algorithms.

There are several different ways to carry out execution tracing. In our case, we choose to trace system calls of worm and benign executables and use the trace to perform classification (and hence worm detection). The reasons for doing this are straightforward. Tracing all Microsoft Windows Application Programming Interface (API) functions can capture more details about the run-time behavior of executables. However, in comparison with tracing only system calls, API tracing increases OS resource consumption and interferes with the execution of other programs. This is because there are far fewer system calls (311 for all the Windows version together [22], 293 for the Linux 2.6 kernel [23]) than there are APIs (over 76,000 for Windows versions before Vista [24] and over 1,000 for Linux [25]). Hence, we choose to trace only system calls to facilitate "light-weight" worm detection.

V. Feature Extraction

Features are key elements of any anomaly-based detection or classification. In this section, we describe the method to extract and process the features that are used to learn the classifier and carry out worm detection.

A. N-gram from System-Call Trace

System-call traces of executables are the system-call sequences (time series) of the execution, which contain temporal information about the program execution and thus the respective dynamic behavior information. In our system, we need to extract appropriate features that can capture common or similar temporal information "hidden" in the system-call sequences of all worm executables, which is different from the temporal information hidden in the system-call sequences of all benign executables.

The n-gram is a well-accepted and frequently adopted temporal feature in various areas of (statistical) natural language processing and genetic sequence analysis [26]. It also fits our temporal analysis requirement. An n-gram is a subsequence of n items from a given sequence. For example, if a system call sequence is {NtReplyWaitReceivePortEx, NtOpenKey, NtReadVirtualMemory, NtCreateEvent, NtQuerySystemInformation}, then the 3-grams from this sequence are {NtReplyWaitReceivePortEx, NtOpenKey, NtReadVirtualMemory}, {NtOpenKey, NtReadVirtualMemory, NtCreateEvent}, and {NtReadVirtualMemory, NtCreateEvent, NtQuerySystemInformation}.

We use n-grams as the features in our system for the following reasons. Imagine the difference between one *line* of source code and one *block* of source code in a program. The line of code provides little meaningful information about a program, but the block of code usually represents a meaningful and self-contained small task in a program, which is the logical unit of programming. Similarly, one system call only provides very limited information about the behavior of an executable, whereas a segment of system calls might represent a meaningful and self-contained action taken during the program execution. Worm and benign executables have different behaviors, and this difference can be represented as the difference between their source code blocks, or the segments (*i.e.*, n-grams) of their system calls. Hence, we use these system-call segments, or the n-grams, as the features to classify worm and benign executables, which proves to be very effective throughout our experiments as described in Section VII.

B. Length of N-gram

A natural question is: what n-gram length is best for classifying worms from benign executables? On one hand, in order to capture the dynamic program behavior, n should be greater than 1. Otherwise, the extracted 1-gram list is actually the list of system calls invoked by the executables. This special case is the same as the method used by static program analysis to detect worms, which has no dynamic run-time information of executables.

On the other hand, n should not be very large for the following two reasons. First, if n is too large, it is very unlikely that we will find common or similar n-grams among different worm executables. In one extreme case, when n becomes very large, the n-grams are no longer *small tasks*. Instead, they encompass the entire execution of the programs. Because different worms cannot have the exact same sequence of system-call invocations (otherwise they are the same worm), the classifier learning algorithms cannot find a common feature (*i.e.*, the same system-call invocations) among them, and the algorithms cannot be used to define a class in which all the worms are included. In this case, the classification will not work. Second, if n is too large, the number of possible distinct n-grams—311^n for MS Windows as Windows has 311 system calls, 293^n for Linux as Linux has 293 system calls—will be too large to to be analyzed in practice. We investigate the impact of n-gram length on worm detection in our experiments and report the results in Section VII.

VI. Classifier Learning and Worm Detection

In this section, we describe the details of the last step in the off-line classifier learning process (*i.e.*, how to apply the classifier learning algorithm to learn the classifier after extracting the features). In particular, we use two classification algorithms: the Naive Bayes algorithm, which is a simple but popular learning algorithm, and the Support Vector Machine (SVM) algorithm, which is a more powerful but more computationally-expensive learning algorithm. We also discuss how to conduct on-line worm detection with each of the algorithms in detail.

A. Naive Bayes-based Classification and Worm Detection

The Naive Bayes classifier (also known as the Simple Bayes classifier) is a simple probabilistic classifier based on applying Bayes' Theorem [19]. In spite of its naive design, the Naive Bayes classifier may perform better than more sophisticated classifiers in some cases, and it can be trained very efficiently with a labeled training dataset. Nevertheless, in order to use the Naive Bayes classifier, one must make the assumption that the features used in the classification occur independently.

In our case, we use the Naive Bayes classifier to calculate the likelihood that an executable is a worm executable (*i.e.*, in the worm class) and the likelihood that it is a benign executable (*i.e.*, in the benign class). Then, based on which of the two classes have a larger likelihood, the detection decision is made.

1) Off-line Classifier Learning: We represent each executable by an m-dimensional feature vector, $\mathbf{X} = (x_1, x_2, \ldots, x_m)$, where m is the number of distinct n-grams in the dataset, x_i ($i = 0, \ldots, m-1$) is the i^{th} distinct n-gram such that $x_i = 1$ if x_i appears in the executable's system call trace and $x_i = 0$ otherwise. We have two classes: the worm class C_w and the benign class C_b. Given the feature vector $\mathbf{X}$ of an unknown executable, we need to predict which class $\mathbf{X}$ belongs to. The prediction is done as follows. First, for each class, we calculate the likelihood that the executable belongs to that class. Second, we make a decision based on the larger likelihood value, *i.e.*, the executable belongs to the class that has the larger likelihood.

Actually, the off-line "classifier" learning process of the Naive Bayes algorithm is the preparation for the calculation of the above two likelihoods. In particular, this preparation is the calculation of some statistical probabilities based on the training data. These probabilities comprise the posterior probability of each n-gram—say, x_i—conditioned on each class, C_w and C_b. Hence, the off-line "classifier" learning process in our Naive Bayes classification actually is the calculation of $P(x_i|C_j)$ $i = 1, \ldots, m, j = w$ or b based on the training dataset.[2]

[2]In some implementations, the classifier learning based on the Naive Bayes algorithm may conduct extra procedures, such as selection of features and cross-validation, but they are not the core procedures for the Naive Bayes algorithm.

2) On-line Worm Detection: During the on-line worm detection, for each unknown executable, the feature vector $\mathbf{X}$ for that executable is built first. Then we predict that $\mathbf{X}$ belongs to the class that has a higher posterior probability, conditioned on $\mathbf{X}$. That is, the Naive Bayes classifier assigns an unknown sample $\mathbf{X}$ to the class C_j if and only if

$$P(C_j|\mathbf{X}) > P(C_k|\mathbf{X}), \text{ where } j,\ k = w \text{ or } b, j \neq k. \quad (1)$$

Based on Bayes' Theorem, $P(C_j|\mathbf{X})$ can be calculated by

$$P(C_j|\mathbf{X}) = \frac{P(\mathbf{X}|C_j)P(C_j)}{P(\mathbf{X})}. \quad (2)$$

In order to predict the class of $\mathbf{X}$, we will calculate $P(\mathbf{X}|C_j)P(C_j)$ for $j = w$ or b and consequently compare $P(C_w|\mathbf{X})$ to $P(C_b|\mathbf{X})$. Now we discuss how to calculate $P(\mathbf{X}|C_j)P(C_j)$. First, if the class prior probabilities $P(C_w)$ and $P(C_b)$ are unknown, then it is commonly assumed that the classes are equally likely, *i.e.*, $P(C_w) = P(C_b)$. Otherwise, $P(C_j)$ can be estimated by the proportion of class C_j in the dataset. Second, as we assume the features are independent, $P(\mathbf{X}|C_j)$ can be calculated by

$$P(\mathbf{X}|C_j) = \prod_{i=1}^{m} P(x_i|C_j), \quad (3)$$

where $P(x_i|C_j)$ can be calculated during the off-line classifier learning process.

3) Discussion: The Naive Bayes classifier is effective and efficient in many applications. The theoretical time complexity for learning a Naive Bayes classifier is $O(Nd)$, where N is the number of training examples and d is the dimensionality of the feature vectors. The complexity of classification for an unknown example (an unknown executable in our case) is only $O(d)$.

However, the Naive Bayes classifier has two limitations in our case. First, worm writers can use it to make worm detection less effective for new worms. In our approach, it includes a set of probabilities that the n-grams appear in each class. Worm writers can directly use this information to make new worms similar to benign executables by either using or avoiding certain n-grams (system-call sequences). Second, high accuracy of the Naive Bayes classifier is based on the assumption that the features are independent of each other. However, in reality, the n-grams in the system-call trace of an executable may not be independent. In order to address these problems of Naive Bayes classifier, we use the Support Vector Machine (SVM) in our worm detection as described in the following subsection.

B. Support Vector Machine-based Classification and Worm Detection

The *Support Vector Machine* (SVM) is a type of learning machine based on statistical learning theories [27]. SVM-based classification includes two processes: classifier learning and classification. Classifier learning is the learning of a classifier or model using the training dataset. The learned classifier is used to determine or predict the class "label" of instances that are not contained in the training dataset. The SVM is a sophisticated and accurate classification algorithm. It is computationally expensive and its trained classifier is difficult to interpret. Its outstanding accuracy and low interpretability match our requirements for accurate worm detection and interpretation difficulty for worm writers.

1) Off-line Classifier Learning: A typical SVM classifier-learning problem is to label (classify) N training data $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ to positive and negative classes[3] where $\mathbf{x}_i \in \mathbf{R}^d$ $(i = 1, \ldots, N)$ and d is the dimensionality of the samples. Thus, the classification result is $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, $y_i \in \{-1, +1\}$. In our case, $\mathbf{x}_i$ is the feature vector built for the i^{th} executable in our dataset. That is, $\mathbf{x}_i = \{x_{i,1}, \ldots, x_{i,d}\}$, where d is the number of distinct n-grams, $x_{i,j}$ $(j = 1, \ldots, d)$ is the j^{th} n-gram, $x_{i,j} = 1$ if $x_{i,j}$ appears in the i^{th} executable's system call trace and $x_i = 0$ otherwise. $y_i = -1$ means that $\mathbf{x_i}$ belongs to the worm class and $y_i = +1$ means that $\mathbf{x}_i$ belongs to the benign-executable class.

There are two cases for the SVM classifier learning problems:

1) The samples in the two classes are linearly separable;
2) The samples in the two classes are *not* linearly separable.

Unfortunately, case (2) holds for most real-world problems. In the SVM, in order to learn an optimal classifier, the non-linearly-solvable problem in case (2) must be transformed into a linearly-solvable problem in case (1) first. Then the optimal classifier can be learned through linear optimization [27], [28]. In the following, we first present the algorithm for case (1) followed by the algorithm for case (2).

a) Classes are linearly separable: If the two classes are linearly separable, then we can find a hyperplane to separate the examples in two classes as shown in the right side of Fig. 3. Examples that belong to different classes should be located on different sides of the hyperplane. The intent of the classifier learning process is to obtain a hyperplane which can maximally separate the two classes.

Mathematically, if the two classes are linearly separable, then we can find a hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ with a vector $\mathbf{w}$ and an intercept b that satisfies the following constraints:

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq +1 \text{ for } y_i = +1 \text{ and} \quad (4)$$

$$\mathbf{w} \cdot \mathbf{x}_i - b \leq -1 \text{ for } y_i = -1, \quad (5)$$

or, equivalently,

$$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1 \leq 0 \ \forall i. \quad (6)$$

Examples in the training set that satisfy the above inequality are referred to as *support vectors*. The support vectors define two hyperplanes: one that goes through the support vectors of the positive class, and the other goes through the support vectors of the negative class. The distance between these two hyperplanes defines a margin and this margin is maximized when the norm of the vector $\mathbf{w}$, $\|\mathbf{w}\|$, is minimized. When the

[3]The SVM algorithm can be extended to classification for more than two classes, but the two classes are the typical and basic cases. Our problem is a two-class classification problem.

margin is maximized, the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ separates the two classes maximally, which is the optimal classifier in the SVM algorithm. The dual form of Equation 6 reveals that the above optimization actually is to maximize the following function:

$$W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{1} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j (\mathbf{x}_i \cdot \mathbf{x}_j) y_i y_j, \qquad (7)$$

subject to the constraint that $\alpha_i \geq 0$. The SVM algorithm can achieve the optimal classifier by finding out $\alpha_i \geq 0$ for each training sample $\mathbf{x_i}$ to maximize $W(\alpha)$.

b) Classes are not linearly separable: In the above case, linearly-separable classes can be optimized. However, real-world classification problems cannot usually be solved by the linear optimization algorithm. This case is illustrated as the left side of Fig. 3, in which there is no linear hyperplane (in this case, it is a straight line in 2-dimensional space) that can separate the examples in two classes (shown here with different colors). In other words, the required classifier must be a curve, which is difficult to optimize.

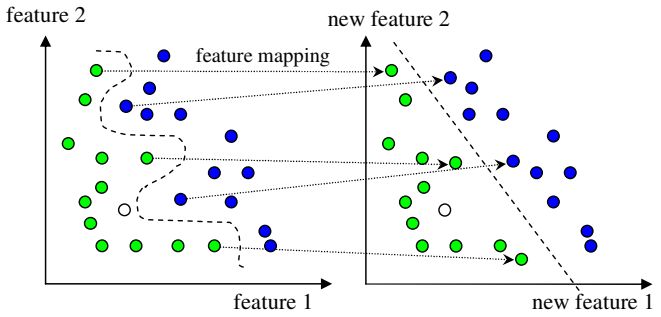

Fig. 3. Basic idea of kernel function in SVM.

The SVM provides a solution to this problem by transforming the original feature space into some other, potentially high-dimensional, *Euclidean* space. Then the *mapped* examples in the training set can be linearly separable in the new space, as demonstrated by the right side of Fig. 3. This space transformation in Equation (7) can be implemented by a *kernel function*,

$$K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j), \qquad (8)$$

where Φ is the mapping from the original feature space to the new Euclidean space. We would only need to use K in the classifier training process with Equation (7), and would never need to explicitly even know what Φ is. The SVM kernel function can be linear or non-linear. Common non-linear kernel functions include the Polynomial Function, Radial Basis Function (RBF), and Sigmoid Function, among others.

2) On-line Worm Detection: On-line worm detection is the classification of new executables using the SVM classification algorithm along with the optimal SVM classifier learned during the previously-discussed off-line learning process.

For an unknown executable (a worm or benign executable), its feature vector $\mathbf{x}_k$ must be built first. The method is the same as the aforementioned process on the executables in the training set, *i.e.*, the system-call trace during the execution is recorded, then the n-grams with a certain value of n are extracted. Afterwards, the feature vector $\mathbf{x}_k$ is formed from the trace of the executable using the same method as in the off-line classifier learning process.

Recall that during the classifier learning process, the optimal hyperplane is found. Then for a new example $\mathbf{x}_k$, shown as the white circle in Fig. 3, the on-line classification checks on which side of the optimal hyperplane $\mathbf{x}_k$ is. Mathematically, the classification is conducted through signing a class to the executable by

$$C(\mathbf{x_k}) = sign(\mathbf{w} \cdot \mathbf{x}_k - b), \qquad (9)$$

where

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i. \qquad (10)$$

If $C(\mathbf{x}_i)$ is positive, we predict that the executable is a worm. Otherwise, we predict that it is benign.

3) Complexity of SVM: The classifier learning process of SVM is relatively time-consuming because of the large volume of the training set, the high-dimensionality of our feature space, and the complexity of classifier calculation and optimization. No matter which kernel function is used for N training examples with feature vectors of dimensionality d and N_S support vectors, the SVM classifier learning algorithm has complexity $O(N_S^3 + N_S^2 N + N_S d N)$. However, the SVM classification process for each new executable is fast and involves only limited calculations. Its complexity is $O(MN_S)$, where M is the complexity of the kernel function operation. For Radial Basis Function (RBF) kernel functions, M is $O(d)$.

VII. Experiments

In this section, we first present our experimental setup and metrics, and then we report the results of our experiments.

A. Experiment Setup and Metrics

In our experiments, we use 722 benign executables and 1589 worms in Microsoft Windows or DOS Portable Executable (PE) format as the data source, though our approach works for worm detection on other operating systems as well. We use this data source to learn the generic worm classifier and further evaluate the trained classifier to detect worms. The executables are divided into two classes: worm and benign executables. The worms are obtained from the Web site *VX Heavens* (`http://vx.netlux.org`); they include e-mail worms, peer-to-peer (P2P) worms, Instant Messenger (IM) worms, Internet Relay Chat (IRC) worms and other, non-classified worms. The benign executables in our experiments include Microsoft software, commercial software from other companies and free, "open source" software. This diversity of executables enables us to comprehensively learn classifiers that capture the behavior of both worm and benign executables. We

use 80% of each class (worm and benign) as the training set to learn the classifiers. We use the remaining 20% as the test set to evaluate accuracy of the classifiers, *i.e.*, the performance of our detection approach.

We install MS Windows Professional 2000 with service pack 4 on our virtual machines (VMs). On these VMs, we launch each collected executable and use `strace` for Windows NT [29] to trace their system calls for 10 seconds.[4] From the trace file of each executable, we extract the system-call name sequences in temporal order. Then we obtain the segment of system calls (i.e., the n-grams), given different value of n for each executable. Afterwards, we build the vector inputs for the classification learning algorithms.

Recall that the classification in our worm detection problem is in a high-dimensional space. There are a large number of dimensions and features that cannot be handled or handled efficiently by many data-mining tools. We choose the following data-mining tools: Naive Bayes classification tools from University of Magdeburg in Germany [30] and `svm_light` [31]. Both tools are implemented in the C language and thus have efficient performance, especially for high-dimensional classification problems. When we apply SVM algorithm with `svm_light`, we choose the Gaussian Radial Basis Function (Gaussian RBF), which has been proven as an effective kernel function [19]. The feature distribution is a Gaussian distribution. The Gaussian RBF is in the form of

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}, \tag{11}$$

which means Equation (8) must be replaced by Equation (11) in the classifier learning process and on-line worm detection process. The value of γ is optimized through experiments and comparison.

In order to evaluate the performance of our classification for new worm detection, we use two metrics: *Detection Rate* (P_D) and *False Positive Rate* (P_F). In particular, the detection rate is defined as the probability that a worm is correctly classified. The false positive rate is defined as a benign executable classified mistakenly as a worm.

B. Experiment Results

In this subsection, we report the performance of our worm detection approaches. The results of Naive Bayes- and SVM-based worm detections in terms of *Detection Rate* and *False Positive Rate* under different n-gram length (n) are shown in Tables I and II, respectively. From these two tables, we make the following observations.

1) Effectiveness of Our Approaches: We conclude that our approaches of using both the Naive Bayes and SVM algorithms correlate with detected worms at a high detection rate and low false positive rate when the length of n-grams is reasonably large. For example, when the length of n-grams is 5, the detection based on the SVM algorithm achieves 99.5% detection rate and 2.22% false positive rate and the detection based on the Naive Bayes algorithm achieves 96.4% detection rate and 6.67% false positive rate, respectively.

We also conclude that SVM-based detection performs better than Naive Bayes-based detection in terms of both detection rate and false positive rate. There are two reasons for this. First, the Naive Bayes classification assumes that features are independent, which may not always be the case in reality. Second, the Naive Bayes-based classification calculates the likelihood for classifying a new executable based on the vectors of the training set executables in the feature space. Then it simply predicts the class of the new executable based on the likelihood comparison. In contrast, the SVM attempts to optimize the classifier (hyperplane) by finding the hyperplane that can maximally separate the two classes in the training set.

2) Impacts of N-gram Length: Another important observation is the length of n-gram, *i.e.*, the value of n, impacts the detection performance. When n increases from 1 to 4, the performance keeps increasing. When n further increases after that, the performance does not increase or only marginally increases. The reason can be explained as follows. First, when $n = 1$, each n-gram only contains one system call and thus contains neither dynamic system-call sequence information nor executable behavior information. Actually, this special case is that of static program analysis, which only investigates the list of system calls used by the executables. Second, when n is larger, the n-grams contain larger lengths of system call sequences, thereby capturing more dynamic behavior of the traced executables, hence increasing the detection performance. This also demonstrates that our dynamic program analysis approach outperforms the traditional static program analysis-based approaches.

From the above observation on the length of the n-gram, we conclude that a *certain n-gram length* is sufficiently effective for worm detection. This length (value of n) can be learned through experiments: when the increase of n does not greatly increase detection performance gain, that n value is *good enough* and can be used in practice. This method is actually used for other n-gram based data mining applications. Furthermore, with respect to the efficiency of worm detection, the n value should not be very large, as we discuss in Section VI.

VIII. Discussions

In this paper, we develop a worm detection approach that allows us to mine program execution, thereby detecting new and unseen worms. There are a number of possibilities for extending this work. A detailed discussion follows.

A. Classification for Different Worms

Presently, we discuss how to generalize our approach to support classification for different worms. Recall that our work in this paper only focuses on distinguishing two classes of executables: worm and benign. In practice, knowledge of the specific types of worms (*e.g.*, e-mail worms, P2P worms) can provide better ways to defend against them. In order to classify different worms, the approach studied in Section VI can be extended as follows. First, we collect the training dataset

[4] We launch the executables in the dataset for a longer time and then use a slide window to capture traces of a certain length for the classifier training. We found that a 10 second trace suffices to provide high detection accuracy.

TABLE I

DETECTION RESULTS FOR THE NAIVE BAYES-BASED DETECTION

n-gram length	1	2	3	4	5	6
Detection Rate (P_D)	69.8%	81.4.0%	85.0%	90.9%	93.6%	96.4%
False Positive Rate (P_F)	33.2%	18.6%	11.5%	8.89%	6.67%	6.67%

TABLE II

DETECTION RESULTS FOR THE SVM-BASED DETECTION

n-gram length	1	2	3	4	5	6
Detection Rate (P_D)	89.7%	96.0%	98.75%	99.5%	99.5%	99.5%
False Positive Rate (P_F)	33.3%	18.75%	7.14%	4.44%	2.22%	2.22%

including large number of worms labeled by their *types*. Second, we use the same approach discussed in Section VI to train the classifiers, which are capable of profiling multiple classes according to the *types*. Third, trained classifiers are used to determine the type (class) of an un-labeled new worm.

B. Integration of Network-based and Host-based Detection.

In this paper, our focus is the study of host-based detection and we did not consider information about the traffic generated by the executables during the worm detection. As we know, a worm executable will expose multiple behaviors, such as generating scan traffic (*i.e.*, messages that intend to identify vulnerable computers) and conducting malicious acts on the infected computers. Since these worm behaviors are exposed from different perspectives, consideration of multiple behaviors could provide more accurate worm detection. In fact, traffic generated by worms can also be classified and used to distinguish them from normal traffic. For instance, the distribution of destination IP addresses in network traffic can provide accurate worm detection through traffic analysis [7]. Hence one ongoing work is to combine the traffic logs and system calls generated by the worms and benign executables. The integration of traffic and system calls can learn more reliable classifiers to detect worms.

IX. RELATED WORK

In this section, we review some existing work related to our study including worm detection and data mining to the security research.

As we mentioned, there are two types of worm detection systems: network-based detection and host-based detection. Many network-based worm-detection schemes are proposed in the literature, which mainly focus on network traffic analysis. For example, Jung *et al*. in [32] developed a threshold-based detection algorithm to identify anomalous scan traffic generated by a computer. Venkataraman *et al*. and Weaver *et al*. in [4], [5] proposed schemes to examine statistics of scan traffic volume. Zou *et al*. presented a trend-based detection scheme to examine the exponential increase pattern of scan traffic [6]. Lakhina *et al*. in [33] proposed schemes to examine other features of scan traffic, such as the distribution of destination addresses. Other work studies worms that attempt to "take on" new patterns to avoid detection [7], [34].

Similarly, many host-based detection schemes have been proposed. For example, a binary text scan program was developed to extract the human-readable strings from the binary, which reveal information about the function of the executable binary [35]. Wagner *et al*. in [36] proposed an approach that analyzes program executables and generates a non-deterministic finite automaton (NDFA) or a non-deterministic pushdown automaton (NDPDA) from the global control-flow graph of the program. The automaton was then used to monitor the program execution on-line. Gao *et al*. in [37] presented an approach for detecting anomalous behavior of an executing process. The central idea of their approach is that processes potentially running the same executable should behave similarly in response to a common input. Feng *et al*. [38] proposed a formal analysis framework for pushdown automata (PDA) models. Based on this framework, they studied program analysis techniques, incorporating system calls or stack activities. Other schemes detect anomalous behavior of executables by examining call stack information. For example, Cowan *et al*. in [39] proposed a method called *StackGuard* to detect buffer overflow attacks. The difference that distinguishes our work form theirs is that we attempt to capture the common dynamic behavioral features of worms by mining the execution of a large number of worms.

Many articles have examined the use of data mining for security research. Lee *et al*. in [40] described a data-mining framework for adaptively building intrusion detection models. Martin *et al*. in [41] proposed an approach that learned statistical patterns of outgoing emails from local hosts. Yang *et al*. in [42] proposed an approach to apply machine learning to automatically fingerprint polymorphic worms, which are capable of changing their appearance every time they are executed. In our work, we use data mining to obtain the dynamic behaviorial difference between worms and benign executables.

X. FINAL REMARKS

In this paper, we proposed a new worm detection approach that is based on mining the dynamic execution of programs. Our approach is capable of capturing the dynamic behavior of executables to provide efficient and accurate detection against both seen and unseen worms. Using a large number of real-world worms and benign executables, we run executables on virtual machines and record system call traces. We apply two data mining classification algorithms to learn classifiers off-line, which are subsequently used to carry out on-line worm detection. Our data clearly show the effectiveness of

our proposed approach in detection worms in terms of both a very high detection rate and a low false positive rate.

Our proposed approach has the following advantages. It is practical with low overhead during both classifier learning and run-time detection. It does not rely on investigation for individual executable; rather, it examines the common dynamic properties of executables. Therefore, it can automatically detect new worms. Furthermore, our approach attempts to build a "black-box" classifier which makes it difficult for the worm writers to interpret our detection.

ACKNOWLEDGMENTS

We thank Chao Wang, Lei Ding, Yipeng Li and Yan Tang for their invaluable suggestions and comments on this work. We also thank the anonymous reviewers for their constructive feedbacks. This work is partially supported by NSF and ARO under grants No. CF-0546668 and AMSRD-ACC-R-50521-CI, respectively.

REFERENCES

[1] D. Moore, C. Shannon, and J. Brown, "Code-red: a case study on the spread and victims of an internet worm," in *Proceedings of the 2-th Internet Measurement Workshop (IMW)*, Marseille, France, November 2002.

[2] D. Moore, V. Paxson, and S. Savage, "Inside the slammer worm," *IEEE Magazine of Security and Privacy*, vol. 4, no. 1, pp. 33–39, July 2003.

[3] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage, "Opportunistic measurement: Extracting insight from spurious traffic," in *Proceedings of the 4-th ACM SIGCOMM HotNets Workshop (HotNets)*, College Park, MD, November 2005.

[4] J. Wu, S. Vangala, and L. X. Gao, "An effective architecture and algorithm for detecting worms with various scan techniques," in *Proceedings of the 11-th IEEE Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Febrary 2004.

[5] S. Venkataraman, D. Song, P. Gibbons, and A. Blum, "New streaming algorithms for superspreader detection," in *Proceedings of the 12-th IEEE Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Febrary 2005.

[6] C. Zou, W. B. Gong, D. Towsley, and L. X. Gao, "Monitoring and early detection for internet worms," in *Proceedings of the 10-th ACM Conference on Computer and Communication Security (CCS)*, Washington DC, October 2003.

[7] W. Yu, X. Wang, D. Xuan, and D. Lee, "Effective detection of active worms with varying scan rate," in *Proceedings of IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, Baltimore, MD, August 2006.

[8] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the 10th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, Seattle, WA, August 2004.

[9] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2001.

[10] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control flow graph matching," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Berlin, Germany, July 2006.

[11] SANS, *Internet Storm Center*, http://isc.sans.org/.

[12] CERT, *Advisory CA-2003-20 W32/Blaster worm*, http://www.cert.org /advisories/CA-2003-20.html, 2003.

[13] MetaPHOR, http://securityresponse.symantec.com/avcenter/venc/data/ w32.simile.html.

[14] P. Ferrie and P. Ször. Zmist, *Zmist opportunities*, Virus Bullettin, http://www.virusbtn.com.

[15] M. Ernst, "Static and dynamic analysis: Synergy and duality," Portland, Oregon, May 2003.

[16] Shengying Li, *A Survey on Tools for Binary Code Analysis*, Department of Computer Science, Stony Brook University, http://www.cs.sunysb.edu/ lshengyi/papers/rpe/RPE.htm, 2004.

[17] H. H. Feng, O. M. Kolesnikov, P.Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2003.

[18] M. H. Dunham, *Data Mining: Introductory and Advanced Topics*, Prentice Hall, 1 edition, 2002.

[19] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, 2 edition, 2006.

[20] VMWare Inc., *www.vmware.com/virtual-machine*.

[21] Microsoft, *Microsoft Virtual PC*, http://www.microsoft.com/windows /virtualpc/default.mspx.

[22] Metasploit LLC, *Windows System Call Table*, http://www.metasploit.com/users/opcode/syscalls.html.

[23] Operating System Inside, *Linux System Call Table*, http://osinside.net /syscall/system_call_table.htm, 2006.

[24] Paul Thurrott, *Windows "Longhorn" FAQ*, http://www.winsupersite.com /faq/longhorn.asp.

[25] GNU Project, *Linux Function and Macro Index*, http://www.gnu.org /software/libc/manual/html_node/Function-Index.html#Function-Index.

[26] K. F. Lee and S. Mahajan, *Automatic Speech Recognition: the Development of the SPHINX System*, Springer, 1988.

[27] V. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, New York, 1995.

[28] V. Vapnik, *Statistical Learning Theory*, John Wiley and Sons, New York, 1998.

[29] BindView Corporation, *Strace for NT*, http://www.bindview.com/ Services/RAZOR/Utilities/Windows/strace_readme.cfm.

[30] B. Christian, *Full and Naive Bayes Classifiers*, http://fuzzy.cs.uni-magdeburg.de/ borgelt/doc/bayes/bayes.html.

[31] T. Joachims, *Advances in Kernel Methods: Support Vector Machines*, MIT Press, Cambridge, Massachusetts, 1998.

[32] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Proceedings of the 25-th IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2004.

[33] A. Lakhina, M. Crovella, , and C. Diot, "Mining anomalies using traffic feature distribution," in *Proceedings of ACM SIGCOMM*, Philadelphia, PA, August 2005.

[34] R. Perdisci, O. Kolesnikov, P. Fogla, M. Sharif, and W. Lee, "Polymorphic blending attacks," in *Proceedings of the 15-th USENIX Security Symposium (SECURITY)*, Vancouver, B.C., August 2006.

[35] *Binary Text Scan*, http://netninja.com/files/bintxtscan.zip.

[36] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2001.

[37] D. Gao, M. Reiter, and Dawn Song, "Behavioral distance for intrusion detection," in *Proceedings of Symposium on Recent Advance in Intrusion Detection (RAID)*, Seattle, WA, September 1999.

[38] H. H Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2004.

[39] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stack-guard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of 7th USENIX Security Symposium (SECURITY)*, San Antonio, TX, August 1998.

[40] W. Lee, S. J. Stolfo, and W. Mok, "A data mining framework for building intrusion detection models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 1999.

[41] S. Martin, A. Sewani, B. Nelson, K. Chen, and A. Joseph, "Analyzing behavioral features for email classification," in *Proceedings of the 2th International conference on email and anti-span (CEAS)*, Mountain view, CA, August 2003.

[42] S. Yang, J. P. Song, H. Rajamani, T. W. Cho, Y. Zhang, and R. Mooney, "Fast and effective worm fingerprinting via machine learning," in *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)*, Dublin, Ireland, June 2006.