Using Text Categorization Techniques for Intrusion Detection

Yihua Liao, V. Rao Vemuri
Department of Computer Science
University of California, Davis
One Shields Avenue, Davis, CA 95616
{yhliao, rvemuri}@ucdavis.edu

Abstract

A new approach, based on the *k*-Nearest Neighbor (*k*NN) classifier, is used to classify program behavior as normal or intrusive. Short sequences of system calls have been used by others to characterize a program's normal behavior before. However, separate databases of short system call sequences have to be built for different programs, and learning program profiles involves time-consuming training and testing processes. With the *k*NN classifier, the frequencies of system calls are used to describe the program behavior. Text categorization techniques are adopted to convert each process to a vector and calculate the similarity between two program activities. Since there is no need to learn individual program profiles separately, the calculation involved is largely reduced. Preliminary experiments with 1998 DARPA BSM audit data show that the *k*NN classifier can effectively detect intrusive attacks and achieve a low false positive rate.

1 Introduction

Intrusion detection has played an important role in computer security research. Two general approaches to intrusion detection are currently popular: misuse detection and anomaly detection. In misuse detection, basically a pattern matching method, a user's activities are compared with the known signature patterns of intrusive attacks. Those matched are then labeled as intrusive activities. That is, misuse detection is essentially a model-reference procedure. While misuse detection can be effective in recognizing known intrusion types, it tends to give less than satisfactory results in detecting novel attacks.

Anomaly detection, on the other hand, looks for patterns that deviate from the normal (for example, [1, 2]). In spite of their capability of detecting unknown attacks, anomaly detection systems suffer from a basic difficulty in defining what is "normal". Methods based on anomaly detection tend to produce many false alarms because they are not capable of discriminating between abnormal patterns triggered by an otherwise authorized user and those triggered by an intruder [3].

Regardless of the approach used, almost all intrusion detection methods rely on some sort of usage tracks left behind by users. People trying to outsmart an intrusion detection system can deliberately cover their malicious activities by slowly changing their behavior patterns. Some examples of obvious features that a user can manipulate are the time of log-in and the command set used [4]. This, coupled with factors emanating from privacy issues, makes the modeling of user activities a less attractive option.

Learning program behavior and building program profiles is another possibility. Indeed building program profiles, especially those of privileged programs, has become a popular alternative to building user profiles in intrusion detection [5, 6, 7, 8]. Capturing the system call history associated with the execution of a program is one way of creating the execution profile of a program. Program profiles appear to have the potential to provide concise and stable descriptions of intrusion activity. To date, almost all the research in this area has been focused on using short sequences of system calls generated by individual programs. The local ordering of these system call sequences is examined and classified as normal or intrusive. There is one theoretical and one practical problem with this approach. Theoretically, no justification has been provided for this definition of "normal" behavior. Notwithstanding this theoretical gap, this procedure is tedious and costly. Although some automated tools may help to capture system call sequences, it is difficult and time consuming to learn individually the behavior profiles of all the programs (i.e., system programs and application programs). While the system programs are not generally updated as often as

the application programs, the execution traces of system programs are likely to be dynamic also, thus making it difficult to characterize "normality".

This paper treats the system calls differently. Instead of looking at the local ordering of the system calls, our method uses the frequencies of system calls to characterize program behavior for intrusion detection. This stratagem allows the treatment of long stretches of system calls as one unit, thus allowing one to bypass the need to build separate databases and learn individual program profiles. Using the text processing metaphor, each system call is then treated as a "word" in a long document and the set of system calls generated by a process is treated as the "document".This analogy makes it possible to bring the full spectrum of well-developed text processing methods [9] to bear on the intrusion detection problem. One such method is the *k*-Nearest Neighbor classification method [10, 11].

The rest of this paper is organized as follows. In Section 2 we review some related work. Section 3 is a brief introduction to the *k*NN text categorization method. Section 4 describes details of our experiments with the 1998 DARPA data. We summarize our results in Section 5, and Section 6 contains further discussions.

2 Related Work

Ko et al. at UC Davis first proposed to specify the intended behavior of some privileged programs (setuid root programs and daemons in Unix) using a program policy specification language [12]. During the program execution, any violation of the specified behavior was considered "misuse". The major limitation of this method is the difficulty of determining the intended behavior and writing security specifications for all monitored programs. Nevertheless, this research opened the door of modeling program behavior for intrusion detection. Uppuluri et al. applied the specification-based techniques to the 1999 DARPA BSM data using a behavioral monitoring specification language [13]. Without including the misuse specifications, they were able to detect 82% of the attacks with 0% false positives. The attack detection rate reached 100% after including the misuse specifications.

Forrest's group at the University of New Mexico introduced the idea of using short sequences of system calls issued by running programs as the discriminator for intrusion detection [5]. The Linux program *strace* was used to capture system calls. Normal behavior was defined in terms of short sequences of system calls of a certain length in a running Unix process, and a separate database of normal behavior was built for each process of interest. A simple table look-up approach was taken, which scans a new audit trace, tests for the presence or absence of new sequences of system calls in the recorded normal database for a handful of programs, and thus determines if an attack has occurred. Lee et al. [7] extended the work of Forrest's group and applied RIPPER, a rule learning program, to the audit data of the Unix *sendmail* program. Both normal and abnormal traces were used. Warrender et al. [6] introduced a new data modeling method, based on Hidden Markov Model (HMM), and compared it with RIPPER and simple enumeration method. For HMM, the number of states is roughly the number of unique system calls used by the program. Although HMM gave comparable results, the training of HMM was computationally expensive, especially for long audit traces. Ghosh and others [8] employed artificial neural network techniques to learn normal sequences of system calls for specific UNIX system programs using the 1998 DARPA BSM data. More than 150 program profiles were established. For each program, a neural network was trained and used to identify anomalous behavior.

Wagner et al. proposed to implement intrusion detection via static analysis [14]. The model of expected application behavior was built statically from program source code. During a program's execution, the ordering of system calls was checked for compliance to the precomputed model. Dynamic linking, thread usage and modeling library functions pose difficult challenges to static analysis. Another limitation of this approach is the running time involved in building models for individual programs from lengthy source code.

Unlike most researchers who concentrated on building individual program profiles, Asaka et al. [15] introduced a method based on discriminant analysis. Without examining all system calls, an intrusion detection decision was made by analyzing only 11 system calls in a running program and calculating the program's Mahalanobis' distances to normal and intrusion groups of the training data. There were four instances that were misclassified out of 42 samples. Due to its small size of sample data, however, the feasibility of this approach still needs to be established.

Ye et al. attempted to compare the intrusion detection performance of methods that used system call frequencies and those that used the ordering of system calls [16]. The names of system calls were extracted from the au-

dit data of both normal and intrusive runs, and labeled as normal and intrusive respectively. It is our impression that they did not separate the system calls based on the programs executing. Since both the frequencies and the ordering of system calls are program dependent, this oversimplification limits the impact of their work.

Our approach employs a new technique based on the *k*-Nearest Neighbor classifier for learning program behavior for intrusion detection. The frequencies of system calls executed by a program are used to characterize the program's behavior. Text categorization techniques are adopted to convert each process to a vector. Then the *k*-Nearest Neighbor classifier, which has been successful in text categorization applications, is used to categorize each new program behavior into either normal or intrusive class.

3 Review of K-Nearest Neighbor Text Categorization Method

Text categorization is the process of grouping text documents into one or more predefined categories based on their content. A number of statistical classification and machine learning techniques have been applied to text categorization, including regression models, Bayesian classifiers, decision trees, nearest neighbor classifiers, neural networks, and support vector machines [9].

The first step in text categorization is to transform documents, which typically are strings of characters, into a representation suitable for the learning algorithm and the classification task. The most commonly used document representation is the so-called vector space model. In this model, each document is represented by a vector of words. A word-by-document matrix **A** is used for a collection of documents, where each entry represents the occurrence of a word in a document, i.e., $\mathbf{A} = (a_{ij})$, where a_{ij} is the weight of word *i* in document *j*. There are several ways of determining the weight a_{ij}. Let f_{ij} be the frequency of word *i* in document *j*, *N* the number of documents in the collection, *M* the number of distinct words in the collection, and n_i the total number of times word *i* occurs in the whole collection. The simplest approach is Boolean weighting, which sets the weight a_{ij} to 1 if the word occurs in the document and 0 otherwise. Another simple approach uses the frequency of the word in the document, i.e., $a_{ij} = f_{ij}$. A more common weighting approach is the so-called *tf ·idf* (term frequency - inverse document frequency) weighting:

$$a_{ij} = f_{ij} \times log\left(\frac{N}{n_i}\right). \tag{1}$$

A slight variation [17] of the *tf ·idf* weighting, which takes into account that documents may be of different lengths, is the following:

$$a_{ij} = \frac{f_{ij}}{\sqrt{\sum_{l=1}^{M} f_{lj}^2}} \times log\left(\frac{N}{n_i}\right). \tag{2}$$

For matrix **A**, the number of rows corresponds to the number of words *M* in the document collection. There could be hundreds of thousands of different words. In order to reduce the high dimensionality, stop-word (frequent word that carries no information) removal, word stemming (suffix removal) and additional dimensionality reduction techniques, feature selection or reparameterization [9], are usually employed.

To classify a class-unknown document *X*, the *k*-Nearest Neighbor classifier algorithm ranks the document's neighbors among the training document vectors, and uses the class labels of the *k* most similar neighbors to predict the class of the new document. The classes of these neighbors are weighted using the similarity of each neighbor to *X*, where similarity is measured by Euclidean distance or the cosine value between two document vectors. The cosine similarity is defined as follows:

$$sim(X, D_j) = \frac{\sum_{t_i \in (X \cap D_j)} x_i \times d_{ij}}{\|X\|_2 \times \|D_j\|_2} \tag{3}$$

where *X* is the test document, represented as a vector; D_j is the *j*th training document; t_i is a word shared by *X* and D_j; x_i is the weight of word t_i in *X*; d_{ij} is the weight of word t_i in document D_j; $\|X\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2 + ...}$ is the norm of *X*, and $\|D_j\|_2$ is the norm of D_j. A cut-off threshold is needed to assign the new document to a known class.

The *k*NN classifier is based on the assumption that the classification of an instance is most similar to the classification of other instances that are nearby in the vector space. Compared to other text categorization methods such as Bayesian classifier, *k*NN does not rely on prior probabilities, and it is computationally efficient. The main computation is the sorting of training documents in order to find the *k* nearest neighbors for the test document.

We seek to draw an analogy between a text document and the sequence of all system calls issued by a process, i.e., program execution. The occurrences of system calls can be used to characterize program behavior and transform each process into a vector. Furthermore, it is assumed that processes belonging to the same class will

Table 1: Analogy between text categorization and intrusion detection when applying the *k*NN classifier.

Terms	Text categorization	Intrusion Detection
N	total number of documents	total number of processes
M	total number of distinct words	total number of distinct system calls
n_i	number of times *i*th word occurs	number of times *i*th system call was issued
f_{ij}	frequency of *i*th word in document *j*	frequency of *i*th system call in process *j*
D_j	*j*th training document	*j*th training process
X	test document	test process

cluster together in the vector space. Then it is straightforward to adapt text categorization techniques to modeling program behavior. Table 1 illustrates the similarity in some respects between text categorization and intrusion detection when applying the *k*NN classifier.

There are some advantages to applying text categorization methods to intrusion detection. First and foremost, the size of the system-call vocabulary is very limited. There are less than 100 distinct system calls in the DARPA BSM data, while a typical text categorization problem could have over 15000 unique words [9]. Thus the dimension of the word-by-document matrix **A** is significantly reduced, and it is not necessary to apply any dimensionality reduction techniques. Second, we can consider intrusion detection as a binary categorization problem, which makes adapting text categorization methods very straightforward.

4 Experiments

4.1 Data Set

We applied the *k*-Nearest Neighbor classifier to the 1998 DARPA data. The 1998 DARPA Intrusion Detection System Evaluation program provides a large sample of computer attacks embedded in normal background traffic [18]. The TCPDUMP and BSM audit data were collected on a network that simulated the network traffic of an Air Force Local Area Network. The audit logs contain seven weeks of training data and two weeks of testing data. There were 38 types of network-based attacks and several realistic intrusion scenarios conducted in the midst of normal background data.

We used the Basic Security Module (BSM) audit data collected from a victim Solaris machine inside the simulation network. The BSM audit logs contain information on system calls produced by programs running on the Solaris machine. See [19] for a detailed description of BSM events. We only recorded the names of system calls. Other attributes of BSM events, such as arguments to the system call, object path and attribute, return value, etc., were not used here, although they could be valuable for other methods.

The DARPA data was labeled with session numbers. Each session corresponds to a TCP/IP connection between two computers. Individual sessions can be programmatically extracted from the BSM audit data. Each session consists of one or more processes. A complete ordered list of system calls is generated for every process. A sample system call list is shown below. The first system call issued by Process 994 was *close*, *execve* was the next, then *open*, *mmap*, *open* and so on. The process ended with the system call *exit*.

Process ID: 994

close	*munmap*	*open*	*munmap*	*chmod*
execve	*mmap*	*mmap*	*open*	*close*
open	*mmap*	*mmap*	*ioctl*	*close*
mmap	*close*	*munmap*	*access*	*close*
open	*open*	*mmap*	*chown*	*close*
mmap	*mmap*	*close*	*ioctl*	*close*
mmap	*close*	*close*	*access*	*exit*

The numbers of occurrences of individual system calls during the execution of a process were counted. Then text weighting techniques were ready to transform the process into a vector. We used Equation (2) to encode the processes.

During our off-line data analysis, our data set included system calls executed by all processes except the processes of the Solaris operating system such as the inetd and shells, which usually spanned several audit log files.

Table 2: List of 50 distinct system calls that appear in the training data set.

access	chown	fchdir	getaudit	login	mmap	pipe	setaudit	setpgrp	su
audit	close	fchown	getmsg	logout	munmap	putmsg	setegid	setrlimit	sysinfo
audition	creat	fcntl	ioctl	lstat	nice	readlink	seteuid	setuid	unlink
chdir	execve	fork	kill	memcntl	open	rename	setgid	stat	utime
chmod	exit	fork1	link	mkdir	pathdonf	rmdir	setgroups	statvfs	vfork

4.2 Anomaly Detection

First we implemented intrusion detection solely based on normal program behavior. In order to ensure that all possible normal program behaviors are included, a large training data set is preferred for anomaly detection. On the other hand, a large training data set means large overhead in using a learning algorithm to model program behavior. There are 5 simulation days that were free of attacks during the seven-week training period. We arbitrarily picked 4 of them for training, and used the fifth one for testing. Our training normal data set consists of 606 distinct processes running on the victim Solaris machine during these 4 simulation days. There are 50 distinct system calls observed from the training data set, which means each process is transformed into a vector of size 50. Table 2 lists all the 50 system calls.

Once we have the training data set for normal behavior, the *k*NN text categorization method can be easily adapted for anomaly detection. We scan the test audit data and extract the system call sequence for each new process. The new process is also transformed to a vector with the same weighting method. Then the similarity between the new process and each process in the training normal process data set is calculated using Equation (3). If the similarity score of one training normal process is equal to 1, which means the system call frequencies of the new process and the training process match perfectly, then the new process would be classified as a normal process immediately. Otherwise, the similarity scores are sorted and the *k* nearest neighbors are chosen to determine whether the new program execution is normal or not. We calculate the average similarity value of the *k* nearest neighbors (with highest similarity scores) and set a threshold. Only when the average similarity value is above the threshold, is the new process considered normal. The pseudo code for the adapted *k*NN algorithm is presented in Figure 1.

In intrusion detection, the Receiver Operating Characteristic (ROC) curve is usually used to measure the performance of the method. The ROC curve is a plot of intrusion detection accuracy against the false positive probability. It can be obtained by varying the detection threshold. We formed a test data set to evaluate the performance of the *k*NN classifier algorithm. The BSM data of the third day of the seventh training week was chosen as part of the test data set (none of the training processes was from this day). There was no attack launched on this day. It contains 412 sessions and 5285 normal processes. The rest of the test data set consists of 55 intrusive sessions chosen from the seven-week DARPA training data. There are 35 clear or stealthy attack instances included in these intrusive sessions (some attacks involve multiple sessions), representing all types of attacks and intrusion scenarios in the seven-week training data. Stealthy attacks attempt to hide perpetrator's actions from someone who is monitoring the system, or the intrusion detection system. Some duplicate attack sessions of the types *eject* and *warezclient* were skipped and not included in the test data set. When a process is categorized as abnormal, the session that the process is associated with is classified as an attack session. The intrusion detection accuracy is calculated as the rate of detected attacks. Each attack counts as one detection, even with multiple sessions.

build the training normal data set D;
for each process X in the test data **do**
  if X has an unknown system call **then**
    X is abnormal;
  else then
    for each process D_j in training data **do**
      calculate $sim(X, D_j)$;
      if $sim(X, D_j)$ equals 1.0 **then**
        X is normal; exit;
    find k biggest scores of $sim(X, D)$;
    calculate sim_avg for k-nearest neighbors;
    if sim_avg is greater than $threshold$ **then**
      X is normal;
    else then
      X is abnormal;

Figure 1: Pseudo code for the *k*NN classifier algorithm for anomaly detection.

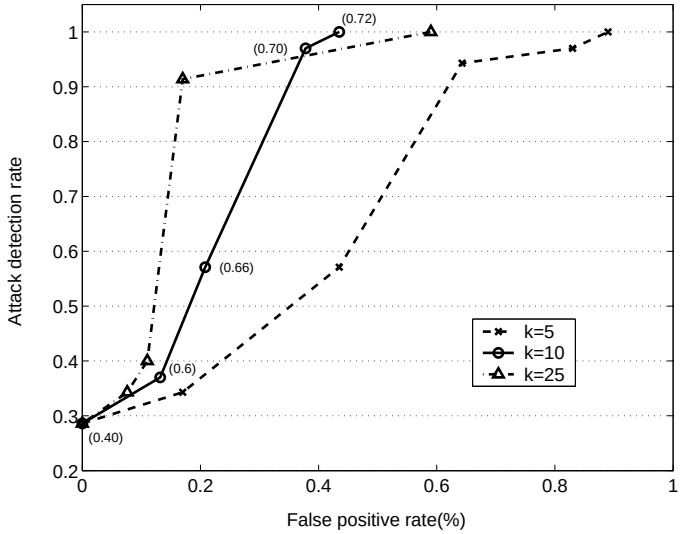

Figure 2: Performance of the *k*NN classifier method expressed in ROC curves. False positive rate vs attack detection rate for *k*=5, 10 and 25. Corresponding threshold values are shown in the parentheses for *k*=10.

Unlike the groups who participated in the 1998 DARPA Intrusion Detection Evaluation program [20], we define our false positive probability as the rate of mis-classified processes, instead of mis-classified sessions.

The performance of the *k*NN classifier algorithm also depends on the value of *k*, the number of nearest neighbors of the test process. Usually the optimal value of *k* is empirically determined. We varied *k*'s value from 5 to 25. Figure 2 shows the ROC curves for different *k* values. For this particular data set, *k*=10 is a better choice than other values in that the attack detection rate reaches 100% faster. For *k*=10, the *k*NN classifier algorithm can detect 10 of the 35 attacks with zero false positive rate. The detection rate reaches 100% rapidly when the threshold is raised to 0.72 and the false positive rate remains as low as 0.44% (23 false alarms out of 5285 normal processes) for the whole simulation day.

The RSTCORP group gave good performance during the evaluation of the 1998 DARPA BSM data [20]. By learning normal sequences of system calls for more than 150 programs, their Elman neural networks [8] were able to detect 77.3% of all intrusions with no false positives, and 100% of all attacks with about 10% miss-classified normal sessions, which means 40 to 50 false positive alarms for a typical simulation day with 500 sessions. Their test data consisted of 139 normal sessions and 22 intrusive sessions. Since different test data sets were used, it is difficult to compare the performance of our *k*NN classifier with that of the Elman networks. Although the *k*NN classifier has lower attack detection rate at zero false positive rate, the attack detection rate reaches 100% quickly, and hence a low false alarm frequency can be achieved.

4.3 Anomaly Detection Combined with Signature Verification

We have just shown that the *k*NN classifier algorithm can be implemented for effective abnormality detection. The overall running time of the *k*NN method is *O(N)*, where *N* is the number of processes in the training data set (usually *k* is a small constant). When *N* is large, this method could still be computationally expensive for some real-time intrusion detection systems. In order to detect attacks more effectively, the *k*NN anomaly detection can be easily integrated with signature verification. The malicious program behavior can be encoded into the training set of the classifier. After carefully studying the 35 attack instances within the seven-week DARPA training data, we generated a data set of 19 intrusive processes. This intrusion data set covers most attack types of the DARPA training data. It includes the most clearly malicious processes, including *ejectexploit*, *formatexploit*, *ffbexploit* and so on.

For the improved *k*NN algorithm, the training data set includes 606 normal processes as well as the 19 aforementioned intrusive processes. The 606 normal processes are the same as the ones in subsection 4.2. Each new test process is compared to intrusive processes first. Whenever there is a perfect match, i.e., the cosine similarity

Table 3: Attack detection rate for DARPA testing data (k=10 and *threshold*=0.8) when anomaly detection is combined with signature verification.

Attack	Instances	Detected	Detection rate
Known attacks	16	16	100%
Novel attacks	8	6	75%
Total	24	22	91.7%

is equal to 1.0, the new process is labeled as intrusive behavior (one could also check for near matches). Otherwise, the abnormal detection procedure in Figure 1 is performed. Due to the small amount of the intrusive processes in the training data set, this modification of the algorithm only causes minor additional calculation for normal testing processes.

The performance of the modified kNN classifier algorithm was evaluated with 24 attacks within the two-week DARPA testing audit data. The DARPA testing data contains some known attacks as well as novel ones. Some duplicate instances of the *eject* attack were not included in the test data set. The false positive rate was evaluated with the same 5285 testing normal processes as described in Section 4.2. Table 3 presents the attack detection accuracy for k=10 and the threshold of 0.8. The false positive rate is 0.59% (31 false alarms) when the threshold is adjusted to 0.8.

The two missed attack instances were a new denial of service attack, called *process table*. They matched with one of training normal processes exactly, which made it impossible for the kNN algorithm to detect. The *process table* attack was implemented by establishing connections to the telnet port of the victim machine every 4 seconds and exhausting its process table so that no new process could be launched [21]. Since this attack consists of abuse of a perfectly legal action, it did not show any abnormality when we analyzed individual processes. Characterized by an unusually large number of connections active on a particular port, this denial of service attack, however, could be easily identified by other intrusion detection methods.

Among the other 22 detected attacks, eight were captured with signature verification. These eight attacks could be identified without signature verification as well. With signature verification, however, we did not have to compare them with each of the normal processes in the training data set.

5 Summary

In this paper we have proposed a new algorithm based on the k-Nearest Neighbor classifier method for modeling program behavior in intrusion detection. Our preliminary experiments with the 1998 DARPA BSM audit data have shown that this approach is able to effectively detect intrusive program behavior. Compared to other methods using short system call sequences, the kNN classifier does not have to learn individual program profiles separately, thus the calculation involved with classifying new program behavior is largely reduced. Our results also show that a low false positive rate can be achieved. While this result may not hold against a more sophisticated data set, the k-Nearest Neighbor classifier appears to be well applicable to the domain of intrusion detection.

The $tf \cdot idf$ text categorization weighting technique was adopted to transform each process into a vector. With the frequency-weighting method, where each entry is equal to the number of occurrences of a system call during the process execution, each process vector does not carry any information on other processes. A new training process could be easily added to the training data set without changing the weights of the existing training samples. This could make the kNN classifier method more suitable for dynamic environments that require frequent updates of the training data.

In our current implementation, we used all the system calls to represent program behavior. The dimension of process vectors, and hence the classification cost, can be further reduced by using only the most relevant system calls.

6 Discussion

In spite of the encouraging initial results, there are several issues that require deeper analysis.

Our approach is predicated on the following properties: the frequencies of system calls issued by a program appear consistently across its normal executions and unseen system calls will be executed or unusual frequencies of the invoked system calls will appear when the program is exploited. We believe these properties hold true for many programs. However, if an intrusion does not reveal any anomaly in the frequencies of system calls, our method would miss it. For example, attacks that consist of abuse of perfectly normal processes such as *process table* would not be identified by the kNN clas-

sifier.

With the *k*NN classifier method, each process is classified when it terminates. We argue that it could still be suitable for real-time intrusion detection. Each intrusive attack is usually conducted within one or more sessions, and every session contains several processes. Since the *k*NN classifier method monitors the execution of each process, it is highly likely that an attack can be detected while it is in operation. However, it is possible that an attacker can avoid being detected by not letting the process exit. Therefore, there is a need for effective classification during a process's execution, which is a significant issue for our future work.

7 Acknowledgment

The authors wish to thank Dr. Marc Zissman of Lincoln Laboratory at MIT for providing us the DARPA training and testing data. We also thank the reviewers for their valuable comments. Special thanks to Dr. Vern Paxton for his insightful comments that helped us to improve the quality and readability of the final version. This work is supported in part by the AFOSR grant F49620-01-1-0327 to the Center for Digital Security of the University of California, Davis.

References

[1] H.S. Javitz and A. Valdes, *The NIDES Statistical Component: Description and Justification*, Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1994.

[2] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity", *Proceedings of 1989 IEEE Symposium on Security and Privacy*, 280-289, 1989.

[3] E. Lundin and E. Johnsson, "Anomaly-based intrusion detection: privacy concern and other problems", *Computer Networks*, vol. 34, 623-640, 2000.

[4] V. Dao and V. R. Vemuri, "Computer Network Intrusion Detection: A Comparison of Neural Networks Methods", *Differential Equations and Dynamical Systems*, (Special Issue on Neural Networks, Part-2), vol.10, No. 1&2, 2002.

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Logstaff, "A Sense of Self for Unix process", *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 120-128, 1996.

[6] C. Warrender, S. Forrest and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models", *Proceedings of 1999 IEEE Symposium on Security and Privacy*, 133-145, 1999.

[7] W. Lee, S. J. Stolfo and P. K. Chan, "Learning Patterns from Unix Process Execution Traces for Intrusion Detection", *Proceedings of AAAI97 Workshop on AI Methods in Fraud and Risk Management*, 50-56, 1997.

[8] A. K. Ghosh, A. Schwartzbard and A. M. Shatz, "Learning Program Behavior Profiles for Intrusion Detection", *Proceedings of 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 1999.

[9] K. Aas and L. Eikvil, *Text Categorisation: A Survey*, `http://citeseer.nj.nec.com/aas99text.html`, 1999.

[10] Y. Yang, *An Evaluation of Statistical Approaches to Text Categorization*, Technical Report CMU-CS-97-127, Computer Science Department, Carnegie Mellon University, 1997.

[11] Y. Yang, "Expert Network: Effective and Efficient Learning from Human Decisions in Text Categorization and Retrieval", *Proceedings of 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'94)*, 13-22, 1994.

[12] C. Ko, G. Fink and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring", *Proceedings of 10th Annual Computer Security Applications Conference*, Orlando, FL, Dec, 134-144, 1994.

[13] P. Uppuluri and R. Sekar, "Experiences with Specification-Based Intrusion Detection", *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS 2212, Springer, 172-189, 2001.

[14] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis", *Proceedings of IEEE Symposium on Research in Security and Privacy*, Oakland, CA, 2001.

[15] M. Asaka, T. Onabuta, T. Inoue, S. Okazawa and S. Goto, "A New Intrusion Detection Method Based on Discriminant Analysis", *IEEE TRANS. INF. & SYST.*, Vol. E84-D, No. 5, 570-577, 2001.

[16] N. Ye, X. Li, Q. Chen S. M. Emran and M. Xu, "Probabilistic Techniques for Intrusion Detection Based on Computer Audit Data", *IEEE Trans. SMC-A*, Vol. 31, No. 4, 266-274, 2001.

[17] J. T.-Y. Kwok, "Automatic Text Categorization Using Support Vector Machine", *Proceedings of International Conference on Neural Information Processing*, 347-351, 1998.

[18] MIT Lincoln Laboratory, `http://www.ll.mit.edu/IST/ideval/`.

[19] Sun Microsystems, *SunShield Basic Security Module Guide*, 1995.

[20] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Webber, S. Webster, D. Wyschograd, R. Cunninghan and M. Zissan, "Evaluating Intrusion Detection Systems: the 1998 DARPA off-line Intrusion Detection Evaluation", *Proceedings of the DARPA Information Survivability Conference and Exposition, IEEE Computer Society Press*, Los Alamitos, CA, 12-26, 2000.

[21] K. Kendall, "A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems", *Master's Thesis*, Massachusetts Institute of Technology, 1998.