Workshop on
Data Mining for Computer Security

www.cs.fit.edu/~pkc/dmsec03/

in conjunction with

IEEE International Conference on Data Mining

November 19-22, 2003

Melbourne, Florida

<u>Workshop Organizers</u>:

Philip Chan, Florida Tech

Vipin Kumar, University of Minnesota

Wenke Lee, Georgia Tech

Srinivasan Parthasarathy, Ohio State University

Program Committee:

- Wenke Lee, Georgia Tech (Co-Chair)
- Srinivasan Parthasarathy, Ohio State U (Co-Chair)
- Daniel Barbara, GMU
- Philip Chan, Florida Tech
- Eleazar Eskin, Hebrew U
- Wei Fan, IBM Watson
- Anup Ghosh, DARPA
- Sushil Jajodia, GMU
- Vipin Kumar, U. Minnesota
- Terran Lane, U. New Mexico
- Aleksandar Lazarevic, U. Minnesota
- Richard Lippmann, MIT Lincoln Lab
- Matthew Mahoney, Florida Tech
- Roy Maxion, CMU
- Chris Michael, Cigital
- R. Sekar, Stony Brook U
- Jaideep Srivastava, U. Minnesota
- Salvatore Stolfo, Columbia U
- Kymie Tan, CMU
- Alfonso Valdes, SRI

External Reviewers:

- Zoran Duric, George Mason University
- Eric Eilertson, University of Minnesota
- Levent Ertoz, University of Minnesota
- Amol Ghoting, Ohio State University
- Matthew Otey, Ohio State University
- Aysel Ozgur, University of Minnesota
- Joseph Pamula, George Mason University
- Xinzhou Qin, Georgia Institute of Technology
- Sankardas Roy, George Mason University

TABLE OF CONTENTS

INVITED TALKS

Authenticating Users by Profiling Behavior 1
Tom Goldring, NSA

Behavior-based Security 1
Salvatore J. Stolfo, Columbia University

ANOMALY DETECTION

One Class Support Vector Machines for Detecting Anomalous Windows Registry Accesses 2
Katherine Heller, Krysta Svore, Angelos Keromytis, and Salvatore Stolfo
[Columbia University]

One Class Training for Masquerade Detection 10
Ke Wang and Salvatore J. Stolfo [Columbia University]

Learning Rules from System Call Arguments and Sequences for Anomaly Detection 20
Gaurav Tandon and Philip Chan [Florida Institute of Technology]

Detection of Novel Network Attacks Using Data Mining 30
Levent Ertoz, Eric Eilertson , Aleksandar Lazarevic, Pang-Ning Tan,
Paul Dokas, Vipin Kumar, and Jaideep Srivastava [University of Minnesota]

FEATURE EXTRACTION

Passive Operating System Identification from TCP/IP Packet Headers 40
Richard Lippmann, David Fried, Keith Piwowarski, and William Streilein
[MIT Lincoln Laboratory]

Boundary Detection in Tokenizing Network Application Payload for Anomaly Detection 50
Rachna Vargiya and Philip Chan [Florida Institute of Technology]

MISUSE DETECTION

Detecting Privilege-Escalating Executable Exploits 60
Jesse C. Rabek, Robert K. Cunningham, and Roger I. Khazan
[MIT Lincoln Laboratory]

VISUALIZATION

A Prototype Tool for Visual Data Mining of Network Traffic for Intrusion Detection 67
William Yurcik, Kiran Lakkaraju, James Barlow and Jeff Rosendale
[NCSA/University of Illinois at Urbana-Champaign]

Introduction

Computer security is a broad field that encompasses issues both theoretical and practical aspects. It is of incredible importance to a wide variety of practical domains ranging from the banking industry to multi-national corporations, from space exploration to the intelligence community and so on. Computer security is frequently associated with three core areas: confidentiality, integrity and authentication. Although security policies and mechanisms address all three of these areas, they are not perfect and more and more organizations are becoming vulnerable to a wide variety of security breaches due to decreasing cost of the information processing and Internet accessibility. The most common security breaches include different cyber attacks to single computers, computer networks, wireless networks, databases or authentication compromises (e.g. masquerading).

The main aim of this workshop is to bring together leading figures from academia, government and industry to explore the applications of data mining in computer security.

Presentations in this workshop focus on several aspects of computer security, mainly in the area of intrusion detection. They are organized in the following four sessions:

- Anomaly Detection
- Feature Extraction
- Misuse Detection
- Visualization

The first session presents different data mining based anomaly detection techniques for recognizing novel and emerging computer attacks. Papers in the feature extraction session investigate various attributes that may be beneficial in data mining based techniques for intrusion detection. The paper in the misuse detection session presents a statistical based detector of malicious codes, while in the visualization session new data mining based prototype is presented to help security analysts to interactively assess security situational awareness of an entire network traffic.

The workshop program contains 8 papers selected from 17 submissions after a peer review process. Since two of the organizers (Chan and Kumar) submitted papers to the workshop, the other two organizers (Lee and Parthasarathy) organized the reviewing process and made the decisions on paper acceptance to avoid conflicts of interests. Three reviews were sought for each paper -- in select cases a fourth review was solicited either in the event of a missing review or in the event of low-confidence reviews. We like to thank the program committee members and external reviewers for their help in reviewing the submissions and providing comments for the authors. Special thanks are due to Xinzhou Qin (Georgia Tech) for setting up the workshop management system that facilitated paper submission and reviewing, and to Aleksandar Lazarevic (University of Minnesota) for workshop publicity as well as putting together the workshop proceedings. Lastly, we would like to express our appreciation to Tom Goldring for his invited talk on “Authenticating Users by Profiling Behavior” and Sal Stolfo on “Behavior-based Security”.

INVITED TALKS

Authenticating Users by Profiling Behavior

Tom Goldring, NSA

Building profiles of computer user activity entails collecting user session data, then learning models from this data, which can be used to classify new sessions. From the Computer Security viewpoint, the purpose would be to authenticate logins and detect insider misuse. A good data source will reflect user behavior and allow us to filter out system noise, both with a high degree of accuracy. Numerous published studies have used command line data, but this is probably no longer a viable source in today's environment.

The next step is feature selection, which allows us to choose among various existing classification algorithms to solve the authentication problem. But even the best algorithms will perform badly if the features are poor. Depending on what the data looks like, finding the right features and coaxing them into a usable form can be nontrivial. For nearly two years we have been monitoring "real" users on an operational Windows NT network that was part of a closed, internal network laboratory. In this talk we will describe our data, discuss the features we are currently using, and present results obtained to date.

Behavior-based Security

Salvatore J. Stolfo, Columbia University

Abstract. Behavior-based security systems are a new generation of computer security technologies that defend and protect critical IT assets by detecting deviations from a system's normal behavior. Behavior-based security systems provide the means of detecting attacks from remote sources, and from within, i.e. the insider problem.

The Email Mining Toolkit (EMT) is a data mining system that computes behavior profiles or models of user email accounts. These models may be used for a variety of forensic analyses and detection tasks. In this talk we describe the application of these models to detect the early onset of a viral propagation without "content-based" (or signature-based) analysis in common use in virus scanners. We present several experiments using real email from 15 users with injected simulated viral emails and describe how the combination of different behavior models improves overall detection rates. The performance results vary depending upon parameter settings, approaching 99% true positive (TP) (percentage of viral emails caught) in general cases and with 0.38% false positive (FP) (percentage of emails with attachments that are mislabeled as viral).

The principle behind behavior-based security is to model communication flows between systems and users, (possibly including content) using well grounded statistical techniques. The statistics gathered may be used to determine "social clique and communication communities" that typically exchange information, and the frequency of messages and the typical times and days those messages are exchanged. All this information can be used to model accounts, hosts or systems to determine typical behaviors that may be used to detect deviations of interest that may indicate misbehavior or security breaches.

We believe EMT thus serves as a model anomaly detection system for any audit stream and detection problem of interest. This work suggests a general framework that is the subject matter of our ongoing work. This framework posits that anomaly detection is best cast as a problem to optimally correlate multiple detectors, where each detector models normal behavior using different features of the audit stream. These detectors generate alerts when there are violations of volume and velocity statistics, anomalous values exhibited in an audit stream, and abnormal or inconsistent formation of vertices when viewing data in the audit stream in graph theoretic formulations. All of these concepts and modeling techniques are embodied in EMT.

One Class Support Vector Machines for Detecting Anomalous Windows Registry Accesses

Katherine A. Heller Krysta M. Svore Angelos D. Keromytis Salvatore J. Stolfo
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue
New York, NY 10025
{heller,kmsvore,angelos,sal}@cs.columbia.edu

Abstract

We present a new Host-based Intrusion Detection System (IDS) that monitors accesses to the Microsoft Windows Registry using Registry Anomaly Detection (RAD). Our system uses a one class Support Vector Machine (OCSVM) to detect anomalous registry behavior by training on a dataset of normal registry accesses. It then uses this model to detect outliers in new (unclassified) data generated from the same system. Given the success of OCSVMs in other applications, we apply them to the Windows Registry anomaly detection problem. We compare our system to the RAD system using the Probabilistic Anomaly Detection (PAD) algorithm on the same dataset. Surprisingly, we find that PAD outperforms our OCSVM system due to properties of the hierarchical prior incorporated in the PAD algorithm. In the future, these properties may be used to develop an improved kernel and increase the performance of the OCSVM system.

1. Introduction

One of the most popular and most often attacked operating systems is Microsoft Windows. Malicious software is often run on the host machine to inflict attacks on the system. Several methods can be used to combat malicious attacks, such as virus scanners and security patches. However, these methods are not able to combat unknown attacks, so frequent updates of the virus signatures and security patches must be made.

An alternative to these methods is a Host-based Intrusion Detection System (IDS). Host-based IDS systems detect intrusions on a host system by monitoring system accesses. Most IDS systems utilize signature based algorithms that rely on knowing the attacks and their signatures, which limits their ability to detect unknown attack methods. Alternatively, "behavior-blocking" technology aims to detect and stop malicious activities using a set of signature-based descriptions of good behavior, i.e. what is expected of program or system execution. To improve performance, data mining techniques have recently been applied to IDS systems [20, 22] to automatically learn models of "good behavior" and "bad behavior" by observing a system under normal operation. In this paper, we describe a new approach based on anomaly detection, utilizing a method that trains on normal data and looks for anomalous behavior that deviates from the normal model [11, 12, 13]. This method can better identify unknown attacks. Previous work using IDS systems has been done using system call analysis [14, 15, 17, 19, 24] and network intrusion detection [13, 18, 21].

We use the Registry Anomaly Detection (RAD) system to monitor Windows registry queries [9]. During normal computer activity, a certain set of registry keys are typically accessed by Windows programs. Users tend to use certain programs regularly, so registry activity is fairly regular and thus provides a good platform to detect anomalous behavior. We apply an OCSVM algorithm to the RAD system to detect anomalous activity in the windows registry. Although OCSVMs have previously been applied successfully to other anomaly detection problems, they have never before been used to detect anomalous accesses to the Windows registry. The OCSVM builds a model from training on normal data and then classifies test data as either normal or attack based on its geometrical deviation from the normal training data [23]. We present our results of the RAD system using the OCSVM algorithm and demonstrate its abilities to detect anomalous behavior with several different kernels. We also compare our system with work done on the RAD system using the Probabilistic Anomaly Detection (PAD) algorithm [14, 9]. PAD outperforms the OCSVM

system due to the use of the estimator developed by Friedman and Singer [16]. This estimator uses a Dirichlet-based hierarchical prior to smooth the distribution and account for the likelihoods of unobserved elements in sparse data sets by adjusting their probability mass based on the number of values seen during training. An understanding of the differences between these two models and the reasons for differences in detection performance may help to construct a more discriminative kernel, and is critical to the development of effective anomaly detection systems in the future.

2. The Windows Registry and the RAD system

The Windows registry is a database that stores configuration settings for programs, security information, user profiles, and many other system parameters. The registry consists of entries, which are called registry keys, and their associated values. Programs query the registry for information by accessing a specific registry key. Each registry query has five components: the name of the process, the type of query, an associated key, the result, and the success status of the query. The process may be an attack or normal process. Each record in both our test dataset and training dataset contains all five of these entries. A sample record entry appears as:

Process: `EXPLORER.EXE`
Query: `OpenKey`
Key: `HKCR\CLSID\B41DB860-8EE4-11D2-9906 -E49FADC173CA\shellex\MayChange DefaultMenu`
Response: `SUCCESS`
ResultValue: `NOTFOUND`

The Registry Anomaly Detection (RAD) system has three parts: an audit sensor, a model generator, and an anomaly detector. Each registry access is either stored as a record in the training set or sent to the detector for analysis by the audit sensor. The model generator develops a model of normal behavior from the training dataset, and the anomaly detector uses this model to classify new registry accesses as normal or anomalous.

The Registry Anomaly Detection (RAD) system utilizes the five raw features given above, such that the algorithm used for anomaly detection classifies each entry as either normal or attack according to these feature values. The process is the name of the process querying the registry. The query is the type of access being sent to the registry. The key is the key currently being accessed. The response is the outcome of the query. The value of the accessed key is the result value. For more detailed information on RAD and the Windows registry, refer to [9].

3. The PAD Algorithm

The Probabilistic Anomaly Detection (PAD) algorithm, developed by Eskin [14, 9], trains a model over normal data features. It is essentially density estimation, where the estimation of a density function $p(x)$ over normal data allows the definition of anomalies as data elements that occur with low probability. The detection of low probability data (or events) are represented as consistency checks over the normal data, where a record is labeled anomalous if it fails any one of these tests.

First and second order consistency checks are applied. First order consistency checks verify that a value is consistent with observed values of that feature in the normal data set. It computes the likelihood of an observation of a given feature, $P(X_i)$, where X_i are the feature variables. Second order consistency checks determine the conditional probability of a feature value given another feature value, denoted by $P(X_i|X_j)$, where X_i and X_j are the feature variables.

One way to compute these probabilities would be to estimate a multinomial that computes the ratio of the counts of a given element to the total counts. However, this results in a biased estimator when there is a sparse data set. Instead, the estimator given by Friedman and Singer is used to determine these probability distributions [16]. Let N be the total number of observations, N_i be the number of observations of symbol i, α be the "pseudo count" that is added to the count of each observed symbol, k^0 be the number of observed symbols, and L be the total number of possible symbols. Then the probability for an observed element i is given by:

$$P(X=i) = \frac{N_i+\alpha}{k^0\alpha+N}C \tag{1}$$

and the probability for an unobserved element i is:

$$P(X=i) = \frac{1}{L-k^0}(1-C) \tag{2}$$

where C, the scaling factor, accounts for the likelihood of observing a previously observed element versus an unobserved element. In [16], they compute C as:

$$C = (\Sigma_{k=k^0}^{L} \frac{k^0\alpha+N}{k\alpha+N} m_k)(\Sigma_{k\geq k^0} m_k)^{-1} \tag{3}$$

where $m_k = P(S=k)\frac{k!}{k=k^0}\frac{\Gamma(k\alpha)}{\Gamma(k\alpha+N)}$ and $P(S=k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability.

In PAD, however, the above computation of C is too costly, so a heuristic method is used, where C is given by:

$$C = \frac{N}{N+L-k^0} \tag{4}$$

They normalize the consistency check to account for the number of possible outcomes of L by considering if P is the probability estimated from the consistency check, then they report $log(P/(1/L)) = log(P) + log(L)$.

Since there are five feature values for each record in the RAD system, there are 5 first order consistency checks and 20 second order consistency checks. A record is labeled anomalous if any of the 25 consistency checks is below a given threshold. This method labels every record in the dataset as normal or anomalous. To improve the detection rate, pairs of features are examined since a record may have a set of feature values that are inconsistent even though all single feature values are consistent for that record. Most attacks effect a large number of records.

The PAD algorithm takes time $O(v^2 R^2)$, where v is the number of unique record values for each record component and R is the number of record components. The space required to run the algorithm is $O(vR^2)$.

4. One Class Support Vector Machine (OCSVM)

Instead of using PAD for model generation and anomaly detection, we apply an algorithm based on the one class SVM algorithm given in [23]. Previously, OCSVMs have not been used in Host-based anomaly detection systems. The OCSVM code was developed by [10] and has been modified to compute kernel entries dynamically due to memory limitations. The OCSVM algorithm maps input data into a high dimensional feature space (via a kernel) and iteratively finds the maximal margin hyperplane which best separates the training data from the origin. The OCSVM may be viewed as a regular two-class SVM where all the training data lies in the first class, and the origin is taken as the only member of the second class. Thus, the hyperplane (or linear decision boundary) corresponds to the classification rule:

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b \tag{5}$$

where $\mathbf{w}$ is the normal vector and b is a bias term. The OCSVM solves an optimization problem to find the rule f with maximal geometric margin. We can use this classification rule to assign a label to a test example $\mathbf{x}$. If $f(\mathbf{x}) < 0$ we label $\mathbf{x}$ as an anomaly, otherwise it is labeled normal. In practice there is a trade-off between maximizing the distance of the hyperplane from the origin and the number of training data points contained in the region separated from the origin by the hyperplane.

4.1. Kernels

Solving the OCSVM optimization problem is equivalent to solving the dual quadratic programming problem:

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j K(x_i, x_j) \tag{6}$$

subject to the constraints

$$0 \leq \alpha_i \leq \frac{1}{\nu l} \tag{7}$$

and

$$\sum_i \alpha_i = 1 \tag{8}$$

where α_i is a lagrange multiplier (or "weight" on example i such that vectors associated with non-zero weights are called "support vectors" and solely determine the optimal hyperplane), ν is a parameter that controls the trade-off between maximizing the distance of the hyperplane from the origin and the number of data points contained by the hyperplane, l is the number of points in the training dataset, and $K(x_i, x_j)$ is the kernel function. By using the kernel function to project input vectors into a feature space, we allow for nonlinear decision boundaries. Given a feature map:

$$\phi : X \rightarrow \mathbb{R}^N \tag{9}$$

where ϕ maps training vectors from input space X to a high-dimensional feature space, we can define the kernel function as:

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \tag{10}$$

Feature vectors need not be computed explicitly, and in fact it greatly improves computational efficiency to directly compute kernel values $K(x, y)$. We used three common kernels in our experiments:

Linear kernel: $K(x, y) = (x \cdot y)$
Polynomial kernel: $K(x, y) = (x \cdot y + 1)^d$, where d is the degree of the polynomial
Gaussian kernel: $K(x, y) = e^{-\|x-y\|^2/(2\sigma^2)}$, where σ^2 is the variance

Our OCSVM algorithm uses sequential minimal optimization to solve the quadratic programming problem, and therefore takes time $O(dL^3)$, where d is the number of dimensions and L is the number of records in the training dataset. Typically, since we are mapping into a high dimensional feature space d exceeds R^2 from the PAD complexity. Also for large training sets L^3 will significantly exceed v^2, thereby causing the OCSVM algorithm to be a much

more computationally expensive algorithm than PAD. An open question remains as to how we can make the OCSVM system in high bandwidth real time environments work well and efficiently. All feature values for every example must be read into memory, so the required space is $O(d(L+T))$, where T is the number of records in the test dataset. Although this is more space efficient than PAD, we compute our kernel values dynamically in order to conserve memory, resulting in the added d term to our time complexity. If we did not do this the memory needed to run this algorithm would would be $O(d(L+T)^2)$ which is far too large to fit in memory on a standard computer for large training sets (which are inherent to the windows anomaly detection problem).

5. Experiments and Results

The one class SVM system we develop detects abnormal accesses to the Windows registry. The training and testing datasets were developed from real usage of the Windows system, and each experiment took one to two weeks to run on a 1.5GHZ Pentium IV dual processor. The training data we used was collected on Windows NT 4.0 and consists of approximately 500,000 attack-free records. These attack-free records are labeled normal and consist of operating system programs and typical Windows programs. The test data consists of approximately 300,000 records of which approximately 2,000 are labeled attacks. Possible attacks include aimrecover, browslist, setuptrojan, and other publicly available attacks [1, 2, 3, 4, 5, 6, 7, 8].

We obtained kernels from binary feature vectors by mapping each record into a feature space such that there is one dimension for every unique entry for each of the five given record values. This means that a particular record has the value 1 in the dimensions which correspond to each of its five specific record entries, and the value 0 for every other dimension in feature space. We then computed linear kernels, second order polynomial kernels, and gaussian kernels using these feature vectors for each record.

We also computed kernels from frequency-based feature vectors such that for any given record, each feature corresponds to the number of occurences of the corresponding record component in the training set. For example, if the second component of a record occurs three times in the training set, the second feature value for that record is three. We then used these frequency-based feature vectors to compute linear and polynomial kernels.

To evaluate the system's accuracy, two statistics have been computed: detection rate and false positive rate. The detection rate is the percentage of attack records that have been correctly identified. The false positive rate is the percentage of normal records that have been mislabeled as anomalous. The threshold is the value that determines if a record is normal or attack. Table 1 includes a sample of the varying thresholds and their effects on the detection rate and false positive rate.

Threshold	False Positive Rate (%)	Detection Rate (%)
-1.08307	0.790142	0.373533
-1.08233	0.828005	0.480256
-1.07139	1.54441	0.533618
-0.968913	1.65734	1.17396
-0.798767	3.58736	3.89541
-0.79858	3.63784	5.60299
-0.798347	3.68999	6.77695
-0.767411	3.72054	6.83031
-0.746663	4.35691	7.47065
-0.746616	4.63025	8.00427
-0.71255	8.34283	20.9712
-0.712503	8.75201	22.0918

Table 1. The effects of varying the threshold on the false positive rate and the detection rate.

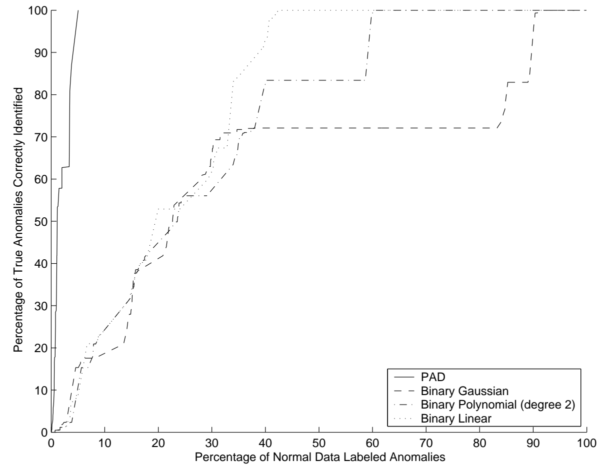

Figure 1. ROC curve for the kernels using binary feature vectors (false positives versus true positives).

We can measure the performance of the one class SVM on our test data by plotting its Receiver Operator Characteristic (ROC) curve. The ROC curve plots the percentage of false positives (normal records labeled as attacks) versus the percentage of true positives. As the discriminant threshold increases, more records are labeled as attacks. Random classification results in 50% of the area lying under the curve, while perfect classification results in 100% of the area lying under the curve. Results from our one class SVM system are shown with the results of the PAD system on the same dataset in Figures 1 and 2. Figure 1 is the ROC curve for the linear and polynomial kernels using binary feature

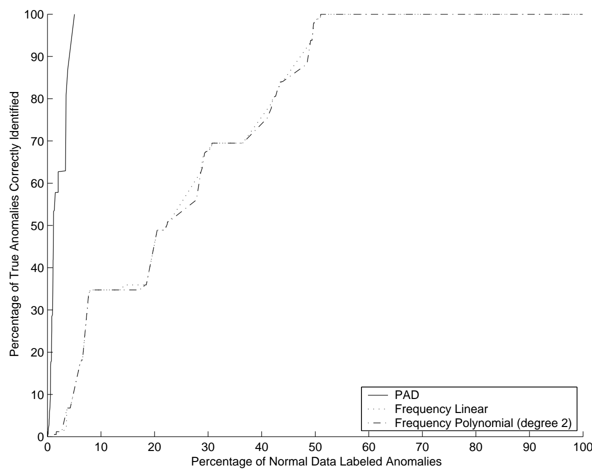

Figure 2. ROC curve for the kernels using frequency-based feature vectors (false positives versus true positives).

vectors. We have used a sigma value of 0.84 for our gaussian function. The binary linear kernel most accurately classifies the records. Figure 2 is the ROC curve for the linear and polynomial kernels using frequency-based feature vectors. The frequency-based linear and frequency-based polynomial kernels demonstrate similar classification abilities. Overall, in our experiments, the linear kernel using binary feature vectors results in the most accurate classification.

In Tables 2 and 3, information on the records and their discriminants are listed for the linear and polynomial kernels using binary feature vectors. From Table 2, it is seen that if the threshold is set at -1.423272, then the bo2kcfg.exe would be labeled as attack, as would msinit.exe and ononce.exe. False labels would be given to WINLOGON.exe, systray.exe and other normal records.

The results of the OCSVM system produce less accurate results than the PAD system demonstrated in [9, 14]. The PAD system is able to more accurately discriminate between normal and anomalous records. The OCSVM system labels records with fair accuracy, but could be improved with a stronger kernel, where more significant information is captured in the data representation.

The ability of the OCSVM to detect anomalies is highly dependent on the information captured in the kernel (the data representation). Our results show that kernels computed from binary feature vectors or frequency-based feature vectors alone do not capture enough information to detect anomalies as well as the PAD algorithm. With other choices of kernels, similar results will occur unless a novel technique which incorporates more discriminative information is used to compute the kernel. A simple example of this is if we have a dataset in which good discrimination depends upon pairs of features, then we will not be able to discriminate well with a linear decision boundary regardless of how we tweak its parameters. However, if we use a polynomial kernel we can account for pairs of features and will discriminate well. In this manner, having a well defined kernel which accounts for highly discriminative information is extremely important. For the purpose of this research, we believe our kernel choices are sufficient to reliably compare the OCSVM system with PAD.

The advantage of the PAD algorithm over the OCSVM system lies in the use of a hierarchical prior to estimate probabilities. A scaling factor (see equation (4)) is computed and applied to a Dirichlet prediction which assumes that all possible elements have been seen, giving varying probability mass to outcomes unseen in the training set. In general, knowing the likelihood of encountering a previously unencountered feature value is extremely important for anomaly detection, and it would be valuable to be able to incorporate this information into a kernel for use with our OCSVM system, perhaps by adding weighted "pseudo-counts" to the features in our frequency-based feature vectors.

6. Conclusions

By monitoring the Windows registry activity on a host system, we were able to use our OCSVM algorithm to label all records in the given experiments as either normal or attack with moderate accuracy and a low false positive rate. We have shown that since registry activity is regular, it can be used as a reliable anomaly detection platform. Note that it would also be informative to study detection rates for specific attack processes as a function of the discriminant threshold.

In the comparitive evaluation of our OCSVM system and the PAD system, we have shown that PAD is more reliable. However, understanding the reasons for this will lead to an improvement of the OCSVM system and will expedite the future development of anomaly detectors. Since there is currently no effective way to learn a "most optimal" kernel for a given dataset, we must rely on our domain knowledge in order to develop a kernel that leads to a highly accurate anomaly detection system. By analyzing algorithms (such as PAD) which currently discriminate well, we can identify information which is important to capture in our data representation and is crucial for the development of a more optimal kernel.

In the future, we plan on testing the system on file system accesses and on the Unix platform. We also plan to create a system to update the model as new data is labeled. This will help counter the effects of concept drift over time. Finding an efficient means of remodeling the data over time within

the OCSVM framework could improve the accuracy of the system.

Finally, since most users accept the default installation location when installing a program, the location of programs tends to be the same on all computers. Thus an attack does not need to query the registry for program location information. By forcing a location declaration other than the default location, a given program will not have the same location on all Windows machines. Attacks will have to query the registry to discover program locations, thus forcing all attacks to be monitored by the anomaly detector. A system such as this would improve the anomaly detection capabilities of the RAD system since no malicious attacks can bypass querying the registry. This would enhance the protection of the system against malicious users.

7. Acknowledgements

We would like to thank Eleazar Eskin, Shlomo Hershkop, Andrew Howard, and Ke Wang for their helpful comments. Katherine Heller was supported by an NSF graduate research fellowship. Krysta Svore was supported by an NPSC graduate fellowship.

References

[1] Aim recovery. URL: `http://www.dark-e.com/des/software/aim/index.shtml`.

[2] Back orifice. URL: `http://www.cultdeadcow.com/tools/bo.html`.

[3] Backdoor.xtcp. URL: `http://www.ntsecurity.new/Panda/Index.cfm?FuseAction=Virus\&VirusID=659`.

[4] Browselist. URL: `http://e4gle.org/files/nttools/,http://binaries.faq.net.pl/security_tools`.

[5] Happy99. URL: `http://www.symantex.com/qvcenter/venc/data/happy99.worm.html`.

[6] Ipcrack. URL: `http://www.geocities.com/SiliconValley/Garage/3755/toolicq.htmlhttp://home.swipenet.se/~w-65048/hacks.htm`.

[7] L0pht crack. URL: `http://www.astack.com/research/lc`.

[8] Setup trojan. URL: `http://www.nwinternet.com/~pchelp/bo/setuptrojan.txt`.

[9] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo. Detecting malicious software by monitoring anomalous windows registry accesses. *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, 2002.

[10] A. Arnold. Svm anomaly detection c code. *IDS Lab, Columbia University*, 2002.

[11] V. Bartnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons, 1994.

[12] M. DeGroot. *Optimal Statistical Decisions*. McGraw-Hill, New York, NY, 1970.

[13] D. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, SE-13:222–232, February 1987.

[14] E. Eskin. Anomaly detection over noisy data using learned probability distributions. *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000.

[15] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 120–128, 1996.

[16] N. Friedman and Y. Singer. Efficient bayesian parameter estimation in large discrete domains. *Advances in Neural Information Processing Systems*, 11, 1999.

[17] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[18] H. Javitz and A. Valdes. The nides statistical component: Description and justification. *Technical Report, SRI International, Computer Science Laboratory*, 1993.

[19] W. Lee, S. Stolfo, and P. Chan. Learning patterns from unix processes execution traces for intrusion detection. *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56, 1997.

[20] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. *IEEE Symposium on Security and Privacy*, pages 120–132, 1999.

[21] W. Lee, S. Stolfo, and K. Mok. Data mining in work flow environments: Experiences in intrusion detection. *Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999.

[22] M. Mahoney and P. Chan. Detecting novel attacks by identifying anomalous network packet headers. *Technical Report CS-2001-2*, 2001.

[23] B. Scholkopf, J. Platt, J. Shawe-Taylor, A. Smola, and R. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1472, 2001.

[24] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

Program Name	Label	Number of Records	Min. Record Value	Max. Record Value
REGMON.EXE	NORMAL	259	-0.794953	-0.280406
SPOOLSS.EXE	NORMAL	72	-1.152717	-0.021361
CloseKey	NORMAL	429	-1.082720	-0.374784
OpenKey	NORMAL	502	-0.959895	-0.365539
QueryValue	NORMAL	594	-1.082909	-0.374972
EnumerateValue	NORMAL	28	-0.570206	-0.284935
DeleteValueKey	NORMAL	3	-1.078758	-0.370822
AimRecover.exe	NORMAL	61	-1.082720	-0.374784
aim.exe	NORMAL	1702	-1.064796	-0.356860
ttssh.exe	NORMAL	12	-0.969706	-0.375161
ttermpro.exe	NORMAL	1639	-1.083098	-0.285123
NTVDM.EXE	NORMAL	271	-0.798204	-0.410065
notepad.exe	NORMAL	2673	-1.083098	-0.285123
CMD.EXE	NORMAL	116	-1.139322	-0.375161
TASKMGR.EXE	NORMAL	99	-0.570017	-0.284935
_INS0432._MP	NORMAL	443	-1.423272	-1.423272
WINLOGON.EXE	NORMAL	399	-1.423272	-1.423272
systray.exe	NORMAL	17	-1.423272	-1.423272
em_exec.exe	NORMAL	29	-1.423272	-1.423272
OSA9.EXE	NORMAL	705	-1.083098	-0.375161
fi ndfast.exe	NORMAL	176	-1.083098	-0.375161
WINWORD.EXE	NORMAL	1541	-1.083098	-0.375161
winmine.exe	NORMAL	21	-0.429351	-0.429351
POWERPNT.EXE	NORMAL	617	-1.083098	-0.285123
PING.EXE	NORMAL	50	-1.083098	-0.375161
QueryKey	NORMAL	11	-0.712317	-0.375161
wscript.exe	NORMAL	527	-1.083098	-0.375161
AcroRd32.exe	NORMAL	1598	-1.083098	-0.375161
0”	NORMAL	404	-1.083098	-0.375161
WINZIP32.EXE	NORMAL	3043	-1.083098	-0.375161
explore.exe	NORMAL	108	-1.083098	-0.375161
EXCEL.EXE	NORMAL	1782	-1.083098	-0.375161
bo2kss.exe[2]	ATTACK	12	-0.712317	-0.375161
bo2k_1_0_intl.e[2]	ATTACK	78	-1.083098	-0.375161
browselist.exe[4]	ATTACK	32	-0.798770	-0.411763
bo2kcfg.exe[2]	ATTACK	289	-1.423272	-1.423272
bo2k.exe[2]	ATTACK	883	-1.423272	-1.091776
mstinit.exe[2]	ATTACK	11	-1.423272	-1.423272
runonce.exe[2]	ATTACK	8	-1.423272	-1.423272
Patch.exe[2]	ATTACK	174	-1.083098	-0.375161
install.exe[3]	ATTACK	18	-1.083098	-0.375161
xtcp.exe[3]	ATTACK	240	-1.083098	-0.285123
l0phtcrack.exe[7]	ATTACK	100	-0.798581	-0.285123
LOADWC.EXE[2]	ATTACK	1	-1.423272	-1.423272
happy99.exe[5]	ATTACK	29	-0.570017	-0.411575

Table 2. Information about test records for the linear kernel in the binary setting. The maximum and minimum discriminants are given for each process, as well as the assigned classification label. Listed next to the attack processes is the attack source. [1] AIMCrack. [2] BackOrifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] L0pht Crack. [8] Setup Trojan.

Program Name	Label	Number of Records	Min. Record Value	Max. Record Value
REGMON.EXE	NORMAL	259	-4.062785	-1.524777
SPOOLSS.EXE	NORMAL	72	-5.422540	-0.272565
CloseKey	NORMAL	429	-5.210662	-1.788163
OpenKey	NORMAL	502	-4.828603	-1.758730
QueryValue	NORMAL	594	-5.211228	-1.789106
EnumerateValue	NORMAL	28	-3.311164	-1.542890
DeleteValueKey	NORMAL	3	-5.1955757	-1.766465
AimRecover.exe	NORMAL	61	-5.210285	-1.792879
aim.exe	NORMAL	1702	-5.148589	-1.703827
ttssh.exe	NORMAL	12	-4.860299	-1.794766
ttermpro.exe	NORMAL	1639	-5.211794	-1.543456
NTVDM.EXE	NORMAL	271	-4.234352	-1.794766
notepad.exe	NORMAL	2673	-5.211794	-1.543456
CMD.EXE	NORMAL	116	-5.388013	-1.794766
TASKMGR.EXE	NORMAL	99	-3.309843	-1.543456
_INS0432._MP	NORMAL	443	-6.239865	-6.239865
WINLOGON.EXE	NORMAL	399	-6.239865	-6.239865
systray.exe	NORMAL	17	-6.239865	-6.239865
em_exec.exe	NORMAL	29	-6.239865	-6.239865
OSA9.EXE	NORMAL	705	-5.211794	-1.789672
fi ndfast.exe	NORMAL	176	-5.211794	-1.794766
WINWORD.EXE	NORMAL	1541	-5.211794	-1.789672
winmine.exe	NORMAL	21	-1.794766	-1.794766
POWERPNT.EXE	NORMAL	617	-5.211794	-1.543456
PING.EXE	NORMAL	50	-5.211794	-1.789672
QueryKey	NORMAL	11	-4.022096	-1.789672
wscript.exe	NORMAL	527	-5.211794	-1.789672
AcroRd32.exe	NORMAL	1598	-5.211794	-1.794766
0”	NORMAL	404	-5.211794	-1.789672
WINZIP32.EXE	NORMAL	3043	-5.211794	-1.789672
explore.exe	NORMAL	108	-5.211794	-1.789672
EXCEL.EXE	NORMAL	1782	-5.211794	-1.789672
bo2kss.exe[2]	ATTACK	12	-4.022096	-1.789672
bo2k_1_0_intl.e[2]	ATTACK	78	-5.211794	-1.789672
browselist.exe[4]	ATTACK	32	-4.087124	-1.789672
bo2kcfg.exe[2]	ATTACK	289	-6.239865	-6.239865
bo2k.exe[2]	ATTACK	883	-6.239865	-5.245378
mstinit.exe[2]	ATTACK	11	-6.239865	-6.239865
runonce.exe[2]	ATTACK	8	-6.239865	-6.239865
Patch.exe[2]	ATTACK	174	-5.211794	-1.789672
install.exe[3]	ATTACK	18	-5.211794	-1.794766
xtcp.exe[3]	ATTACK	240	-5.211794	-1.543456
l0phtcrack.exe[7]	ATTACK	100	-4.194165	-1.543456
LOADWC.EXE[2]	ATTACK	1	-6.239865	-6.239865
happy99.exe[5]	ATTACK	29	-3.309843	-1.794766

Table 3. Information about test records for the second order polynomial kernel in the binary setting. The maximum and minimum discriminants are given, as well as the assigned classification label. Listed next to the attack processes is the attack source. [1] AIMCrack. [2] BackOrifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] L0pht Crack. [8] Setup Trojan.

One-Class Training for Masquerade Detection

Ke Wang Salvatore J. Stolfo
Computer Science Department, Columbia University
500 West 120th Street, New York, NY, 10027
{kewang, sal}@cs.columbia.edu

Abstract

We extend prior research on masquerade detection using UNIX commands issued by users as the audit source. Previous studies using multi-class training requires gathering data from multiple users to train specific profiles of self and non-self for each user. One-class training uses data representative of only one user. We apply one-class Naïve Bayes using both the multi-variate Bernoulli model and the Multinomial model, and the one-class SVM algorithm. The result shows that one-class training for this task works as well as multi-class training, with the great practical advantages of collecting much less data and more efficient training. One-class SVM using binary features performs best among the one-class training algorithms.

1. Introduction

The Masquerade attack may be one of the most serious security problems. It commonly appears as spoofing, where an intruder impersonates another person and uses that person's identity, for example, by stealing their passwords or forging their email address. Masqueraders can be insiders or outsiders. As an outsider, the masquerader may try to gain superuser access from a remote location and can cause considerable damage or theft. A simpler insider attack can be executed against an unattended machine within a trusted domain. From the system's point of view, all of the operations executed by an insider masquerader may be technically legal and hence not detected by existing access control or authentication schemes. To catch such a masquerader, the only useful evidence is the operations he executes, i.e., his behavior. Thus, we can compare one user's recent behavior against their profile of typical behavior and recognize a security breach if the user's recent behavior departs sufficiently from his profiled behavior, indicating a possible masquerader.

The insider problem in computer security is shifting the attention of the research and commercial community from intrusion detection at the perimeter of network systems. Research and development is going on in the area of modeling user behaviors in order to detect anomalous misbehaviors of importance to security; for example, the behavior of user-issued OS commands as represented in this paper, and in email communications [17]. Considerable work is ongoing in certain communities to detect not only impersonation, but also author identification. For example, Sedelow [16] and Vel [18] are two examples bracketing the length of time this topic has existed in the literature.

The masquerade problem is a challenging problem. If the masquerader can mimic the user's behavior successfully, he won't be detected. In addition, if the user himself is behaving much differently than his trained profile, the detector will misclassify him as masquerader, which may cause annoying false alarms. There have been several attempts to solve this problem using *command line sequences*, [14] and [9]. The best results so far reported are 60-70% accuracy with a false positive rate as low as 1-2%. The profiles were computed using supervised machine learning algorithms that classify training data acquired from multiple user. These approaches considered training user profiles as a multi-class supervised learning task where data gathered on a user is treated as an example of one-class, i.e. a distinct user.

In this paper, we consider a different approach with substantial practical advantage. We examine the task of profiling a user by modeling his data exclusively, without using examples from other users, and achieving good detection performance and minimal false positive rates. We also consider alternative machine learning algorithms that may be employed for this "one-class" training approach.

One-class training means that we *only* use the user's own legitimate examples of commands they issue to build the user's self profile. Previous work uses both positive and negative examples to build both self and non-self profiles, except for Maxion [9], who considers the problem of determining how vulnerable a user's behavior may be to mimicry attack. Here we extend this technique using one-class SVM. This is important in many contexts, especially when the only information available is the history of the user's activities. If a one-class training algorithm can achieve similar performance to that exhibited by a multi-class approach, we may provide a significant benefit in real security applications; much less data is required, and training can proceed independently of any other user. The study reported in this paper indicates that indeed one-class training algorithms perform equally well as two class training approaches.

This self profile idea is similar to the widely used "anomaly detection" techniques in intrusion detection system [eg. 2, 3]. For example, the anomaly detector of IDES [8] uses established normal usage profiles, which is the expected behavior, to identify any large usage deviation as a possible attack. Several methods have been used to model the normal data, for example, decision trees [7], neural network [4], and sparse Markov Transducers [2], and Markov chains [19]. In this paper, we applied one-class Naïve Bayes and one-class SVM algorithms to the masquerade dataset of UNIX system call sequences.

In previous work, we believe there were several methodological flaws in the manner in which data was acquired and used. The "Schonlau dataset" from [14] presents each user's command line data with a varying number of artificially created masquerade command blocks, ranging from 0 to 24, out of a total of 100 command blocks to be classified. The previous work only considered the average performance of a given method when it is applied to all of the 50*100 blocks of commands issued by the 50 users. However, since the masquerade blocks are "randomly" inserted into each user's data by using some other user's command block, each user's data has a different number of masquerade blocks, and the content of these masquerade blocks all differ. This data is not a good baseline to compare the effectiveness of alternative detection methods because one method might be better at detecting certain forms of masquerade attack while others are not. Unfortunately, since the distribution of such masquerade blocks appear many times in the dataset, some algorithms appear to have better performance over others, while, in practice or in other contexts, this finding may not be true. To better compare the alternative methods proposed in this work, we follow the exhaustive "1v49" evaluation methodology from [9], which will be described in detail in the section about the experimental methodology and results. The *ROC score* [5] is used to compare several one-class training methods under different false positive rate restrictions. This approach provides a better view to indicate which algorithm is better for most users.

Even though one machine learning algorithm may perform better than another in terms of detection and false positive rate, our most important finding is that one-class training can indeed perform as well as multi-class training. Even so, the overall performance of both approaches indicates that there is much room for improvement.

The rest of the paper is organized as follows. Section 2 describes the dataset we used in our research and provides a review of recent work, and Section 3 discusses the one-class Naïve Bayes and one-class SVM methods. Section 4 describes the experimental setting and our results, and Section 5 concludes the paper with our analysis and findings.

2. Recent Work

Schonlau *et al.* [14] and Maxion [9] provide a dataset collected from keyboard commands on a UNIX platform. The dataset is publicly available at http://www.schonlau.net. Unfortunately, there are neither timestamps nor other detailed command line arguments available in the Schonlau dataset. Hence, the dataset in question consists simply of sequential "truncated" commands issued by each of 70 users. Each user is represented by 15,000 commands, collected over time from a few days to a few months. They randomly chose 50 users to serve as intrusion target, and the remaining 20 users as masqueraders. The first 5,000 commands of each user are "clean data" (legitimately issued by the user), and the next 10,000 commands of the target users were randomly injected with commands issued by the 20 masquerade users, apparently none of whom are any of the 50 users. The commands are grouped into blocks, with 100 commands per block. The commands in one block are either all clean or all masquerade attacks, called "dirty blocks". The task studied in [14] is to accurately classify the "dirty" blocks as masquerade commands, while classifying the clean blocks as belonging to the user. From this point of view, this problem is very similar to a text classification problem, where the commands may be regarded as words or tokens in a document that is to be classified. Their work identifies six masquerade-detection schemes. They are: Bayes 1-Step Markov, Hybrid Multi-Step Markov, Incremental Probabilistic Action Modeling (IPAM), Uniqueness, Sequence-Match, and Compression. These masquerade detection methods were directly applied to this dataset; the first 5000 commands were used for training, and the remaining 10000 commands were inserted and used for testing, where each were processed in blocks of 100 commands. Testing and training were performed on a per block basis. The paper by Maxion [9] presents a clear and concise review of these six methods. The strategies behind each of these methods are quite different, but the results show that no method completely dominates another using cluster analysis and ROC curves.

Maxion [9] improved upon Schonlau's result by applying the Naïve Bayes classification algorithm using the "bag of words" features. Naïve Bayes has been used in text classification for a long time and proved to be very efficient in this context as well. Naïve Bayes was also used in the earlier work [15] on classifying malicious code attachments in email messages. Maxion presents a detailed analysis of the origins of the classification error, revealing why some users are good masquerades and others are not. That paper designed another experiment, called "1v49", to perform this error analysis. We also use the "1v49" experimental setting in our work, but here it is

used to compare the performance of different classifiers when applied to multiple classes.

The results for these reviewed methods are displayed in Table 1 and serve as a baseline for comparison.

Table 1. Results of previous classification methods

Method	Hits	False Positives
N. Bayes (updating)	61.5%	1.5%
N. Bayes (no Upd.)	66.2%	4.6%
Uniqueness	39.4%	1.4%
Hybrid Markov	49.3%	3.2%
1-step Markov	69.3%	6.7%
IPAM	41.4%	2.7%
Sequence Matching	36.8%	3.7%
Compression	34.2%	5.0%

3. Machine learning methods

3.1. Learning task

For this masquerade detection problem, the learning task is to build a classifier that can accurately detect the masquerade commands while not misclassifying the user's legitimate commands as a masquerade. Using the Schonlau dataset, which is organized as a set of blocks of 100 commands, the learning task is to compute a binary classifier whose input is a block of 100 commands and whose output is a classification of that block as either generated by a masquerader or not. The target classification is to detect the masquerader's command blocks. Hence, the masqueraders' data are positive examples, while the user's legitimate data are treated as negative examples. Thus, a true positive outcome is a masquerade block of 100 commands, while a false positive outcome is a block of commands legitimately issued by the user but misclassified as a masquerade. In the following description, we call the masquerade blocks *positive* examples and call the legitimate blocks, those issued by the user himself, *negative* examples. One-class training means that a classifier is computed using *only negative* examples of the user himself as training data to build the classifier, which will be used to classify both positive and negative data. Thus, the task is to positively identify masqueraders, but not to positively identify a particular user.

3.2. One-class or two class

Previous work considered the problem as a multi-class supervised training exercise. The dataset contains data for 50 users. For each user, a specific class, the first 5000 commands are treated *as negative examples*, while the data from the other 49 users are treated as *positive examples*. It is reasonable to assume the negative examples, which belong to the same user, were treated consistently, while the positive examples used in training belong to another user. For the masquerade problem, it is probably impossible and unreasonable to estimate how an attacker would behave. Thus, treating sets of other users' data as positive examples provides a substantive bias (to those users' behavior who probably was not behaving maliciously). We next present the means of implementing one-class training for Naïve Bayes classifier and for SVM, using only data from a single user when training a classifier to profile a distinct user.

3.3. Naïve Bayes Classifier

The Naïve Bayes classifier [12] is a simple and efficient supervised learning algorithm, which has been proved to be very effective in text classification, and many other applications. It is based on Bayes' rule,

$$p(u \mid d) = \frac{p(u)P(d \mid u)}{p(d)}$$

which calculates the probability of a class given an example. Applied to the masquerade problem, it calculates the likelihood that a command block belongs to a masquerader (non-self), or some legitimate user. Different commands c_i , which are used as features here, are assumed independent from each other. This is the Naïve part of this method.

There are two common models used in Naïve Bayes Classifier, one is the multi-variate Bernoulli model, and the other is the multinomial model [11]. In the multi-variate Bernoulli event model, a vector of binary attributes is used to represent a document (in our case, a block of 100 commands), indicating whether the command occurs or doesn't occur in the document. The multinomial model uses the number of command occurrences to represent a document, which is called "bag-of-words" approach, capturing the word frequency information in documents. According to McCallurn [11]'s result, multi-variate Bernoulli model performs better for small vocabulary size, and the multinomial model usually performs better at larger vocabulary size. Because the vocabulary size (the number of distinct commands) of this masquerade problem is 856, which is a moderate in size, we want to compare both of these models for this problem.

Multi-variate Bernoulli model

Using the multi-variate Bernoulli Model, a command block d is represented as a binary vector $\vec{d} = (b_1(d), b_2(d), ..., b_m(d))$, with $b_i(d)$ set to 1 if the command c_i occurs at least once in this block. Here m is the total number of features, i.e., the number of distinct commands. Given $p(c_i \mid u)$, which is the probability estimated for command c_i for user u in the training data, we can compute $p(d \mid u)$ of the test block d as:

$$p(d \mid u) = \prod_{i=1}^{m} (b_i(d) p(c_i \mid u) + (1 - b_i(d))(1 - p(c_i \mid u))) \quad (1)$$

where $p(c_i \mid u)$ is estimated with a Laplacean prior:

$$p(c_i \mid u) = \frac{1 + N(c_i, u)}{2 + N(u)} \quad (2)$$

$N(u)$ is the number of training examples for user u, while $N(c_i, u)$ is the number of documents containing the command c_i for user u.

Multinomial model

Using the standard bag-of-words approach, each command block is represented by a feature vector $\vec{d} = (n_1(d), n_2(d), ..., n_m(d))$, where $n_i(d)$ is the number of times command c_i appears in the command block d. Similarly, given $p(c_i \mid u)$, which is the frequency count computed for command c_i for user u in the training data, we can compute $p(d \mid u)$ of the test block d as:

$$p(d \mid u) = \prod_{i=1}^{m} (p(c_i \mid u))^{n_i(d)} \quad (3)$$

where $p(c_i \mid u)$ is derived from:

$$p(c_i \mid u) = \frac{\sum_{j=1}^{N(u)} n_i(d_j) + \alpha}{\sum_{i=1}^{m} \sum_{j=1}^{N(u)} n_i(d_j) + \alpha * m} \quad (4)$$

Here α is used for smoothing, which controls the sensitivity to previously unseen commands. (This implies there is a non-zero probability any command may be issued by any user.) We set it to 0.01 following [9].

One-class Naïve Bayes

Adapting the above algorithm to one-class Naïve Bayes, which uses only positive examples for training, is very simple. We only compute $p(c_i \mid u)$ for user u's self profile. For the non-self profile, we can assume each command has equal probability $1/m$, which is essentially random. Thus, given a test d, we can compare $p(d \mid self)$ with $p(d \mid nonself)$. The larger the *ratio* of $p(d \mid self)$ to $p(d \mid nonself)$, the more likely this command block d is from the user u.

Applying the one-class Naïve Bayes algorithm to our specific dataset is also quite simple. Since each test document (a block of commands) has a fixed number of 100 commands, the probability of "non-self" is the same for all tested blocks. Since we do not have to compute the probability of non-self; we may compare the probability of being "self" to a threshold in order to decide whether this block is a masquerade block or not. Furthermore, we can easily adjust the threshold to control the false positive and detection rate.

3.4. One-class support vector machine

Support Vector Machines (SVM) have been shown to be highly effective in text classification as well [6], among other important learning tasks. They are *maximal-margin* classifiers, rather than probabilistic as is Naïve Bayes. In the two-class formulation, the basic idea is to map feature vectors to a high dimensional space and to compute a hyperplane that not only separates the training vectors from different classes, but also maximizes this separation by making the *margin* as large as possible.

Scholkopf *et al.* [13] proposed a method to adapt the SVM algorithm for one-class SVM, which only use examples from one-class, instead of multiple classes, for training. The one-class SVM algorithm first maps input data into a high dimensional feature space via a kernel function and treats the origin as the only example from other classes. It then iteratively finds the maximal margin hyperplane that best separates the training data from the origin.

Considering that our training data set $x_1, x_2, ..., x_\ell \in X$, Φ is the feature mapping $X \to F$ to a high-dimensional space, we can define the kernel function as:

$$k(x, y) = (\Phi(x) \cdot \Phi(y))$$

Using kernel functions, the feature vectors need not be computed explicitly, greatly improving computational efficiency since we can directly compute the kernel values and operate on their images. Some common kernels are linear, polynomial, and radial basis function (rbf) kernels:

Linear Kernel: $k(x, y) = (x \cdot y)$

P-th order polynomial kernel: $k(x, y) = (x \cdot y + 1)^p$

rbf kernel: $k(x, y) = e^{-\|x-y\|^2 / 2\sigma^2}$

Now, solving the one-class SVM problem is equivalent to solving the dual quadratic programming (QP) problem:

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j k(x_i, x_j)$$

$$\text{subject to } 0 \le \alpha_i \le \frac{1}{\nu\ell}, \sum_i \alpha_i = 1.$$

where α_i is a Lagrange multiplier, which can be thought of as a weight on example x_i, and ν is a parameter that controls the trade-off between maximizing the number of data points contained by the hyperplane and the distance of the hyperplane from the origin.

After solving for α_i, we can use a decision function to classify data. The decision function is:

$$f(x) = \text{sgn}(\sum_i \alpha_i k(x_i, x) - \rho)$$

where the offset ρ can be recovered by

$$\rho = \sum_j \alpha_j k(x_j, x_i).$$

In our work, we used the LIBSVM 2.4 [1] available at http://www.csie.ntu.tw/~cjlin/libsvm for our experiments. LIBSVM is an integrated tool for support vector classification and regression that implemented Sholkopf's algorithm for one-class SVM. We used the default rbf kernel and the default values of the parameters for one-class SVM.

Another problem to consider for one-class SVM is how to represent the features. We used both a word count representation and a binary representation, which are equivalent to the multinomial model and multi-variate Bernoulli model of Naïve Bayes algorithm, respectively. The vectors are normalized to length 1.

4. Evaluation

We conducted two sets of experiments. The first experiment repeats the experimental methodology of [14]. We show that the performance of one-class training is almost the same as the performance of multi-class training. This is a significant finding on its own.

The second experiment aims to compare the performance of the two one-class training algorithms when applied to multiple users. Following [9], we will call the first the SEA experiment, which is from the authors' names in [14], Schonlau *et al*. The second experiment is called 1v49, because we trained using only one user's data and tested on all other 49 user's data.

4.1. SEA Experiment

Recall that in this experiment, the first 5,000 commands of a user serve as positive examples, and the first 5,000 commands of all the other 49 users serve as negative examples. The resultant classifier is tested on the rest of the 10,000 commands of the user. These have inserted "dirty" command blocks under a probability distribution from other users besides the 50 users whose commands were trained on. For our one-class algorithm, the test data remains the same, but we only use one user's training data (the first 5,000 commands) as negative examples for training; there are no positive examples used at all to compute the classifier. First we compare the Naïve Bayes classifier using the multi-variate Bernoulli model and the multinomial model. Then we compare the one-class training algorithms against the multi-class training algorithms. All the classification algorithms we tested do no updating during the test of any parameters because we want to compare the pure performance of each of them. Maxion [9] used an empirically pre-fixed threshold to update the classifier during the experiment, which is not an intrinsic characteristic of the classification algorithm.

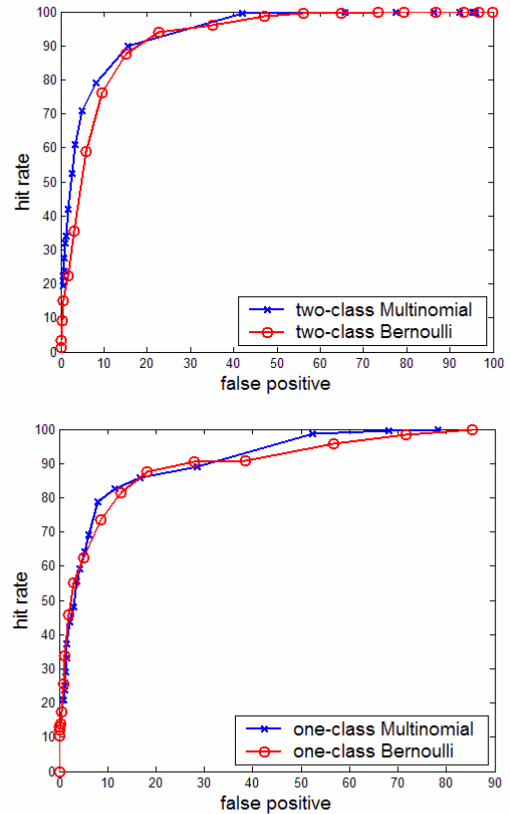

Figure 1. ROC Curves of Naïve Bayes classifiers using multinomial model and multi-variate Bernoulli model, two class and one-class training, respectively.

Figure 1 displays plots comparing the multi-variate Bernoulli model and the multinomial model of Naïve Bayes classifier. When using multi-class training, the multinomial model is obviously better than the Bernoulli model. But the difference is not so obvious in one-class training, especially when the false positive rate is low. We

thus compare both models in the following 1v49 experiment.

To compare the performance of the one-class training algorithms against the multi-class training algorithm on the same test data, we plot the ROC curves as displayed in Figure 1. For the multi-class training algorithm, we only use the multinomial model Naïve Bayes algorithm as the baseline for comparison, which is better than Bernoulli model and has been proved to the best among the variety of methods as described in [9]. For the one-class SVM, we compare both the binary and word count representations. From Figure 2, we can see that only one-class SVM using the word count representation is a little bit worse than the other three methods. One-class SVM using the binary representation and one-class Naïve Bayes achieved almost the same performance as the two class Naïve Bayes algorithm.

We also compare in Figure 3 the performance of all the previous algorithms from Table 1 to one-class SVM algorithm using binary features, which is best one among the one-class training algorithms. One-class SVM-binary is better than most of the previous algorithms except the two-class multinomial Naïve Bayes algorithm with updating.

This experiment confirmed our conjecture that for masquerade detection, one-class training is as effective as two class training.

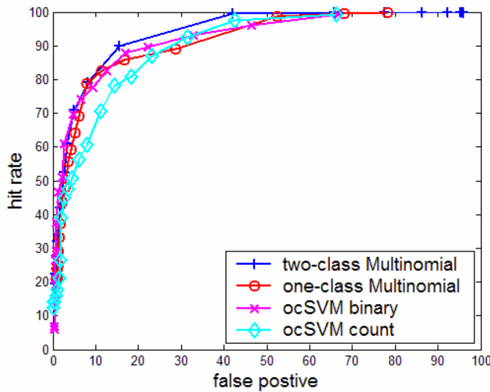

Figure 2. ROC curves of different classification algorithms in SEA experiment.

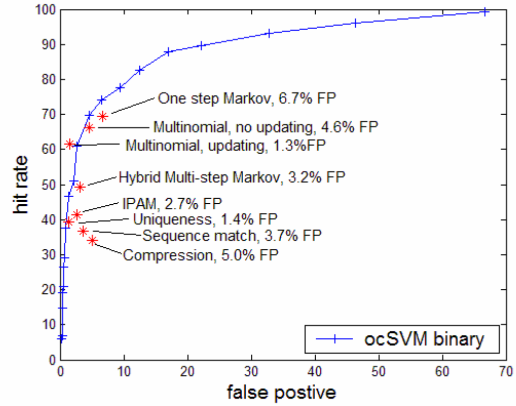

Figure 3. ROC curve for one-class SVM using binary feature. The best-outcome results from other methods are also included for comparison.

4.2. 1v49 Experiment

As we have pointed out, since the dataset used had randomly inserted masquerade blocks in each user's test commands (10,000 commands following the first 5,000), each user has a different number of "dirty" blocks and the origins of these "dirty" blocks also differ. So the result of the SEA experiment may not illustrate the real performance of a classification algorithm. (There are too many unfixed parameters.) To better evaluate the performance of a classification algorithm, we can treat these 50 users as our selected sample of common users. If we can prove algorithm A is better than algorithm B for most of the 50 users, we can infer A is better than B in a general sense.

To meet this requirement, we follow the "1v49" experiment, but for a different purpose. We use one user's first 5,000 commands as *negative* training data to compute a classifier without any positive training data. For test data, we use the non-masquerade blocks from the 10,000 additional commands of the same user as negative test data, and the other 49 users' first 5,000 commands as positive test data. This data is also organized in blocks of 100 commands.

As we mentioned before, the same algorithm might perform quite differently for different users. Figure 4 illustrates the difference. Figure 4 shows the ROC curve for user 2, 20 and 40 using one-class SVM with the binary feature representation. Such a difference occurs no matter which algorithm has been used; the difference is determined by the characteristic of each user.

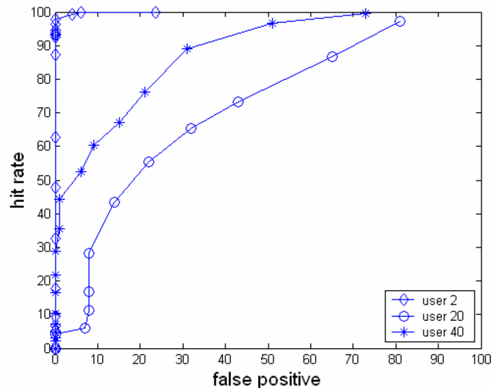

Figure 4: ROC curves for different users using one-class SVM-binary algorithm.

To compare the different methods for multiple users, we compute the *ROC score* for each user. In general, a ROC score is the fraction of the area under the ROC curve, the larger the better. A ROC score of 1 means perfect detection without any false positives. Figure 5 below shows the ROC scores for users 20 and 40 using the one-class SVM-binary algorithm.

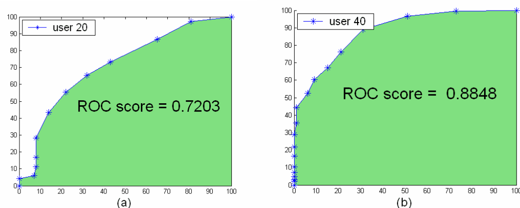

Figure 5: Example for ROC score using user 20 and user 40: the fraction of shaded area under ROC curve.

Figure 6 illustrates the performance of several one-class training algorithms as measured by ROC scores. The figure includes results for all 50 users. From Figure 6, we can see that one-class SVM using word-count features is the worst among the four algorithms. At the high ROC score region, with a ROC score higher than 0.8 (which is what we prefer) one-class SVM using binary features performs best among all. There is no big difference between Naïve Byaes using the multinomial model or the multi-variate Bernoulli model.

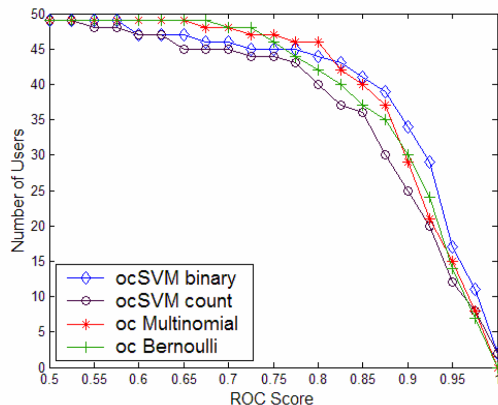

Figure 6: Comparison of four masquerade detection methods. The graph plots the total number of users for which a given method exceeds a ROC score threshold.

For the masquerade problem, we are more interested in the region of the ROC curve with a low false positive rate; otherwise, the "annoyance level" of false alarms would render the detector useless in practical use. Therefore, we restrict the ROC scores to the curves with false positive lower than P, which is called the ROC-P score. For example, if we want to restrict the false positives to be lower than 5% of all command blocks, we can compute ROC-5. Similar to the general ROC score, the ROC-P score is the fraction of the area under the ROC curve where the false positive rate is lower than P%. Figure 7, displays an example of ROC-10, based on the ROC-curves of users 20 and 40. Only part of the ROC curve is drawn here to highlight the plots.

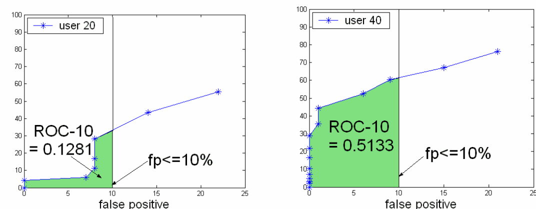

Figure 7: ROC-10 for user 20 and user 40, which is the fraction of the shaded area left to fp<=10% threshold line.

Since we can see that one-class SVM using the binary feature is generally better than one-class SVM using the word count feature, as depicted in Figure 6; here we only compare the one-class SVM using the binary representation with the multinomial model Naïve Bayes and Bernoulli model Naïve Bayes in the following ROC-P comparison. Figures 8 plots the comparison for ROC-5 and ROC-1, which means false positives are below 5% and 1%, respectively. From these two plots, we can

determine that one-class SVM using the binary feature is almost always better than the other two one-class Naïve Bayes methods.

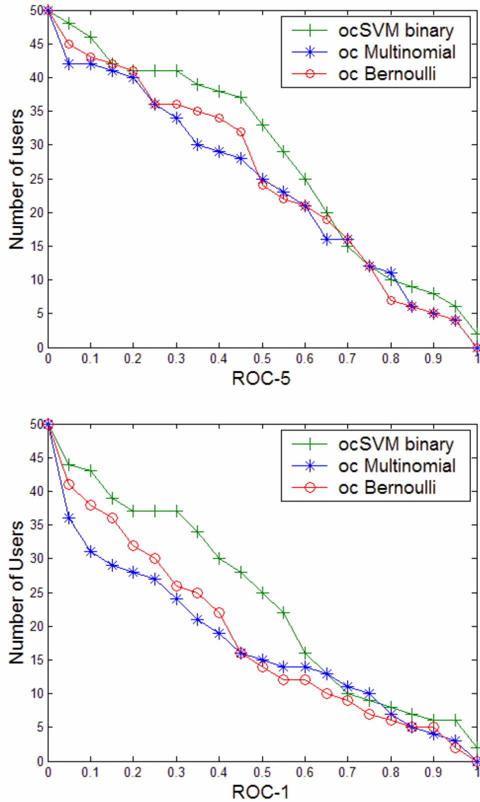

Figure 8: Comparison of ocSVM-binary with one-class Multinomial NB and one-class Bernoulli NB, restricted to 5% and 1% false positive rate, respectively. The graph plots the total number of users for which a given method exceeds an ROC-5, ROC-1 score threshold.

To compare the performance of different algorithms on an individual user basis, we compare the ROC-P score user by user. Figure 9 shows a user-by-user comparison of one-class SVM using the binary feature representation and one-class Naïve Bayes using the multinomial model, when the false positive rate is lower than 1%. Again we can see, for most of the 50 users, one-class SVM with binary features is better than one-class Naïve Bayes using the multinomial model. However, there are still some users whose data exhibit better performance using the one-class Naïve Bayes. This suggests that we can choose the best algorithm to use for an individual user to improve the whole system's performance.

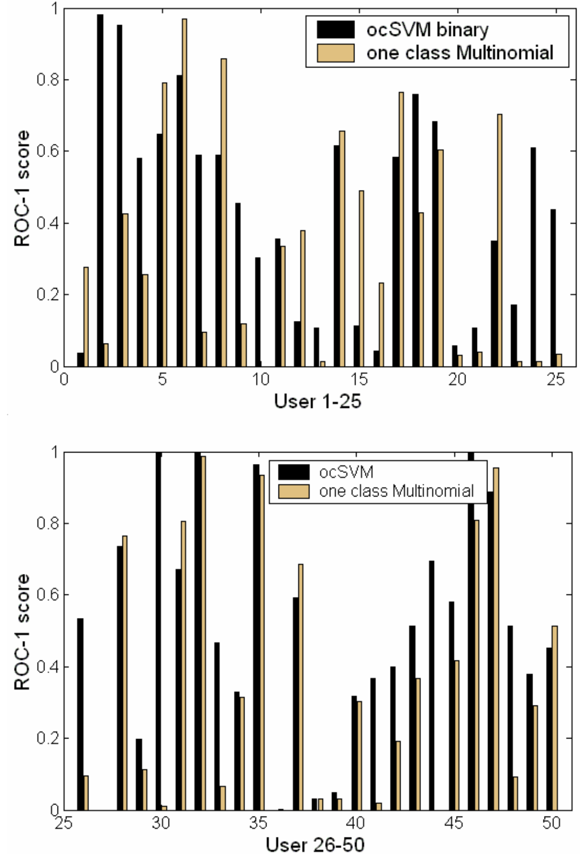

Figure 9: User-by-user comparison of ocSVM-binary and one-class Naïve Bayes using the multinomial model, restricted to 1% false positive.

5. Discussion

From our work we can see that one-class SVM using binary features performs better than one-class Naïve Bayes and one-class SVM using word count features.

Even so, masquerade detection is a very hard problem, and all three algorithms did not achieve very high accuracy with near to zero false positive rates for every user. This is partly caused by the inherent nature of the data available and the difficulty of this problem. We would like to reapply these methods using a richer set of data as described by Maxion [10], incorporating command arguments. We also believe that temporal data associated with each user's sequential commands will provide considerable value as well to improve performance.

Another problem to consider for the practical utility of these approaches is resiliency to direct attack; i.e. how could we protect the models that were computed from, for example, a mimicry attack by the masquerader?

In the experiments performed, we did not evaluate feature selection. We tested one-class SVM using 100,

200, and 300 of the most frequently used UNIX commands. Each of the results is worse than had we used all of the available UNIX commands, whose total number is around 870. We also conjectured that 2-gram features (adjacent pairs of commands) would perform better than individual commands (1-grams) as a feature. However, we found that the results were worse when we used all of the 2-grams. In further work, we would evaluate some feature selection methods to improve performance. For example, we believe a selection of some features using both 1-gram and 2-grams may improve the quality of the user profiles, and thus the accuracy of the detector.

A system to detect masqueraders as described in this paper should not be viewed as a single detector, but rather as evidence to be correlated with other sensors and other detectors. Thus, although the performance of the detectors described herein and in prior work seemingly are not accurate enough, when one wishes to limit false positives, it may be wise to relax the threshold to generate higher true positive rates. If the output of the detector were combined with other evidence (for example, file system access anomaly detection, or other sensors), it may be possible to raise substantially the bar in protecting hosts from malicious abuse.

6. Conclusion

In this paper, to solve the masquerade detection problem, we use one-class training algorithms which only train on a user's clean data. It has been demonstrated that one-class training algorithms can achieve similar performance as multiple class methods, but require much less effort in data collection and centralized management. Besides masquerade detection, we believe one-class training is also good for some other intrusion detection problems where sample intrusion data are hard to get or too variable to cluster.

We also give a detailed comparison of the performance of different one-class algorithms as applied to multiple users. The results show that for most users one-class SVM using the binary feature representation is better than one-class Naïve Bayes and one-class SVM using the word count representation, especially when we want to restrict the false positive rate to a relatively low level.

In our future work, we plan to include command arguments, not only truncated commands, as features to improve the accuracy of masquerade detection. As the number of features increase, we also plan to do feature selection to find the most informative features and to discard those features that have no value for the target task.

Acknowledgments

This work was partially supported by DARPA contract No. F30602-02-2-0209. We also thank Prof. Tony Jebara for helpful suggestions and valuable comments.

Reference:

[1] Chih-Chung Chang and Chih-Jen Lin, "LIBSVM: a library for support vector machines", 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[2] Eleazar Eskin, Wenke Lee and Salvatore J. Stolfo, "Modeling System Calls for Intrusion Detection with Dynamic Window Sizes", *Proceedings of DISCEX II*, June, 2001.

[3] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff, "A sense of self for UNIX processes", In *Proceedings of IEEE Symposium on Security and Privacy*, 1996.

[4] Anup K. Ghosh and Aaron Schwartzbard, "A study in using neural networks for anomaly and misuse detection", In *Proceedings of USENIX Security Symposium* 1999

[5] M. Gribskov and N. L. Robinson, "Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching", *Computers and Chemistry*, 20(1):25–33, 1996.

[6] Thorsten Joachims, "Text categorization with support vector machines: Learning with many relevant features", In *Proc. of the European Conference on Machine Learning (ECML)*, pp. 137-142, 1998.

[7] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection", In *Proceedings of USENIX Security Symposium* 1998

[8] T. Lunt, A.Tamaru, F. Gilham, R. Jagannathan, C. Jalai, H.S. Javitz, A. Valdes, and P.G. Neumann, "A Real-Time Intrusion Detection Expert System," *SRI CSL Tecnical Report*, SRI-CSL-90-05, June 1990.

[9] Maxion, Roy A. and Townsend, Tahlia N, "Masquerade Detection Using Truncated Command Lines", *International Conference on Dependable Systems and Networks (DSN-02)*, pp. 219-228, Washington, D.C. 23-26 June 2002.

[10] Maxion, Roy A. "Masquerade Detection Using Enriched Command Lines", In *International Conference on Dependable Systems & Networks (DSN-03)*, pp. 5-14, San Francisco, California, 22-25 June 2003. IEEE Computer Society Press, Los Alamitos, California, 2003.

[11] A. McCallurn, K. Nigam, "A Comparison of Event Models for Naive Bayes Text Classification", *AAAI-98 Workshop on Learning for Text Categorization*, 1998

[12] T. M. Mitchell, Bayesian Learning, Chapter 6 in *Machine Learning*, pp. 154-200. McGraw-Hill, 1997.

[13] B. Scholkopf, J.C. Platt, J. Shawe-Taylor, A.J. Smola, and R.C. Williamson, "Estimating the support of a high-dimensional distribution". *Technique report, Microsoft Research*, MSR-TR-99-87, 1999.

[14] M. Schonlau, W. DuMouchel, W. -H. Ju, A. F. Karr, M. Theus, and Y. Vardi, “Computer intrusion: Detecting masquerades”, *Statistical Science*, 16(1):58-74, February 2001.

[15] Matthew G. Schultz, Eleazar Eskin, and Salvatore J. Stolfo, “Malicious Email Filter - A UNIX Mail Filter that Detects Malicious Windows Executables”, *Proceedings of USENIX Annual Technical Conference - FREENIX Track,* Boston, MA: June 2001.

[16] S. Y. Sedelow, “The Computer in the Humanities and Fine Arts”, *ACM Computing Surveys* 2(2): 89-110 (1970)

[17] Salvatore J. Stolfo, Shlomo Hershkop, Ke Wang, Olivier Nimeskern, and Chia-Wei Hu, “Behavior Profiling of Email”, *1st NSF/NIJ Symposium on Intelligence & Security Informatics (ISI 2003),* June 2-3, 2003, Tucson, Arizona.

[18] O. De Vel, A. Anderson, M. Corney, and G. Mohay, “Mining Email Content for Author Identification Forensics”, *SIGMOD: Special Section on Data Mining for Intrusion Detection and Threat Analysis*, December 2001.

[19] Nong Ye, “A Markov Chain Model of Temporal Behavior for Anomaly Detection”, *Proceedings of the IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, 2000.

Learning Rules from System Call Arguments and Sequences for Anomaly Detection

Gaurav Tandon and Philip Chan
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901
{gtandon, pkc}@cs.fit.edu

Abstract

Many approaches have been suggested and various systems been modeled to detect intrusions from anomalous behavior of system calls as a result of an attack. Though these techniques have been shown to be quite effective, a key element seems to be missing – the inclusion and utilization of the system call arguments to create a richer, more valuable signature and to use this information to model the intrusion detection system more accurately. We put forth the idea of adopting a rule learning approach that mobilizes rules based upon system calls and models the system for normal traffic using system call arguments and other key attributes. We present variations of our techniques and compare the results with those from some of the well known techniques based upon system call sequences. The results show that system call argument information is crucial and assists to successfully detect U2R, R2L and Data attacks generating lesser false alarms.

1. Introduction

Motivation: The Internet has invariably been a medium for malicious purposes. Attacks on computers, be it some graduate students trying to hack systems to prove their mettle or intruders with more damaging intentions, is on a steady rise. Moreover, novel attacks and hacking schemes are developed all the time, making it hard for systems to be made immune to all these vulnerabilities. It has thus become imperative that these be checked early to minimize losses.

Two different lines of approach have been adopted to detect intrusions. The first technique, *misuse (signature) detection*, is similar to pattern matching -- systems are modeled upon known attack patterns and the test data is checked for the occurrence of these patterns. These systems have a high degree of accuracy but fail to detect new attacks. The other method, *anomaly detection*, models normal behavior and significant deviations from this behavior are considered anomalous. The primary advantage of this approach is that it can detect novel attacks, the drawback being that it can generate a lot of false alarms. This is attributed to the fact that not all anomalies are necessarily attacks and will thus result in false positives.

Intrusion Detection Systems (IDSs) can also be categorized as network-based and host-based. In the former, header fields of the various network protocols are used to detect intrusions. For example, the IP header fields - source IP address, destination IP address, source port number, destination port number and others can be used to check for malicious intent. In the latter approach (a host-based IDS), the focus shifts to the operating system level. System call data is extracted from audit logs like the Solaris Basic Security Module (BSM) [16] and their behavior is studied to detect attacks.

Most of the present techniques for host-based anomaly detection systems revolve around sequences of system calls. These techniques are based upon the observation that an illegitimate activity results in an abnormal (novel) sequence of system calls.

Problem: The efficacy of such systems might be improved upon if more information is utilized. For system calls the most intuitive option lies in the system call arguments. Some other attributes related with system calls are the path for the object, the return value and the error status. Does adding these attributes assist in modeling a host-based anomaly detection system better? How do such systems fare (in terms of detections, false alarms, space and time

requirements) as compared to the systems based only upon the sequence of system call information? These are some of the key issues we seek to explore in this paper.

Approach: We extract system calls, their arguments, path, return value and error status from the Solaris BSM audit logs [16]. We then propose a host-based anomaly-detection system using system calls and other aforementioned key attributes by using variants of *LERAD* (Learning Rules for Anomaly Detection) [14], which is a conditional rule-learning algorithm. We aim at forming rules for our anomaly detection system based upon the system calls and their attributes. We suggest that including these attributes to the system calls will result in learning more information, thereby enabling us to model our systems better and detecting more attacks. We propose three models – the first one modeling system call sequences using *LERAD*, the second modeling system call arguments and other attributes, and the third approach being a combination of the two. We juxtapose these techniques and also compare them with some of the previous well-known sequence-based techniques, namely *tide*, *stide*, *t-stide* [20].

Contributions:

- We proposed the use of system call argument information to enrich the representation of program behavior in anomaly detection.
- We proposed modifications to *LERAD* to learn rules that allow one of the attributes to be designated as a pivotal attribute (system call in our case -- explanation in *Section 3.2.2*) on which the rules are based.
- As compared to *tide*, *stide* and *t-stide*, three well known sequence-based techniques (more details in *Section 2*), our argument-based systems are able to detect more attacks at lower false alarm rates.
- Our method that uses both sequence and argument information generally detected the most attacks with different false alarm rates.

Organization: *Section 2* describes the related work in the field of anomaly detection. In *Section 3*, we discuss the approach that we adopt for prepare the data set for our anomaly detection models. We give a brief explanation of *LERAD* on which our models are based. Then we describe the three variants of *LERAD* that are used to investigate different issues. *Section 4* gives a brief description of evaluation data, procedure and criteria. Then we analyze the results obtained from the experiments we performed. In *Section 5*, we conclude and put forth some views for future endeavors.

2. Related Work

Forrest et al. [2] proposed an approach for host based anomaly detection called time-delay embedding (*tide*), wherein traces of normal application executions were noted. A sliding look-ahead window of a fixed length was used to record correlations between pairs of system calls. These correlations were stored in a database of normal patterns, which was then used to monitor sequences during the testing phase. Anomalies were accumulated over the entire sequence and an alarm was raised if the anomaly count exceeded the threshold. *tide* forms correlations between pairs of system calls within a certain preset window size. Some of the issues involved in their approach were: using a small window does not help to form correlations over a long period of time. Similar sequences with minor variations could still be flagged as anomalous.

Later work by Warrender et al [20] extended this technique in sequence time-delay embedding (*stide*), which memorized all contiguous sequences of predetermined, fixed lengths during training. An anomaly count was defined as the number of mismatches in a temporally local region. A threshold was set for the anomaly score above which a sequence is flagged anomalous, indicating a possible attack. *stide* memorizes all fixed length sequences from the training data, irrespective of the number of instances found in the dataset. An extension, called sequence time-delay embedding with (frequency) threshold (*t-stide*), was similar to *stide* with the exception that the frequencies of these fixed length sequences were also taken into account. Rare sequences were ignored from the normal sequence database in this approach. When encountered during the testing phase, they were also counted as mismatches and aggregated to the locality frame counts (anomaly counts). All these techniques modeled normal behavior by using fixed length patterns of training sequences. But there was no rationale in fixing the length to a predetermined constant value.

Wespi et al. [21], [22] proposed a scheme to generate variable length patterns by using Teiresias [17], a pattern-discovery algorithm in biological sequences. These techniques improved upon the fixed length pattern methods cited above. Some extensions to (fixed and variable length) sequence-based methods were also proposed in [6], [7] and [8]. Though all the above mentioned approaches use system call sequences, none of them make use of the system call arguments. Given some knowledge about the system being used, attackers can devise some methodologies to evade such intrusion detection systems. Wagner and Soto [19] made such an

attempt to model a malicious sequence by adding "no-ops" (system calls having no effect) to compromise an IDS based upon the sequence of system calls. This brings to surface yet another shortcoming of sequence-based methods. Such attacks would fail if the system call arguments are also taken into consideration.

Sekar and others [18] proposed a method to build a compact finite state automaton (FSA) in an efficient way to detect intrusive activities. But no frequency information is stored in the FSA. Again, there lies the inherent drawback that the system call arguments are not considered. In [3], Feng et al proposed a method that dynamically extracts return address information from the call stack and program counter information is recorded at each system call. This technique performs equally well as compared to the deterministic FSA approach in terms of detections, convergence and false positives.

Artificial neural networks (ANNs) have been employed for both anomaly and misuse (signature) detection. Ghosh and Schwartzbad [4] expressed the idea of a process-based intrusion detection system that can generalize from previously observed behavior to recognize future unseen behavior. But their system ignores isolated anomalies.

Machine learning approaches have also been used to model intrusion detection systems. Lee at el. [11] verified the feasibility of rule-learning approaches by using an algorithm called *RIPPER* [1]. Mahoney and Chan [14] introduced a machine-learning algorithm called *LERAD* (Learning Rules for Anomaly Detection) to detect network intrusions. This technique extended the network traffic model to include a larger number of attributes. They also introduced and used the concept of a non-stationary model in [13], [14] and [15], in which the probability of an event depends upon its most recent occurrence and not on the frequency. *LERAD* is a conditional rule-learning algorithm that selects good rules from a vast rule space. This paper uses variants of *LERAD* for a host-based anomaly detection system.

3. Approach

Rule learning techniques have been shown that they can be successfully adapted to model systems for intrusion detection [14]. Since our goal is to detect host-based intrusions and we are dealing with BSM audit data, system calls are instrumental in our system. We thus extend upon the machine learning approach and incorporate the system calls with its arguments to generate a richer set of rules and measure the performance on the basis of number of detections and the false alarm rate. We study and evaluate three different variations of modeling a system using *LERAD*: sequence of system calls, system calls and their arguments, and a fusion of the previous two methodologies. We compare and contrast the results from these three models of our approach with *tide*, *stide* and *t-stide*.

3.1. Learning Rules for Anomaly Detection (*LERAD*)

LERAD is an efficient conditional rule-learning algorithm that picks up attributes in a random fashion. *LERAD* is briefly described here. More details can be obtained from [14]. *LERAD* learns rules of the form:

$$A = a, B = b, \ldots \Rightarrow X \in \{x_1, x_2, \ldots.\} \quad (1)$$

where *A*, *B*, and *X* are attributes and *a*, *b*, *x1*, *x2* are values to the corresponding attributes. The learned rules represent the patterns present in the training data that consist of normal behavior. The set {*x1*, *x2*, …} in the consequent constitutes all unique values of *X* when the antecedent occurs in the training data. (These rules are different from typical classification rules or association rules.)

Records that match the antecedent but not the consequent of a rule are considered anomalous. The degree of anomaly is based on a probabilistic model. For each rule, from the training data, the probability, *p*, of observing a value not in the consequent is estimated by:

$$p = \Pr(X \notin \{x_1, x_2 \ldots\} \mid A = a, B = b, \ldots) = r / n \quad (2)$$

where '*r*' is the cardinality of the set, $\{x_1, x_2, \ldots\}$, in the consequent and '*n*' is the number of records that satisfy the antecedent. This probability estimation of novel (zero frequency) events is due to Witten and Bell [23]. Since *p* estimates the probability of a novel event, the larger *p* is, the less anomalous a novel event is. Hence, during detection, when a novel event is observed, the degree of anomaly, or Anomaly Score, is estimated by:

$$AnomalyScore = 1 / p = n / r \quad (3)$$

The rule generation phase of *LERAD* comprises of three main steps:

(i) Candidate rules are generated from patterns observed in randomly selected pairs of training examples: Training samples are picked up at random and then an initial set of rules is generated based upon common attributes between the samples. The conditional rules formed are of the type depicted in *Equation (1)* above.

(ii) The rule set is minimized by removing rules that do not cover/describe additional training examples: Redundant rules are discarded and a minimal set of rules is generated.

(iii) A subset of the training set is chosen as a validation set on which no training is performed: Rules learnt so far are used to test the data in this validation set. Rules are removed if they cause a false alarm in the validation set. This is due to the fact that the validation data set comprises of clean data (no attacks) and any anomaly implies a false alarm.

The rule generation methodology of *LERAD* is described next using *Table 1*.

Table 1: *LERAD* rule generation example: S1 – S6 are training samples with attributes A, B, C and D.

Training Sample	A	B	C	D
S1	1	2	3	4
S2	1	2	3	5
S3	6	7	8	4
S4	1	0	9	5
S5	1	2	3	4
S6	6	3	8	5

Step (i) Samples, say S1 and S2, are picked at random to create an initial rule set. Rules are generated by selecting matching attributes in a random order. In this example, the S1 and S2 have the matching attributes A, B and C. Selecting them in the order B, C and A, we get the following 3 rules:

Rule1: * $\Rightarrow$ B $\in$ {2}

Rule 2: C=3 $\Rightarrow$ B $\in$ {2}

Rule 3: A=1, C=3 $\Rightarrow$ B $\in$ {2}

A rule so generated implies that the attribute in the consequent can have a value from a set of values only if the conditions in the antecedent are satisfied. It may so happen that there is a consequent but no antecedent in a rule formed by *LERAD*. This means that an attribute can take any value from its set of values without the need to satisfy any other condition. Such a situation is presented in Rule 1 where the antecedent is represented by a wildcard character *.

Step (ii) Coverage test is applied to a subset of the training set (say S1-S3) and rules are modified as follows:

Rule1: * $\Rightarrow$ B $\in$ {2, 7}

Rule 2: C=3 $\Rightarrow$ B $\in$ {2}

Rule 3: A=1, C=3 $\Rightarrow$ B $\in$ {2}

Once we have the extended rule set, the probability p -- described in *Equation (2)* above -- is associated with every rule. The rules are then sorted in increasing order of the probability p:

Rule 2: C=3 $\Rightarrow$ B $\notin$ {2} [p = 1/2]

Rule 3: A=1, C=3 $\Rightarrow$ B $\notin$ {2} [p = 1/2]

Rule 1: * $\Rightarrow$ B $\notin$ {2, 7} [p = 2/3]

When the probabilities are equal, the rule with lesser number of conditions in the antecedent is given higher priority (Rule 2 is higher in priority than Rule 3 in our example). Next, we desire a minimal set of rules. This is achieved by removing those rules that do not give any new information. In our example, Rule 2 is satisfied by samples 1 and 2. Rule 3 does not add any new value to the attribute B and is thus deemed as redundant and is removed from the rule set. The last rule (Rule 1) covers sample 3 as well and is kept in the rule set.

Extending the two rules to the entire training (minus validation) set (samples S1-S5 in our example), we get

Rule 2: C=3 $\Rightarrow$ B $\notin$ {2} [p = 1/3]

Rule 1: * $\Rightarrow$ B $\notin$ {2, 7, 0} [p = 3/5]

Step (iii): The last step comprises of testing the above set of rules on the validation set, which is a subset of the training data for which rules have not been generated. Any rule which produces anomaly in the validation set is removed. In our example, sample S6 forms the validation set. Rule 1 is violated since attribute B has a novel value 3 in this sample. Thus, we are left with the following rule:

C=3 $\Rightarrow$ B $\notin$ {2} [p = 1/3]

A non-stationary model is assumed for *LERAD* – frequency is made irrelevant and only the last occurrence of an event is assumed important. Since novel events are bursty in conjunction with attacks, a '*t*' factor was introduced to capture the non-stationary characteristic, where '*t*' is the time interval since the last novel (anomalous) event. When a novel event occurred recently, or *t* is small, a novel event is more likely to occur at the present moment. Hence, the anomaly score is measured by *t*/*p*. Since a record can deviate from the consequent of more than one rule, the total anomaly score of a record is:

$$TotalAnomalyScore = \sum_i t_i / p_i = \sum_i t_i n_i / r_i \qquad (4)$$

where '*i*' is the index of a rule from which the record has deviated. The anomaly score is aggregated over all the rules to combine the effect from violation of multiple rules. The more the violations, more critical the anomaly is, and the higher the anomaly score should be. *LERAD* yields successful results for network-based anomaly detection systems. This paper extends the algorithm for host-based anomaly detection systems.

3.2. Variants of *LERAD*

Our goal is to create a system that can detect any anomaly across any application/program. We developed a taxonomy of the entire data set from the BSM audit log. We classified the data into various applications/programs and generated a model for each of them.

3.2.1. Sequence of system calls: *S-LERAD*

Using sequence of system calls is a very popular approach for anomaly detection. We performed experiments wherein we extracted system calls from the data. We used a window of fixed length 6 (as this is claimed to give best results in *stide* and *t-stide* [20]) and fed these sequences of six system call tokens as input to *LERAD*. We called this technique as *S-LERAD* since we are trying to capture system call sequences by using *LERAD*.

For input to *LERAD*, we thus have a set of following attributes: date and time when system call information logged, the last two bytes of the destination IP address used for identifying the hosts during the evaluation, a system call and the previous five system calls, thereby making it a sequence of 6 system calls. *LERAD* uses these attributes at random to generate rules as described in *Section 3.1*.

The purpose of performing this experiment was to explore whether *LERAD* would be able to capture the correlations among system calls in a sequence. Also, this experiment would assist us in comparing results by using the same algorithm for system call sequences as well as system call arguments. Since *stide* and *t-stide* report best results for sequences of length 6, we increased the maximum number of allowed attributes in the antecedent of the rules generated by *LERAD* from 3 to 5, keeping the consequent fixed at 1 attribute.

A sample rule learned in a particular run of *S-LERAD* is:

$$SC1 = close(), SC2 = mmap(), SC6 = open() \Rightarrow SC3 \in \{munmap()\}$$

n/*r* value = 455/1

This rule is analogous to encountering *close*() as the first system call (represented as *SC 1*), followed by *mmap*() and *munmap*(), and *open*() as the sixth system call (*SC 6*) in a window of size 6 sliding across the audit trail. Each rule is associated with an n/*r* value, as explained in *Section 3.1*. The number 455 in the numerator refers to the number of training instances that comply with the rule (*n* in Equation 3). The number 1 in the denominator implies that there exists just one distinct value of the consequent (munmap() in this case) when all the conditions in the premise hold true (*r* in Equation 3 of *Section 3.1*).

3.2.2. System call arguments and other key attributes: *A-LERAD*

We propose that argument and other key attribute information is integral to modeling a good host-based anomaly detection system. In this experiment, we extracted arguments, object path, return value and error status of system calls from the Solaris BSM audit log and examined the effects of learning rules based upon system calls along with these attributes.

We built models per application using *LERAD* with the modification that the rules were forced to have system call in the antecedent since it is the key attribute in a host based system. The generic version of *LERAD* could have been used to generate rules, but the motivation behind this is that ours is a host-based system and is centered upon system calls. We term the system call as a *pivotal attribute*

since our rules are based upon it. Thus, the system call will always be a condition in the antecedent of the rule.

This model is given the nomenclature *A-LERAD* since our motive here is to generate rules for various attributes given the system calls. Any value for the other arguments (given the system call) that was never encountered in the training period for a long time would raise an alarm. A sample rule is of the form:

$$SC = munmap() \Rightarrow Arg1 \in \{0x134, 0102, 0x211, 0x124\}$$

n/*r* value = 500/4

In the above rule, 500/4 refers to the *n*/*r* value for the rule (*Equation 3* in *Section 3.1*), that is, the number of training instances complying with the rule (500 in this case) divided by the cardinality of the set of allowed values in the consequent. This rule gives the 4 different values encountered for the first argument when the system call is munmap().

The maximum number of arguments has been chosen as 5 since most of system calls do not take more than 5 arguments. Considering more number of arguments results in more null values for the same and may cause formation of not-so-important rules thereby degrading the system performance. Thus only the high frequency arguments were selected from the data set. There may be several other approaches that can be adopted in this regard. Ours is just one intuitive approach.

3.2.3. Merging argument information and sequence of system calls: *M-LERAD*

The third set of experiments we conducted was to combine the techniques discussed in *Sections 3.3.1* and *3.3.2*. The first is a well acclaimed technique based upon sequence of system calls and is known to be an effective technique; the second one takes into consideration the attributes (arguments, path, return value and error status), whose efficacy we claim in this paper; so fusing the two to study the effects was an obvious choice. We call this technique as *M-LERAD* (short form for the merged system), as we desire to combine system call sequences and the related key attributes. Merging is accomplished by adding more attributes in each tuple before input to *LERAD*. Each tuple now comprises of the system call, arguments, object path, return value, error status and the previous five system calls. The *n*/*r* values obtained from the all rules violated are aggregated into an anomaly score, which is then used to generate an alarm based upon the threshold.

4. Experimental Evaluation

Our goal is to study if the rule-learning algorithm *LERAD* can be modified to determine as many attacks with least number of false alarms in a host-based anomaly detection system.

4.1. Evaluation Data and Procedures

We evaluated out techniques using the 1999 DARPA Intrusion Detection Evaluation Data Set [12]. The test bed involved a simulation of an air force base that has machines that are under frequent attack. These machines comprised of Linux, SunOS, Sun Solaris and Windows NT. Various intrusion detection systems were evaluated using this test bed, which comprised of three weeks of training data obtained from network sniffers, audit logs, nightly file system dumps and BSM logs from Solaris machine that trace system calls. Training was performed on week 3 data (around 2.1 million system calls) and testing on weeks 4 and 5 data (comprising over 7 million system calls) from the BSM audit log. A total of 51 attacks during weeks 4 and 5 were targeted at the Solaris machine, from which the BSM log was collected.

Data from the Basic Security Module (BSM) [16] audit log has to be preprocessed before it can be fed as input to *LERAD*. This was important from the point of view that we want to model process behavior for application. We divided the entire data set into various applications. For each application, we grouped the data on the basis of the process ID. For a given process id, all the data from the exec system call to the exit system call comprised the data for that particular process. Data for which we could not trace the start of the process was excluded from our experiments. The fork system call was dealt in a special way. A parent process spawns a child process with the fork system call, that is, a copy of the parent process is created. Unless fork is followed by exec, the child performs the same tasks as the parent process. Therefore, all the system calls for a child process are for the same application as the parent process until it encounters its own exec system call. In this way, we divided the data into applications, and further into processes belonging to the various applications/programs.

All the system calls (with their arguments) pertaining to a single process were thus differentiated from the set of

system calls (and arguments) for another process belonging to the same application. In a similar manner, sequences of system calls for various processes of different applications were differentiated from one another and were ready to be used for our rule-based learning models.

The parameters for *S-LERAD* were the 6 contiguous system calls; for *A-LERAD* they comprised of the system call, its return value and error status besides other arguments; and for *M-LERAD* it was a combination of the two techniques. For *tide*, the parameters were all the pairs of system calls within a window of fixed size 6; *stide* comprised all contiguous sequences of length 6, and *t-stide* added frequency information to the same. These sequence-based methodologies have been discussed in *Section 2*. In all models, alarms are accumulated for the applications and then evaluated for true detections and false positives.

4.2. Evaluation Criteria

The performance metrics used in this 1999 DARPA evaluation were the attack detection rate and the number of false alarms generated. We have adopted the same for the purpose of evaluating our system. As per the evaluation criteria, a system is considered to have successfully detected an attack if it generates an alarm within 60 seconds of the occurrence of the attack. We also follow the same criterion for evaluating our schemes.

The attacks in the 1999 DARPA evaluation are classified as probes, DoS, R2L, U2R and Data. These are based upon the classification by Kendell[10]. The taxonomy is as follows:

(i) Probes or scan attacks are attempts by hackers to collect information prior to an attack. Examples include illegalsniffer, ipsweep, mscan, portscan amongst others.

(ii) DoS (Denial of Service) attacks are the ones in which a host or a network service is disrupted. For example, arppoison, selfping, dosnuke and crashiis are all DoS attacks.

(iii) R2L (Remote to Local) – In these attacks, an unauthorized user gains access to a system. Examples of R2L attacks are guest, dict, ftpwrite, ppmacro, sshtrojan and framespoof.

(iv) U2R (User to Root) / Data attacks are those in which a local user is able to execute non-privileged commands, which only a super user can execute. Examples are eject, fdformat, ffbconfig, perl, ps and xterm.

Some attacks are combinations, such as a U2R attack that enables the attacker to steal secret data and are therefore categorized as Data-U2R attacks. Similarly, there are also Data-R2L attacks.

Lippmann et al [12] lists poorly detected attacks as the ones even half of whose instances were not detected by the any of the IDSs in the 1999 DARPA Evaluation. For the Solaris host, these were all DoS attacks. Host-based systems that use Solaris based audit data are more inclined to detect R2L, U2R and Data attacks than network-based intrusion detection systems.

As we are using more information (in the form of system call arguments) for our models, another important criterion is the space and the CPU time requirements, which is discussed in *Section 4.4*.

4.3. Results and Analysis of Detection Rates

We built training models for various applications. We reiterate our motivation for forcing rules based on system calls, as they are the pivotal attributes for our model. We trained our system on week 3 of the DARPA data and tested on weeks 4 and 5. Putative detections were considered as true positives if they occurred within 60 seconds of the attack segment for the correct destination (victim) IP address, which in our case was a single Solaris host.

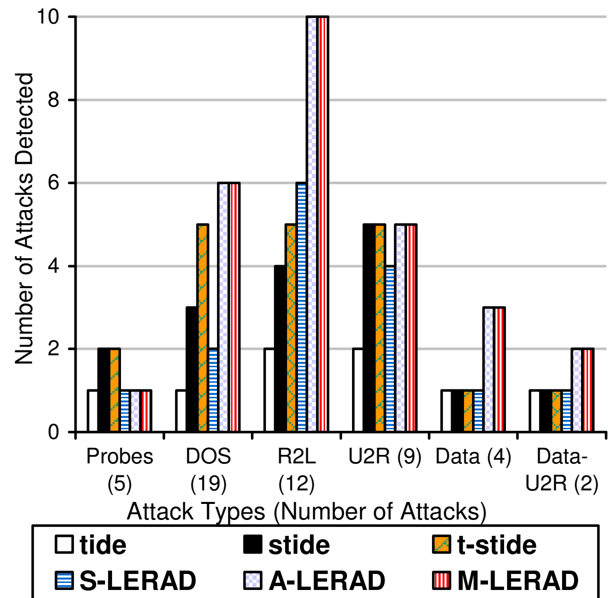

Figure 1: Number of detections with 10 false alarms per day for different attack categories.

Figure 1 plots the result based on a leeway of 10 false alarms per day of testing week, making a total of 100 false alarms for the two weeks of testing. The best technique using sequence-only information was *t-stide*, detecting 2 probes, 5 DoS, 5 R2L, 5 U2R, 1 Data and 1 Data-U2R attacks. Both *stide* and *t-stide* were able to find more probes than our argument-based technique, but our claim lies in finding more R2L, U2R and data attacks. *A-LERAD* was able to detect 10 R2L, 5 U2R, 3 Data, and both the Data-U2R attacks, apart from a probe and 6 DOS attacks. On the other hand, *S-LERAD* was not able to detect many of these attacks. The better performance of *A-LERAD* over *S-LERAD* can be attributed to the inclusion of argument information in the former model. The graph depicts no improvement by adding sequence information to argument information since *A-LERAD* and *M-LERAD* had exactly the same detections for the given false alarm rate. This also suggests that argument information is sufficient for detecting anomalies and there is no need for adding sequence information to *A-LERAD*.

Our techniques were also able to detect some poorly detected attacks quoted in [12]. For the Solaris host, these were *DoS* attacks, some of which we were able to capture accurately. There was only one instance of *tcpreset*, which our system detected successfully. We were also able to detect 2 instances of *warezclient*, both of which were not detected by the best system for that attack in the 1999 DARPA Evaluation.

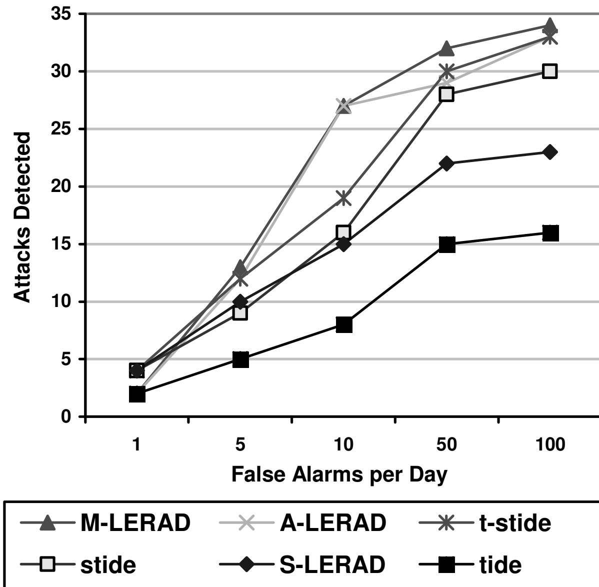

Figure 2: Detections for the 6 techniques at variable false alarms rates (for a total of 51 attacks in 2 weeks of data).

Figure 2 plots the total attacks detected by various techniques at 1, 5, 10, 50 and 100 false alarms per day respectively. *t-stide* maintained to be the best sequence-based technique, followed by *stide*, *S-LERAD* and *tide*. *A-LERAD* fared better than *S-LERAD* and the other sequence-based techniques, suggesting that argument information is more useful than sequence information. The *M-LERAD* curve is usually at or above the other curves, indicating that *M-LERAD* usually detects more attacks at various false alarm rates than the remaining five methods.

It can also be seen that the *A-LERAD* curve closely follows the curve for *M-LERAD*. This may imply that the sequence information is redundant; it is not adding substantial information to what we already have from the arguments. In other words, the attacks detected by using sequence information were also detected by using argument information, thereby giving similar results for *M-LERAD* and *A-LERAD*. A key point to observe is that even though the number of detections is almost same for the two techniques, *M-LERAD* has a faster convergence than *A-LERAD*.

We also observe that the significant difference in the performance of *M-LERAD* and *t-stide* is only at 10 false alarms per day. The reason for this is that the ROC curve is plotted on the basis of 5 discrete points only. For lower false alarm rates (1 and 5 per day), similar number of attacks was easily detected by both techniques. This can be attributed to the fact that these attacks contained both sequence and argument based anomalies. But as we increase the acceptable false alarm rate, we see that sequence anomalies do not necessarily correspond to an attack, whereas the argument anomalies are a good representation of an occurrence of an attack. By relaxing the allowed false alarm rate further (50 or 100 false alarms per day), we certainly expect to get more detections. We notice from the figure that we do get similar performance for *M-LERAD* and *t-stide* in such cases, but it is accompanied with a huge cost in terms of the number of false alarms, which is unacceptable for real-time systems.

By performing the comparison of the various techniques, we were also able to determine the effectiveness of the anomaly scoring function. Amongst the most effective techniques, *A-LERAD* and *M-LERAD* use a time based probabilistic estimation and *t-stide* incorporates frequency information. The way these techniques score anomalies is also a crucial factor in such anomaly detection systems.

One of the issues we investigated was whether to force *LERAD* to form rules based upon a system call as a condition in the antecedent or let it formulate rules without pivoting on a system call (as discussed in *Section 3.2.2*). We performed some experiments using *A-LERAD* with and without the enforcement of system call as a condition in the antecedent. Based upon the empirical evidence, we concluded that this enforcement resulted in the detection of at least as many attacks as in the relaxed case with the generation of fewer false alarms.

4.4. Results and Analysis of the CPU Time and Space Requirements

Compared to sequence-based methods, our techniques extract and utilize more information (system call arguments and other attributes), making it imperative to study the feasibility of our techniques in terms of space and time requirements.

During training, for *t-stide*, all contiguous system call sequences of length 6 along with their respective frequencies are stored in a database. For *M-LERAD*, system call sequences and other attributes are stored. In both the cases, space complexity is of the order of O(*n*), where '*n*' is the total number of system calls, though the *M-LERAD* requirement is more by a constant factor *k* since it stores additional argument information. During detection, *M-LERAD* uses only the learned set of rules (in the range 14-35 at an average of 25.1 rules per application in our experiments). *t-stide*, on the other hand, still requires the entire database of fixed length sequences during testing, which incur larger space overhead during detection. We conducted experiments on the *tcsh* application data. The entire week 3 training data set comprises of over 2 million system calls and the test data (weeks 4 and 5 combined) has over 7 million system calls. For *tcsh,* system calls alongwith their arguments form a 33 MB input file for *M-LERAD*. The rules formed by *M-LERAD* require less than 1.5 KB space, apart from a mapping table to map strings and integers. For the same application, the memory requirements for storing a system call sequence database for *t-stide* were over 5 KB plus a mapping table between strings and integers. The results suggest that *M-LERAD* has better memory requirements during the detection phase. We reiterate that the training can be done offline. Once the rules are generated, *M-LERAD* can be used to do online testing with lower memory requirements.

The time overhead incurred by *M-LERAD* and *t-stide* in our experiments is given in *Table 2*. The CPU times have been obtained on a Sun Ultra 5 workstation with 256 MB RAM and 400 MHz processor speed. We can infer from the results that *M-LERAD* is slower than *t-stide*. During training, *t-stide* is a much simpler algorithm and processes less data than *M-LERAD* for building a model and hence *t-stide* has a much shorter training time. During detection, *t-stide* just needs to check if a sequence in the database, which can be efficiently implemented with a hash table. On the other hand, *M-LERAD* needs to check if a record matches any of the learned rules. Also, *M-LERAD* has to process additional argument information. Run-time performance of *M-LERAD* can be improved with more efficient rule matching algorithm. Also, *t-stide* will incur significantly larger time overhead when the stored sequences exceed the memory capacity and disk accesses become unavoidable – *M-LERAD* does not encounter this problem as easily as *t-stide* since it will still use a small set of rules. More importantly, *M-LERAD*'s time overhead is about tens of seconds for days of data, which is reasonable for practical purposes.

Table 2: Comparison of CPU times during training and testing phases for *t-stide* and *M-LERAD* for top 8 applications in terms of total number of system calls (not necessarily in that order).

Application	Training Time (seconds) [on 1 week of data]		Testing Time (seconds) [on 2 weeks of data]	
	t-stide	M-LERAD	t-stide	M-LERAD
ftpd	0.19	0.99	0.19	0.96
telnetd	0.96	7.87	1.05	9.79
ufsdump	6.76	33.33	0.42	1.78
tcsh	6.32	32.85	5.91	37.58
login	2.41	16.75	2.45	19.86
sendmail	2.73	15.09	3.23	21.63
quota	0.20	3.48	0.20	3.79
sh	0.21	3.25	0.40	5.63

5. Concluding Remarks

Even though system call sequences are beneficial in modeling normal process behavior, they are not omniscient. In this paper, we portrayed the efficacy of incorporating system call argument information and used a rule-learning algorithm to model a host-based anomaly detection system. Our argument-based model, *A-LERAD*, detected more attacks at lower false alarm rates than the sequence-based techniques on the 1999 DARPA evaluation dataset. Combining the two lines of approach

(argument and sequence information) resulted in creating a richer and, more importantly, more accurate model for anomaly detection, as illustrated by the empirical results of *M-LERAD*. Though our techniques incur higher time overhead due to the complexity of our techniques as well as more information to be processed, they build more succinct models that incur much less space overhead--our techniques aim to generalize from the training data, rather than simply memorize the data.

Our techniques can be easily extended to monitor audit trails in continuum. Since we model each application separately, some degree of parallelism can also be achieved to test process sequences as they are being logged. *S-LERAD* fares poorly as compared to *stide* and *t-stide*. We are currently trying to analyze and rectify its shortcomings, which might have an impact on the performance of *M-LERAD* as well. Also, we were able to see from our experiments that the time based probabilistic estimation of anomaly score as proposed in *LERAD* and the frequency component of *t-stide* are effective ways to flag data as anomalous. These two functions can be combined to give a more appropriate anomaly score. It would be interesting to see how this would affect the results. We might perform experiments and publish results for the same in the near future.

Acknowledgements

This work is partially funded by DARPA (F30602-00-1-0603). We thank the LLR members for their help on ideas and the anonymous reviewers for their comments.

References

[1] Cohen W. Fast Effective Rule Induction, in Machine Learning. *Proc. ICML* 1995.

[2] Forrest S., Hofmeyr S., Somayaji A., and Longstaff T. A Sense of Self for UNIX Processes. *1996 IEEE Symposium on Research in Security and Privacy*.

[3] Feng H., Kolesnikov O., Fogla P., Lee W. and Gong W. Anomaly Detection Using Call Stack Information. *IEEE Symposium on Security and Privacy,* 2003.

[4] Ghosh A., and Schwartzbad A. A Study in Using Neural Networks for Anomaly and Misuse Detection. *1999 USENIX Security Symposium*.

[5] Hangal S. and Lam M.S. Tracking Down Software Bugs Using Automatic Anomaly Detection. *International Conference on Software Engineering,* 2002.

[6] Helman P. and Bhangoo J. A statistically based system for prioritizing information exploration under uncertainty. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 1997.

[7] Hofmeyr S. A., Forrest S., and Somayaji A. Intrusion detection using sequences of system calls. *Journal of Computer Securit,y* 1998.

[8] Jiang N., Hua K., and Sheu S. Considering Both Intra-pattern and Inter-pattern Anomalies in Intrusion Detection. *Proc. Intl. Conf. Data Mining*, 2002.

[9] Jones A., Li S. Temporal Signatures for Intrusion Detection. *Computer Security Applications Conference*, 2001.

[10] Kendell K. A *Database of Computer Attacks for the Evaluation of Intrusion Detection Systems*. Masters Thesis, MIT 1999.

[11] Lee W., Stolfo S., and Chan P. Learning Patterns from UNIX Process Execution Traces for Intrusion Detection. *AAAI'97 workshop on AI methods in Fraud and risk management*.

[12] Lippmann R., Haines J., Fried D., Korba J., and Das K. The 1999 DARPA Off-Line Intrusion Detection Evaluation. *Computer Networks*, 2000.

[13] Mahoney M. and Chan P. Packet Header Anomaly Detection for Identifying Hostile Network Traffic, *Florida Tech. Technical Report CS-2001-04*.

[14] Mahoney M., and Chan P. Learning Rules for Anomaly Detection of Hostile Network Traffic, *Proc. of the Third IEEE International Conference on Data Mining, 2003 (to appear)*.

[15] Mahoney M. and Chan P. Learning non-stationary models of normal network traffic for detecting novel attacks. *Proc. Intl. Conf. Knowledge Discovery and Data Mining,* P 376-385, 2002.

[16] Osser W., and Noordergraaf A. *Auditing in the Solaris™ 8 Operating Environment*. Sun Blueprints™ Online - February 2001.

[17] Rigoutsos Isidore and Floratos Aris. Combinatorial pattern discovery in biological sequences. *Bioinformatics*, 1998.

[18] Sekar R., Bendre M., Dhurjati D., Bollineni P. A Fast Automaton-based Method for Detecting Anomalous Program Behaviors. *IEEE Symposium on Security and Privacy*, 2001.

[19] Wagner D., Soto P. Mimicry Attacks on Host-Based Intrusion Detection Systems. *ACM Conference on Computer and Communications Security*, 2002.

[20] Warrender C., Forrest S., Pearlmutter B. Detecting Intrusions Using System Calls: Alternative Data Models. *IEEE Symposium on Security and Privacy*, 1999.

[21] Wespi A., Dacier M., and Debar H. Intrusion detection using variable-length audit trail patterns. *Proc. RAID*, 2000.

[22] Wespi A., Dacier M., and Debar H. An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm. P*roc. EICAR,* 1999.

[23] Witten I. and Bell T., The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. on Information Theory*, 1991.

Detection of Novel Network Attacks Using Data Mining

Levent Ertoz, Eric Eilertson, Aleksandar Lazarevic,
Pang-Ning Tan, Paul Dokas, Vipin Kumar, Jaideep Srivastava

*Computer Science Department, 200 Union Street SE, 4-192, EE/CS Building,
University of Minnesota, Minneapolis, MN 55455, USA*

{ertoz,eric,aleks,ptan,dokas,kumar,srivasta}@cs.umn.edu

Abstract

This paper introduces the Minnesota Intrusion Detection System (MINDS), which uses a suite of data mining techniques to automatically detect attacks against computer networks and systems. While the long-term objective of MINDS is to address all aspects of intrusion detection, in this paper we present two specific contributions. First, we present MINDS anomaly detection module that assigns a score to each connection that reflects how anomalous the connection is compared to the normal network traffic. Experimental results on live network traffic at the University of Minnesota show that our anomaly detection techniques have been successful in automatically detecting several novel intrusions that could not be identified using state-of-the-art signature-based tools such as SNORT. Many of these have been reported on the CERT/CC list of recent advisories and incident notes. We also present the results of comparing the MINDS anomaly detection module to SPADE (Statistical Packet Anomaly Detection Engine), which is designed to detect stealthy scans.

1. Introduction

Traditional methods for intrusion detection are based on extensive knowledge of attack signatures that are provided by human experts. The signature database has to be manually revised for each new type of intrusion that is discovered. A significant limitation of signature-based methods is that they cannot detect novel attacks. In addition, once a new attack is discovered and its signature developed, often there is a substantial latency in its deployment. These limitations have led to an increasing interest in intrusion detection techniques based upon data mining [3, 4, 21, 26, 28], which generally fall into one of two categories: misuse detection and anomaly detection.

In misuse detection, each instance in a data set is labeled as 'normal' or 'intrusive' and a learning algorithm is trained over the labeled data. Research in misuse detection has focused mainly on detecting network intrusions using various classification algorithms [3, 10, 21, 24, 26, 33], rare class predictive models [14-17, 19], association rules [3, 21, 28] and cost sensitive modeling [9, 16]. Unlike signature-based intrusion detection systems, models of misuse are created automatically, and can be more sophisticated and precise than manually created signatures. In spite of the fact that misuse detection models have high degree of accuracy in detecting known attacks and their variations, their obvious drawback is the inability to detect attacks whose instances have not yet been observed. In addition, labeling data instances as normal or intrusive may require enormous time for many human experts.

Anomaly detection algorithms build models of normal behavior and automatically detect any deviation from it [7, 12]. The major benefit of anomaly detection algorithms is their ability to potentially detect unforeseen attacks. In addition, they may be able to detect new or unusual, but non-intrusive, network behavior that is of interest to a network manager, and needs to be added to the normal profile. A major limitation of anomaly detection systems is a possible high false alarm rate. There are two major categories of anomaly detection techniques, namely supervised and unsupervised methods. In supervised anomaly detection, given a set of normal data to train from, and given a new piece of test data, the goal is to determine whether the test data belongs to "normal" or to an anomalous behavior. Recently, there have been several efforts in designing supervised network-based anomaly detection algorithms, such as ADAM [3], PHAD [27], NIDES [2], and other techniques that use neural networks [32], information theoretic measures [22], network activity models [6] etc. Unlike supervised anomaly detection where the models are built only according to the normal behavior on the network, unsupervised anomaly detection attempts to detect anomalous behavior without using any knowledge about the training data. Unsupervised anomaly detection approaches are based on statistical approaches [36, 37],

clustering [8], outlier detection schemes [1, 5, 18, 31], state machines [34], etc.

This paper introduces the Minnesota Intrusion Detection System (MINDS) that uses a suite of data mining techniques to automatically detect attacks against computer networks and systems. While the long-term objective of MINDS is to address all aspects of intrusion detection, in this paper we present only an anomaly detection technique that assigns a score to each network connection reflecting how anomalous the connection is compared to the normal network traffic. We also provide an evaluation of MINDS anomaly detection schemes in the context of real life network data at the University of Minnesota. During the last year, this evaluation has shown that anomaly detection algorithms have been successful in automatically detecting numerous novel intrusions that could not be identified using widely popular tools such as SNORT [35]. In fact, many of these attacks have been reported on the CERT/CC (Computer Emergency Response Team/Coordination Center) list of recent advisories and incident notes. We chose to present results on real life network data since publicly available data sets for evaluation of network intrusion detection systems (e.g. DARPA 1998, DARPA 1999 data sets [23, 25]) are known to have serious limitations [29]. In the absence of labels of network connections (normal vs. intrusive), we are unable to provide real estimate of detection rate, but nearly all connections that are ranked highly by our anomaly detection algorithms are found to be interesting and anomalous by the network security analyst on our team.

2. The MINDS System

The Minnesota Intrusion Detection System (*MINDS)* is a data mining based system for detecting network intrusions. Figure 1 illustrates the process of analyzing real network traffic data using the *MINDS* system. Input to MINDS is Netflow version 5 data collected using Netflow tools. Netflow tools only capture packet header information (i.e., they do not capture message content), and build one way sessions (flows). We are working with Netflow data instead of tcpdump because we currently do not have the capacity to collect and store the tcpdump. Netflow data for each 10 minute window, which typically result in 1 to 2 million flows, is stored in a flat file. The analyst uses MINDS to analyze these 10-minute data files in a batch mode. Before applying MINDS to these data files, a data filtering step is performed by the system administrator to remove network traffic that the analyst is not interested in analyzing. For example, the removed attack-free network data in data filtering step may include the data coming from trusted sources, non-interesting network data (e.g. portions of *http* traffic) or unusual/anomalous network behavior for which it is known that it does not correspond to intrusive behavior.

The first step in MINDS includes constructing features that are used in the data mining analysis. Basic features include source IP address and port, destination IP address and port, protocol, flags, number of bytes, and number of packets. Derived features include time-window and connection-window based features. Time-window based features are constructed to capture connections with similar characteristics in the last *T* seconds, since typically of Denial of Service (DoS) and scanning attacks involve hundreds of connections. A similar approach was used for constructing features in KDDCup'99 data [39]. Table 1 summarizes the time-window based features.

"Slow" scanning attacks, i.e. those that scan the hosts (or ports) and use a much larger time interval than a few seconds, e.g. one scan per minute or even one scan per hour, cannot be detected using derived "time-window" based features. To capture these types of the attacks, we also derive "connection-window" features that capture the same characteristics of the connection records as time-window based features, but are computed in the last *N* connections. The connection-window based features are shown in Table 2.

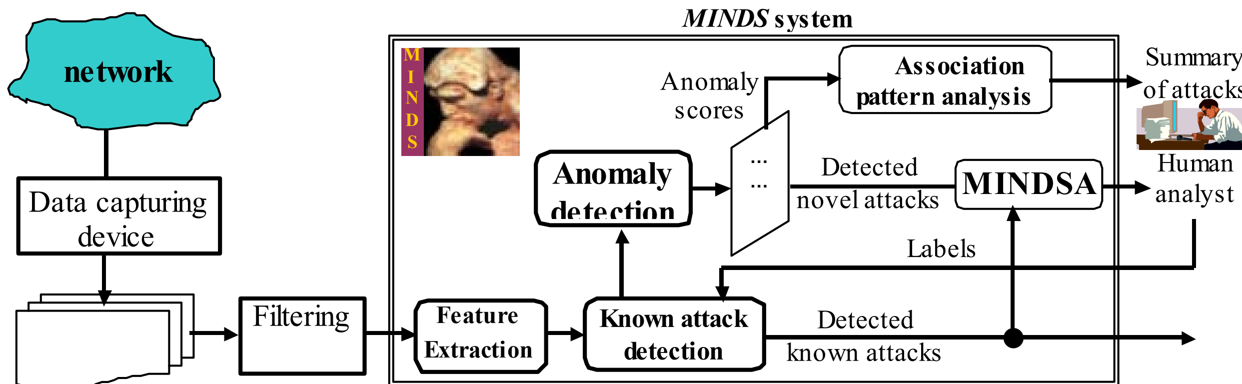

Figure 1 Architecture of MINDS system

Feature Name	Feature description
count_dest_conn	For the same source IP address, number of unique destination IP addresses inside the network in the last *N* connections
count_src_conn	For the same destination IP address, number of unique source IP addresses inside the network in the last *N* connections
count_serv_ src_conn	Number of connections from the source IP to the same destination port in the last *N* connections
count_serv_ dest_conn	Number of connections from the destination IP to the same source port in the last *N* connections

Table 1 The extracted "time-window" features

Feature Name	Feature description
count_dest	For the same source IP address, number of unique destination IP addresses inside the network in the last *T* seconds
count_src	For the same destination IP address, number of unique source IP addresses inside the network in the last *T* seconds
count_serv_src	Number of connections from the source IP to the same destination port in the last *T* seconds
count_serv_dest	Number of connections from the destination IP to the same source port in the last *T* seconds

Table 2 The extracted "connection-window" based features

After the feature construction step, the known attack detection module is used to detect network connections that correspond to attacks for which the signatures are available, and then to remove them from further analysis. For experiments reported in this paper, this step is not performed.

Next, the data is fed into the *MINDS* anomaly detection module that uses an outlier detection algorithm to assign an anomaly score to each network connection. A human analyst then has to look at only the most anomalous connections to determine if they are actual attacks or other interesting behavior.

3. MINDS Anomaly Detection

In this section, we only present the density based outlier detection scheme used in our anomaly detection module. For more detailed overview of our research in anomaly detection the reader is referred to [20].

MINDS anomaly detection module assigns a degree of being an outlier to each data point, which is called the local outlier factor (LOF) [5]. The outlier factor of a data point is local in the sense that it measures the degree of being an outlier with respect to its neighborhood. For each data example, the density of the neighborhood is first computed. The *LOF* of specific data example *p* represents the average of the ratios of the density of the example *p* and the density of its nearest neighbors. To illustrate advantages of the *LOF approach*, consider a simple two-dimensional data set given in Figure 2. It is apparent that the density of cluster C_2 is significantly higher that the density of cluster C_1. Due to the low density of cluster C_1 it is apparent that for every example *q* inside cluster C_1, the distance between the example *q* and its nearest neighbor is greater than the distance between the example p_2 and its nearest neighbor, which is from cluster C_2, and therefore example p_2 will not be considered as outlier.

Hence, the simple nearest neighbor approach based on computing the distances fail in these scenarios. However, the example p1 may be detected as outlier using the distances to the nearest neighbor. On the other side, LOF is able to capture both outliers (p1 and p2) due to the fact that it considers the density around the points.

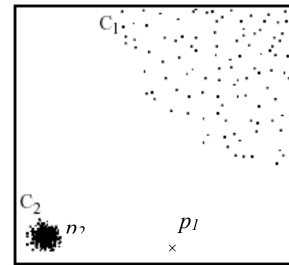

Figure 2 Outlier Examples

LOF requires the neighborhood around all data points be constructed. This involves calculating pair-wise distances between all data points which is an $O(n^2)$ process, which makes it computationally infeasible for millions of connections. To address this problem, we sample the data to use as a training set and compare all data points to this small set, which reduces the complexity to $O(n{\cdot}m)$ where n is the size of the data and m is the size of the sample. Apart from achieving computational efficiency by sampling, anomalous network behavior will not be able to match enough examples in the sample to be called normal. This is because rare behavior will not be represented in the sample.

4. Experimental Evaluation of MINDS Anomaly Detection

The output of the MINDS anomaly detector contains the original Netflow data with the addition of the

anomaly score and relative contribution of each of the 16 attributes to the score. Table 3 shows the 16 attributes. The analyst typically looks at only the top few connections that have the highest anomaly scores. Figure 3 shows the most anomalous connections found by MINDS on January 26th in a 10-minute window, 48 hours after the slammer attack. The University of Minnesota network security analyst has been using MINDS and SNORT independently to analyze the university network traffic for the past seven months. During this period, MINDS has been successful in detecting many novel network attacks and emerging network behavior that could not be detected using SNORT.

In the following, we present a few examples that demonstrate the effectiveness of the MINDS anomaly detection algorithm. In addition, we present a comparison of MINDS performance on detecting scans to SPADE because it is already integrated into SNORT and thus available as open source. Note that other schemes exist that work in high bandwidth environments; e.g. the scheme presented in [38] identifies packets that are likely to be a probe and performs scan detection on only those packets. But most such schemes are not available as open source. Note that the comparison with SPADE is only restricted to detecting scans, as SPADE is not meant to find policy violations and worms, which can be detected by MINDS. We are unable to provide comparisons of MINDS with other anomaly detection systems that are potentially capable of finding intrusions other than scans, as either they are available only in commercial products [38], or require sanitized training data [27].

4.1 MINDS Anomaly Detection Results

Anomalies/attacks picked by MINDS include scanning activities, worms, and non-standard behavior such as policy violations and insider attacks. Many of these attacks detected by MINDS, have already been on the CERT/CC list of recent advisories and incident notes.

- On January 26, 2003, 48 hours after the “SQL Slammer/Sapphire” worm started, network connections related to the worm were only about 2% of the total traffic. Despite this, they were still ranked at the top by the anomaly detection algorithm (see Figure 3). The network connections that are part of the “slammer worm” are highlighted in light gray in Figure 3. It can be observed that the highest contributions to anomaly score for these connections were due to the features 9 and 11 (count_dest and count_serv_src from Table 1). This was due to the fact that the infected machines outside our network were still trying to communicate with many machines inside our network. Similarly, it can be observed from Figure 3 that during this time interval there is another scanning activity (ICMP ping scan, highlighted in dark gray) that was detected again mostly due to the features 9 and 11. The two non-shaded flows are replies from Half-Life game servers (running on port 27016/udp). They were flagged anomalous because those machines were talking to only port 27016/udp. For web connections, it is common to talk only on port 80, and it is well represented in the normal sample. However, since Half-Life connections did not match any normal samples with high counts on feature 15, they became anomalous.

score	srcIP	sPort	dstIP	dPort	protocc	flags	packets	bytes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
37674.69	63.150.X.253	**1161**	128.101.X.29	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.59**	0	0	0	**0**	0
26676.62	63.150.X.253	**1161**	160.94.X.134	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.59**	0	0	0	**0**	0
24323.55	63.150.X.253	**1161**	128.101.X.185	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.58**	0	0	0	**0**	0
21169.49	63.150.X.253	**1161**	160.94.X.71	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.58**	0	0	0	**0**	0
19525.31	63.150.X.253	**1161**	160.94.X.19	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.58**	0	0	0	**0**	0
19235.39	63.150.X.253	**1161**	160.94.X.80	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.58**	0	0	0	**0**	0
17679.1	63.150.X.253	**1161**	160.94.X.220	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.81**	0	**0.58**	0	0	0	**0**	0
8183.58	63.150.X.253	**1161**	128.101.X.108	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.58**	0	0	0	**0**	0
7142.98	63.150.X.253	**1161**	128.101.X.223	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
5139.01	63.150.X.253	**1161**	128.101.X.142	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
4048.49	142.150.X.101	0	128.101.X.127	2048	1	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
4008.35	200.250.X.20	27016	128.101.X.116	4629	17	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0**	0	**0**	0	0	0	**1**	0
3657.23	202.175.X.237	27016	128.101.X.116	4148	17	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0**	0	**0**	0	0	0	**1**	0
3450.9	63.150.X.253	**1161**	128.101.X.62	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
3327.98	63.150.X.253	**1161**	160.94.X.223	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
2796.13	63.150.X.253	**1161**	128.101.X.241	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
2693.88	142.150.X.101	0	128.101.X.168	2048	1	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
2683.05	63.150.X.253	**1161**	160.94.X.43	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
2444.16	142.150.X.236	0	128.101.X.240	2048	1	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
2385.42	142.150.X.101	0	128.101.X.45	2048	1	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
2114.41	63.150.X.253	**1161**	160.94.X.183	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
2057.15	142.150.X.101	0	128.101.X.161	2048	1	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1919.54	142.150.X.101	0	128.101.X.99	2048	1	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1634.38	142.150.X.101	0	128.101.X.219	2048	1	16	[2,4)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1596.26	63.150.X.253	**1161**	128.101.X.160	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
1513.96	142.150.X.107	0	128.101.X.2	2048	1	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1389.09	63.150.X.253	**1161**	128.101.X.30	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
1315.88	63.150.X.253	**1161**	128.101.X.40	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.82**	0	**0.57**	0	0	0	**0**	0
1279.75	142.150.X.103	0	128.101.X.202	2048	1	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1237.97	63.150.X.253	**1161**	160.94.X.32	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1180.82	63.150.X.253	**1161**	128.101.X.61	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0
1107.78	63.150.X.253	**1161**	160.94.X.154	**1434**	17	16	[0,2)	[0,1829)	0	0	0	0	0	0	0	0	**0.83**	0	**0.56**	0	0	0	**0**	0

Figure 3 Most anomalous connections found by MINDS anomaly detection algorithm in a 10-minute window 48 hours after the “slammer worm” started.

- On October 10th, our anomaly detection module detected two activities of the slapper worm that were not identified by SNORT since they were variations of an existing worm code. Once a machine is infected with the worm, it communicates with other machines that are also infected and attempts to infect other machines. The most common version of the worm uses port 2002 for communication, but some variations use other ports. Our anomaly detector flagged these connections as anomalous for two reasons. First, the source or destination ports used in the connection may not have been rare individually but the source-destination port pairs were very rare (the anomaly detector does not keep track of the frequency of pairs of attributes; however, while building the neighborhoods of such connections, most of their neighbors will not have the same source-destination port pairs, which will contribute to the distance). Second, the communication pattern of the worm looks like a slow scan causing the value of the variable count_serv_src_conn (number of connections from the source IP to the same destination port in the last *N* connections) to become large. SNORT has a rule for detecting worm that uses port 2002 (and a few other ports), but not for all possible variations. A single general SNORT rule can be written to detect the variations of the worm at the expense of a higher false positive rate.

1 source IP	5 protocol
2 destination IP	6 duration
3 source Port	7 bytes/packet
4 destination Port	8 # packets
9 cnt_dest	13 cnt_src
10 cnt_dest_conn	14 cnt_src_conn
11 cnt_serv_src	15 cnt_serv_dest
12 cnt_serv_src_conn	16 cnt_serv_dest_conn

Table 3 List of features used in anomaly detection

- On August 9th, 2002, CERT/CC issued an alert for "widespread scanning and possible denial of service activity targeted at the Microsoft-DS service on port 445/TCP" as a novel Denial of Service (DoS) attack. In addition, CERT/CC also expressed "interest in receiving reports of this activity from sites with detailed logs and evidence of an attack." This type of attack was the top ranked outlier on August 13th, 2002, by our anomaly detection module in its regular analysis of University of Minnesota traffic. The port scan module of SNORT could not detect this attack, since the port scanning was slow.
- On August 13th, 2002, our anomaly detection module detected "scanning for an Oracle server" by ranking connections associated with this attack as the second highest ranked block set of connections (the top ranked block of connections belonged to the denial of service activity targeted at the Microsoft-DS service on port 445/TCP). This type of attack is difficult to detect using other techniques, since the Oracle scan was embedded within much larger Web scan, and the alerts generated by Web scan could potentially overwhelm the human analysts. On June 13th, CERT/CC had issued an alert for the attack.
- On August 8th and 10th, 2002, our anomaly detection techniques detected a machine running a Microsoft PPTP VPN server, and another one running a FTP server, which are policy violations, on non-standard ports. Both policy violations were the top ranked outliers. Our anomaly detector module flagged these servers as anomalous since they are not allowed, and therefore very rare. Since SNORT is not designed to look for rogue and unauthorized servers, it was not able to detect these activities. In addition, for the PPTP VPN server, the collected GRE traffic is part of the normal traffic, and not analyzed by tools such as SNORT.
- On January 27, 2003, our techniques detected odd, not routable RFC1918 traffic coming from the Internet. RFC1918 (Request for Comments) serves as Address Allocation for Private Internets, while RFC1918 blocks are segments of IP address space reserved by IANA (Internet Assigned Numbers Authority) for use within an organization. DNS records for RFC1918 addresses are legitimate only within the network on which a host with RFC1918 address resides. However, RFC1918 addresses are not globally routed and they should not appear on the public Internet.
- On February 6, 2003, our technique detected that the IP address 128.101.6.0, which does not correspond to a real computer, but to a network itself, has been targeted with IP Protocol 0 traffic from Korea (61.84.X.97). This type of network traffic is "exceedingly" bad as IP Protocol 0 is not legitimate.
- On February 6, 2003, our techniques detected a computer on the network apparently communicating with a computer in California over a VPN. This scenario in the worst case may correspond to a covert channel by which someone might be gaining access to the University network in an unauthorized way, and in the best case to someone at the University creating unauthorized tunnels between the University and some other network, which is not allowed. However, both types of behavior are extremely useful for security analysts.
- On February 7, 2003, a computer in the CS department talking on IPv6 was detected using our techniques. This type of communication is extremely rare and represents a possible covert tunnel to the outside world. The follow-up analysis diagnosed that a suspect who was doing this is on system staff and is in fact using this as a covert tunnel to his home computers.
- On February 6, 2003, our anomaly detection techniques detected unsolicited ICMP ECHOREPLY

messages to a computer previously infected with Stacheldract worm (a DDos agent). Although infected machine has been removed from the network, other infected machines outside our network were still trying to talk to infected machine from our network.

4.2 Comparison of MINDS and SPADE

SPADE: A brief overview. SPADE is a SNORT plug-in that automatically detects stealthy port scans [36]. Unlike traditional scan detectors that look for X events in Y seconds, SPADE takes a radically different approach and looks at the amount of information gained by probing. It has four different methods of calculating the likelihood of packets. However, the most successful method measures the direct joint probability P(dest IP, dest Port). SPADE examines TCP-SYN packets and maintains the count of packets observed on (destIP, destPort) tuples. When a new packet is observed, SPADE checks the probability of observing that packet on the (dest IP, dest Port) tuple. The lower the probability, the higher the anomaly score. Note that SPADE raises alarms on individual SYN packets regardless of how many other destination IP/ports have been scanned by the same source. In the case of an IPsweep, the scanner will eventually touch a machine that does not have the service being scanned for and therefore will raise a SPADE alarm. In the case of a portscan, an alarm will be raised when the scanner touches a port for which the service is not available on the target machine. In addition, SPADE will raise false alarms on legitimate traffic for which (destination IP, destination port) combinations are infrequent. This will be even more prevalent on outbound connections. The reason is that the number of IPs outside is much bigger than the number of IPs inside the network. As reported in [36], on DARPA99 data, as the number of variables in the direct joint probability is increased to include the source IP and source Port, the accuracy of SPADE decreases. This can be attributed to the fact that when extra attributes are used, the model essentially becomes sparser and therefore gives more false alarms.

MINDS and SPADE both assign a score to each connection that indicates its degree of being an outlier. They are both unsupervised anomaly detection schemes since neither one requires a labeled training set. The key difference is in the method for computing the anomaly score. In SPADE, the anomaly score is inversely related to the probability of observing the connection based upon the features used. If too many features are used (or if any feature has too many values), then the probability estimates are not reliable. The reason is that many of the legitimate combinations of features may be previously unseen or infrequent. In contrast, MINDS does not suffer from increased dimensionality as much as SPADE does, since MINDS constructs neighborhoods around data points which is a better estimate of actual probabilities when there is not enough data to adequately develop the model. This allows MINDS to use a large number of features as long as they are not spurious. Even if MINDS used only two features (destination IP and destination port), we argue that it can provide higher quality outlier scores. To illustrate this consider the following probability distribution (Table 4), where blank represents no occurrences. If we observe a packet P1 on IP2/Port3 and another packet P2 on IP4/Port1, SPADE will assign equal anomaly scores to both of the packets. However, one could argue that P2 should be more anomalous than P1 since in the case of P2 neither the port (Port1) nor the IP (IP4) by themselves are used frequently, whereas for P1, both the port (Port3) and the IP (IP2) are frequently used.

If we use MINDS anomaly detection module with only 2 attributes, namely the destination IP and destination port, packet P2 will be assigned a higher anomaly score than P1. The reason is that P1 will be closer to its neighbors than P2 will be to its neighbors. Both P1 and P2 will have neighbors that are in dense regions. If we compare the ratios of densities of P1 and P2 to the density of their respective neighbors, P2 will have a lower ratio, hence a higher anomaly score.

Frequency	IP1	IP2	IP3	IP4
Port1		Low		**P2**
Port2		High		
Port3	High	**P1**	High	
Port4	Low	High	Low	

Table 4. IP/Port frequency distribution

We ran the latest version of SPADE (v021031.1) on live network traffic at the University of Minnesota for a 10-minute period using a threshold of 8. It generated 296,921 alarms out of approximately one million TCP_SYN packets received during this 10-minute window. Nearly 26% (76,956) of the alarms were on inbound packets. Vast majority of outbound packets were false alarms. This is not surprising since the space of (IP/Port) combinations outside our network is very large and the data is very sparse. This is a serious limitation of SPADE, since outbound alarms tend to be very important, as they often indicate infected machines inside the network. In the rest of the discussion, we focus on alarms on inbound packets. 25% (19020) of inbound alarms were to web alarms, 6% (4608) to common services other than web (https, mail, ftp, ssl enabled imap, web proxies), 28.5% (21895) to peer-to-peer applications (kazaa, gnutella, edonkey) and the remaining 41% (31433) were hard to interpret. If we ignore repetitions of the same alarm (same source IP, destination IP, destination Port), we are left with 28973 alarms. There were a total of 21669 unique sources of

alarms, 20825 of them generated only 1 or 2 alarms. (More detailed information about the distributions is given in Table 5.) Although some of these alerts may indicate very slow stealthy scans, majority of these alerts are likely to be for legitimate connections on rarely used destination IP, destination port combination. Specifically, we can say that the alarms on p2p applications are false alarms (not scans) since p2p applications get the list of active servers from their super-nodes instead of scanning for machines running these applications. In any case, one cannot expect from a system administrator to investigate alarms for so many different sources in a short 10-minute window.

If we raise our threshold and look at only top 10,000 alerts, the distribution is very similar. If we look at the breakdown of alarms by type, in the top 10,000 (threshold = 12.6108), web alerts are 59% (5927), common services are 7.5% (754), p2p applications are 9% (923) and the remaining, hard-to-interpret ones are 24% (2396). There are 3306 unique sources that generate 10,000 alarms, 3159 of which raised either 1 or 2 alarms. If we look at top 1000, 497 out of 547 unique sources raised less than 3 alarms. If we look at top 100, 73 out of 78 unique sources raised less than 3 alarms.

If we do a similar analysis for alarms on web, P2P (peer to peer), common services and the remaining alarms separately, we still get a very similar picture; the number of unique sources generating very few alarms is very large. We argue that most of SPADE alarms are effectively false alarms as the number of unique sources generating the alarms is at a high percentage regardless of the threshold.

SPADE does find some stealthy scans that will be hard to find using simple scan detection schemes that look for source IP's that connect to more than X destination Ports / IP's in a specified time or connection window. For example, among the top 10,000 alarms 66 unique sources generated at least 10 or more alarms. Each one of these sources were either scanning a specific IP for 10 or more ports, or scanning 10 or more IP's for a specific port. But in the process of finding these hard to detect stealthy scans, SPADE generated false alarms for far too many legitimate connections.

Number of alarms	1	2	3	4	
Number of sources	19282	1543	394	151	
Number of alarms	5-6	7-20	21-50	51-150	151+
Number of sources	139	106	45	5	4

Table 5. Distribution of number of alarms from unique sources (inbound alarms)

For the same 10-minute period, we ran MINDS and asked our security expert to analyze top few hundred anomalous connections ranked by the anomaly detector. Most of these were scans, which were interleaved with few non-scan connections. Most of these scans targeted dozens of IPs inside the University of Minnesota networks. Among non-scan anomalous connections, our security expert identified a local machine running a web proxy open to everyone, and lots of people were browsing the web through the proxy to anonymize their connections. Proxy settings were fixed after the issue was identified. For privacy reasons, we cannot report the individual connections but only provide a high level summary. Note that stealthy scans that target only a few machines during the 10-minute window are not likely to receive high anomaly score by MINDS.

4.3 MINDS anomaly detection module versus SNORT

Here we compare general capabilities of SNORT and MINDS in detecting the following categories of attacks and irregular behavior:

- content-based attacks
- scanning activities
- policy violations

Content based attacks. These attacks are out of scope for our anomaly detection module since it does not consider the content of the packets, and therefore SNORT is superior in identifying those attacks. However, SNORT is able to detect only those content-based attacks that have known signatures/rules. Despite the fact that SNORT is more successful in detecting the content-based attacks, it is important to note that once a computer has been attacked successfully, its behavior could become anomalous and therefore detected by our anomaly detection module, as seen in previous examples.

Scanning activities. When detecting various scanning activities SNORT and MINDS anomaly detection module have similar performance for certain types of scans, but they have very different detection capabilities for other types. There are two categories of scanning activities, where SNORT and our anomaly detection module might have different detection performance:

- Fast (regular) scans
- Slow scans

When detecting regular scans, SNORT portscan module keeps track of the number of destination IP addresses accessed by each source IP address in a given time window (default value is 3 seconds). Let's denote this variable count_dest, already defined in Table 1. Whenever the value of count_dest is above a specified threshold (SNORT default value is 4), SNORT raises an alarm, thus indicating a scan by the source IP address. Our anomaly detection module is also able to assign high anomaly score to such network connections, since for most normal connections the value of count_dest is low. In addition, connections from many types of scanning

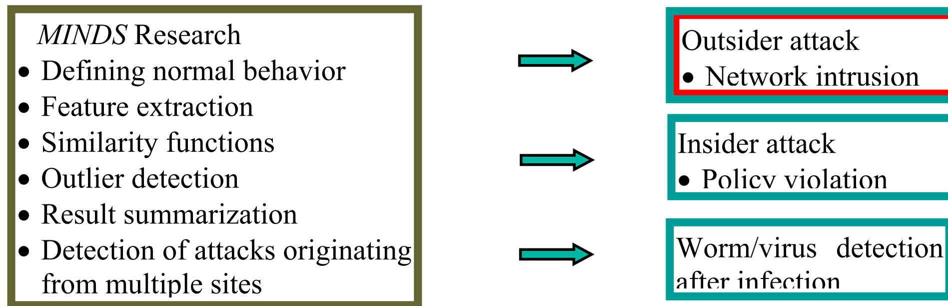

Figure 4 Three types of threats that can be detected by MINDS anomaly detection module

activities tend to have other features that are unusual (such as very small payload), which make additional contributions to the anomaly score.

A scan can be detected by SNORT provided the scan is fast enough for chosen time window (default value is 3 seconds) and count threshold (default value is 4). If a scanning activity is not fast enough (outside specified parameters), it will not be detected by SNORT. However, SNORT can still detect such activities by increasing the time window and/or decreasing the number of events counted within the time-window, but this will tend to increase false alarm rate. On the other side, our anomaly detection module is more suitable for detecting slow scans since it considers both time-window based and connection-window based features (as opposed to SNORT that uses only time-window based features), as well as other features of the connections such as number of packets, number of bytes per packet, etc.

Policy violations. MINDS anomaly detection module is much more successful than SNORT in detecting policy violations (e.g. rogue and unauthorized services), since it looks for unusual network behavior. SNORT may detect these policy violations only if it has a rule for each of these specific activities. Since the number and variety of these activities can be very large and unknown, it is not practical to incorporate them into SNORT for the following reasons. First, processing of all these rules will require more processing time thus causing the degradation in SNORT performance. It is important to note that it is desirable for SNORT to keep the amount of analyzed network traffic small by incorporating rules as specific as possible. On the other hand, very specific rules limit the generalization capabilities of a typical rule based system, i.e., minor changes in the characteristics of an attack might cause the attack to be undetected.

Second, SNORT's static knowledge has to be manually updated by human analysts each time a new suspicious behavior is detected. In contrast, MINDS anomaly detection module is adaptive in nature, and it is particularly successful in detecting anomalous behavior originating from a compromised machine (e.g. attacker breaks into a machine, installs unauthorized software and uses it to launch attacks on other machines). Such behavior is often undetected by SNORT's signatures.

5. Conclusions and Future Work

The overall goal for MINDS is to be a general framework and system for detecting attacks and threats to computer systems. Data generated from network traffic monitoring tends to have very high volume, dimensionality and heterogeneity. Coupled with the low frequency of occurrence of attacks, this makes standard data mining algorithms unsuitable for detecting attacks. In addition, cyber attacks may be launched from several different locations and targeted to many different destinations, thus creating a need to analyze network data from several locations/networks in order to detect these distributed attacks. According to our initial analysis, the intrusions detected by MINDS are complementary to those of SNORT – a signature-based system. This implies that the two can be combined to increase overall attack coverage. In addition, MINDS will have a summarization and visualization tools to aid the analyst in better understanding anomalous/suspicious behavior detected by the anomaly detection engine.

The key anomaly detection approach used by MINDS is based on the analysis of unusual behavior, and is thus suitable for detecting many types of threats. Figure 4 shows three such types. First type of threats corresponds to outsider attacks that represent deviations from normal connection behavior. Second threat type is insider attack, where an authorized user logs into a system with malicious intent. However, the malicious behavior shown by such a user is often at variance with normal procedures, and our behavior-analysis based approach can pick it up as anomalous behavior, reporting it as a possible attack. Since no security mechanism is fool proof, an undetected successful outsider becomes equivalent to an insider attack, and the same ideas apply. Third threat type corresponds to a situation where a

virus/worm has entered an environment – either undetected by a perimeter protection mechanism such as virus scan of attachments, or through bringing in of an infected portable hardware device, e.g. a laptop. The unusual behavior shown by such a machine can potentially be detected by our approach of analyzing anomalous behavior.

A number of applications outside of intrusion detection have similar characteristics, e.g. detecting credit card and insurance frauds, early signs of potential disasters in industrial process control, early detection of unusual medical conditions – e.g. cardiac arrhythmia, etc. We plan to explore the use of our techniques to such problems.

Acknowledgments

This work was partially supported by Army High Performance Computing Research Center contract number DAAD19-01-2-0014 and NSF grants IIS-0308264, ACI-9982274. The content of the work does not necessarily reflect the position or policy of the government and no official endorsement should be inferred. Access to computing facilities was provided by the AHPCRC and the Minnesota Supercomputing Institute.

References

1. C. C. Aggarwal, P. Yu, Outlier Detection for High Dimensional Data, Proceedings of the ACM SIGMOD Conference, 2001.

2. Anderson, D., Lunt, T. F., Javitz, H., Tamaru, A., Valdes, A.: Detecting Unusual Program Behavior Using the Statistical Component of the Next-Generation Intrusion Detection Expert System (NIDES), Technical Report SRI-CSL-95-06, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995.

3. D. Barbara, N. Wu, S. Jajodia, Detecting Novel Network Intrusions Using Bayes Estimators, First SIAM Conference on Data Mining, Chicago, IL, 2001.

4. E. Bloedorn, et al., Data Mining for Network Intrusion Detection: How to Get Started, MITRE Technical Report, August 2001.

5. M. M. Breunig, H.P. Kriegel, R. T. Ng, J. Sander, LOF: Identifying DensityBased Local Outliers, Proceedings of the ACM SIGMOD Conference, 2000.

6. Cabrera, J. B. D., Ravichandran, B., Mehra, R. K.: Statistical Traffic Modeling For Network Intrusion Detection, Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, CA, 2000.

7. D.E. Denning, An Intrusion Detection Model, IEEE Transactions on Software Engineering, SE-13:222-232, 1987.

8. E. Eskin, A. Arnold, M. Prerau, L. Portnoy, S. Stolfo, A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data, Data Mining for Security Applications, Kluwer 2002.

9. W. Fan, S. J. Stolfo, J. Zhang, P. K. Chan, AdaCost: Misclassification Cost-sensitive Boosting, Proceedings of the Sixteenth International Conference on Machine Learning, 97-105, Bled, Slovenia, 1999.

10. A. Ghosh, A. Schwartzbard, A study in Using Neural Networks for Anomaly and Misuse Detection, Proceedings of the Eighth USENIX Security Symposium, 141--151, Washington, DC, August 1999.

11. Incident Storm Center, www.incidents.org.

12. H.S. Javitz, and A. Valdes, The NIDES Statistical Component: Description and Justification, Technical Report, Computer Science Laboratory, SRI International, 1993.

13. K.S.Jones: A Statistical Interpretation Of Term Specificity And Its Application In Retrieval, Journal of Documentation, 28 (1) 11-21, 1972.

14. M. Joshi, V. Kumar, R. Agarwal, Evaluating Boosting Algorithms to Classify Rare Classes: Comparison and Improvements, First IEEE International Conference on Data Mining, San Jose, CA, 2001.

15. M. Joshi, R. Agarwal, V. Kumar, Predicting Rare Classes: Can Boosting Make Any Weak Learner Strong?, Proceedings of Eight ACM Conference ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, Canada, 2002.

16. M. Joshi, R. Agarwal, V. Kumar, PNrule, Mining Needles in a Haystack: Classifying Rare Classes via Two-Phase Rule Induction, Proceedings of ACM SIGMOD Conference on Management of Data, May 2001.

17. M. Joshi, V. Kumar, CREDOS: Classification using Ripple Down Structure (A Case for Rare Classes), in review.

18. E. Knorr, R. Ng, Algorithms for Mining Distance-based Outliers in Large Data Sets, Proceedings of the VLDB Conference, 1998.
19. A. Lazarevic, N. Chawla, L. Hall, K. Bowyer, SMOTEBoost: Improving the Prediction of Minority Class in Boosting, AHPCRC Technical Report, 2002.
20. A. Lazarevic, L. Ertoz, A. Ozgur, V. Kumar, J. Srivastava: A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection, Proceedings of Third SIAM International Conference on Data Mining, May, San Francisco, 2003.
21. W. Lee, S. J. Stolfo, Data Mining Approaches for Intrusion Detection, Proceedings of the 1998 USENIX Security Symposium, 1998.
22. W. Lee, D. Xiang: Information-Theoretic Measures for Anomaly Detection, IEEE Symposium on Security and Privacy, 2001.
23. R. P. Lippmann, R. K. Cunningham, D. J. Fried, I. Graf, K. R. Kendall, S. W. Webster, M. Zissman, Re-sults of the 1999 DARPA Off-Line Intrusion Detection Evaluation, Proceedings of the Second International Workshop on Recent Advances in Intrusion Detection (RAID99), West Lafayette, IN, 1999.
24. R. Lippmann, R. Cunningham, Improving intrusion detection performance using keyword selection and neural networks, Computer Networks, 34(4):597--603, 2000.
25. R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. P. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman, Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation, Proceedings DARPA Information Survivability Conference and Exposition (DISCEX) 2000, Vol 2, pp. 12-26, IEEE Computer Society Press, Los Alamitos, CA, 2000.
26. J. Luo, Integrating Fuzzy Logic With Data Mining Methods for Intrusion Detection, Master's thesis, Department of Computer Science, Mississippi State University, 1999.
27. M. Mahoney, P. Chan: Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks, Proceedings of 8th International Conference on Knowledge Discovery and Data Mining, 376-385, 2002.
28. S. Manganaris, M. Christensen, D. Serkle, and K. Hermix, A Data Mining Analysis of RTID Alarms, Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection (RAID 99), West Lafayette, IN, September 1999.
29. J. McHugh, The 1998 Lincoln Laboratory IDS Evaluation (A Critique), Proceedings of the Recent Advances in Intrusion Detection, 145-161, Toulouse, France, 2000.
30. Net flow tools, www.splintered.net/sw/flow-tools.
31. S. Ramaswamy, R. Rastogi, K. Shim, Efficient Algorithms for Mining Outliers from Large Data Sets, Proceedings of the ACM SIGMOD Conference, 2000.
32. J. Ryan, M-J. Lin, R. Miikkulainen, Intrusion Detection with Neural Networks, Proceedings of AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management, 72-77, AAAI Press, 1997.
33. C. Sinclair, L. Pierce, S. Matzner, An Application of Machine Learning to Network Intrusion Detection, Proceedings of the 15th Annual Computer Security Applications Conference, 371-377, Phoenix, AZ, December 1999.
34. R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, S. Zhou: Specification Based Anomaly Detection: A New Approach for Detecting Network Intrusions, ACM Conference on Computer and Communications Security, 2002.
35. SNORT Intrusion Detection System. www.snort.org.
36. S. Staniford, J. Hoagland, J. McAlerney, Practical Automated Detection of Stealthy Portscans, Journal of Computer Security, vol. 10, No. 1-2, 105-136, 2002.
37. K. Yamanishi, J. Takeuchi, G. Williams, P. Milne, On-line Unsupervised Oultlier Detection Using Finite Mixtures with Discounting Learning Algorithms, Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 320-324, Boston, MA, 2000.
38. S. Robertson, E. V. Siegel, M. Miller, S. J. Stolfo, Surveillance Detection in High Bandwidth Environments, DARPA DISCEX, 2003.
39. KDD cup 99, kdd.ics.uci.edu/databases/kddcup99/task.html

Passive Operating System Identification From TCP/IP Packet Headers*

Richard Lippmann, David Fried, Keith Piwowarski, William Streilein
MIT Lincoln Laboratory
244 Wood Street, Lexington, MA 02173, USA
lippmann@ll.mit.edu

Abstract

Accurate operating system (OS) identification by passive network traffic analysis can continuously update less-frequent active network scans and help interpret alerts from intrusion detection systems. The most recent open-source passive OS identification tool (ettercap) rejects 70% of all packets and has a high 75-class error rate of 30% for non-rejected packets on unseen test data. New classifiers were developed using machine-learning approaches including cross-validation testing, grouping OS names into fewer classes, and evaluating alternate classifier types. Nearest neighbor and binary tree classifiers provide a low 9-class OS identification error rate of roughly 10% on unseen data without rejecting packets. This error rate drops to nearly zero when 10% of the packets are rejected.

1. Introduction

One of the most difficult tasks of security-conscious system administrators is maintaining accurate information on the numbers, locations, and types of hosts being protected. This information is useful for many purposes including configuring network-based intrusion detection systems and maintaining a site security policy.

Configuring network-based intrusion detection systems using operating system (OS) information makes it possible to prioritize the large numbers of extraneous alerts caused by failed attacks and normal background traffic. A re-analysis of the data presented in [10] demonstrates that simply knowing whether the OS of each web server on a class B network is a version of Microsoft Windows, Solaris, or any type of UNIX makes it possible to assign a low-priority to from 33% to 87% of remote-to-local alerts produced by the snort [12] intrusion detection system. Filtering is performed by dismissing an alert when the vulnerability associated with the alert cannot occur for the OS of that host. A prior analysis [5] also demonstrated that knowledge of the OS of monitored hosts on a few small networks could assign roughly 30% of all remote-to-local alerts to a lower priority. This approach is best used to perform a preliminary analysis of recently-connected hosts or of hosts that can not be actively scanned. Much greater alert filtering eliminating as many as 95% of all remote-to-local alerts can be achieved through knowledge of the exact OS version, server software types and versions, and the patch status of hosts [10].

Detecting recently installed hosts and identifying operating systems is also useful for maintaining a site security policy. A site policy may specify the types of hosts that are allowed and it may specifically disallow old OS versions or those that are not supported by the network infrastructure. Detecting newly installed hosts as soon as possible using passive approaches is particularly important because these hosts are often most likely to be running old or insecure operating systems or to be configured incorrectly and be most vulnerable to remote attacks. They also may represent a new host attached by inside attackers to capture traffic or passwords. Knowing the OS may help a system administrator gauge the threat posed by a recently installed host. For example, it may be more urgent to investigate a new host running Linux on a primarily Window’s network than to investigate a new Window’s OS on the same network.

Operating systems can be determined using two complimentary approaches. Active scanning provides detailed information episodically by actively sending queries to hosts while passive analysis of captured network traffic provides instantaneous real-time, but less detailed, information. Active scanning includes the use of automated and often expensive network management systems (e.g. see [1]), semi-automated use of more limited open-source tools such as nmap [7], and manual analysis via direct logins to each host. Active scanning provides the most information about each host, but its use is often limited. Because scanning consumes bandwidth and host resources and may reveal security weaknesses it is often performed infrequently. Scanning durations can also be long on enterprise networks, security analysts responsible

* This work was sponsored by the Federal Aviation Administration under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Table 1. TCP/IP features used in open-source tools.

Feature Name	Range	Description
TCP Window Size (WS)	0-65,535	Data bytes a sender can transmit without receiving an acknowledgement equal to buffer size available on the receiver.
IP Time to Live (TTL)	0-255	Number of routing hops allowed before the packet is dropped, decremented by one by each router (prevents accidental routing loops).
IP Don't Fragment (DF)	0-1	Instructs routers not to fragment this IP packet, but to drop it if it is too large for the next network segment.
TCP Max Segment Size Option* (MSS)	0-65,535	Maximum size of data component of packet that a receiver can accept.
TCP Window Scaling Option Flag* (WSO)	0-1	Flag that indicates the TCP scaling option is being used to obtain bigger WS windows.
TCP Selective Acknowledgments Options Flag* (SOK)	0-1	Flag that indicates when the TCP selective acknowledgements option was set.
TCP NOP Option Flag* (NOP)	0-1	Flag that indicates one or more NOP's were added to align other options on a word boundary.
Packet Size (PS)	0-255	Length of packet in bytes.
TCP Timestamp Option Flag* (TS)	0-1	Flag that indicates one of the TCP timestamp options was included.
SYN vs SYN-ACK Packet Flag (SYN)	0-1	Flag set to one for SYN-ACK packets and zero for SYN packets.

for maintaining network intrusion detection systems are often not permitted to scan parts of enterprise networks, SNMP host clients required to support network management systems may not be allowed because they open up potential vulnerabilities in hosts that can be exploited by attackers, and scanning using such tools as nmap [7] is often not allowed because it can cause some devices to halt. Passive traffic analysis has none of these limitations because no extra network traffic is generated. It can either provide a continuous update to the last active scan or be used when active scanning is not allowed.

A major goal of the work described in this paper is to determine how accurate the OSes of hosts can be determined using passive OS analysis of TCP/IP packet header information. Other goals are to evaluate existing open-source tools that perform OS identification and to develop and evaluate an improved classifier using data mining/pattern-classification techniques.

2. Passive OS "Fingerprinting"

The first paper we are aware of that described approaches to passive OS identification and provided a small database of examples was published in May 2000 [15]. This paper, and open-source tools that have been developed since then rely heavily on the concept of "passive OS fingerprinting". Passive fingerprinting is an extension of "active OS fingerprinting" described in [6] and included in the nmap scanner [7].

Active fingerprinting relies on the observation that the TCP/IP protocol specification does not clearly describe how to handle unusual situations and has left certain header field values and behaviors unspecified. Programmers implementing TCP/IP have made different decisions that have lead to a variety of behaviors and default field values in different OSes. In addition, some programmers have not followed the specifications correctly and some OSes have not incorporated more recent optional advanced features. This has led to "idiosyncrasies" that can be revealed by actively probing a host with both malformed and normal packets and observing the responses. Features extracted from responses can be used for OS classification. These features include default values in the TCP/IP header, flags that indicate whether advanced features are supported, flags that indicate different response behaviors including "no response" and data content in the reply message. All features for an unknown OS form a feature vector that is compared to a database of exemplars containing stored

Table 2. Characteristics of three open-source passive OS identification tools.

Tool	Features	Exemplars	Classes With Three or More Exemplars	Packets
siphon [13]	3	47	6	SYN, ACK
p0f [14]	8	150	14	SYN
ettercap [11]	10	1093	75	SYN, SYN-ACK

feature vectors from known OSes. If a stored exemplar exactly matches the responses of an unknown host, then that exemplar represents the "fingerprint" of the unknown host and the name of the host that originated the exemplar is reported. Nmap currently performs nine different types of tests on a host and extracts a core set of 17 features from the responses to queries in these tests. As many as 55 features are available when an OS responds to all queries. In addition, Nmap has a database containing 989 exemplars, each with feature values and an OS name. Although the performance of active OS fingerprinting has never been carefully evaluated, this approach is in widespread use and many system administrators believe that the results are accurate.

Passive OS fingerprinting doesn't send probes to the unknown host, but instead examines values of fields in the TCP/IP packet header. In contrast to active fingerprinting, passive OS fingerprinting relies only on default packet-header field values and not on other idiosyncrasies elicited by unusual packets. Packet-header field values represent less than a third of the core features used by nmap for active fingerprinting. Passive fingerprinting may thus not be as accurate as active fingerprinting and the concept of "fingerprinting" as used in nmap may not be appropriate. In particular, requiring an exact match to stored exemplars, creating a new class for every new OS, adding any new feature that might seem useful, and storing only exemplars that differ from those currently in a database, may not provide best performance.

3. Passive Open-Source Tools

Three open-source tools have been developed for passive OS classification. All use values of fields in TCP/IP packet headers to identify OSes. Using these values is attractive because header information can be obtained even when popular encrypted protocols such as SSH and HTTPS are used, header fields are well defined, and values are easily extracted. The first three features shown in Table 1 were recommended in [15] because they appeared to vary more with the source host than with characteristics of the transmission channel and they occur in every TCP/IP packet. All three features depend on the host configuration. The Time to Live (TTL) is normally equal to a power of two for most hosts. Because TTL values are decremented by one by every router in the path to the packet capture location, these values cannot be compared directly to those in stored exemplars. Instead, TTL values in captured packets are rounded up to the nearest power of two before the comparison.

A proof-of-concept tool named siphon [13] was developed that implemented suggestions presented in [15]. It extracted the first three features in Table 1 from TCP SYN or ACK packets sent from the source of TCP connections and compared these to stored values recorded previously for known operating systems. The database provided with this tool contains only 47 exemplars holding values of these three features along with corresponding OS names. Classification involves extracting features from new packets and finding the first exact match to a stored exemplar. If there is no match, then no OS is reported and the packet is "rejected".

Following the release of siphon, two other open-source tools were released. Characteristics of these tools and of siphon are shown in Table 2. To the best of our knowledge, none of these tools has been carefully evaluated. Table 2 shows the number of packet-header features extracted for each tool, the number of exemplars provided, the number of OS classes that contain three or more exemplars, and the types of packets that are analyzed. The numbers in this table exclude exemplars with ambiguous OS names or with names that are for scanning tools such as nmap [7] and not for operating systems.

Before any tool evaluations could be performed, it was necessary to map OS names to classes. This task was complicated because OS names were not entered uniformly. Names were normalized initially using the following steps: (1) Map all names to lower case and eliminate spaces and dashes; (2) Whenever a Linux kernel version is provided, use linux and the version as the class; (3) Whenever a Cisco IOS version is provided, use cisco and the version as the class; (4) Identify exemplars that list multiple versions of OSes and keep only the first entry in any list; (5) Keep only the first two decimal separated numbers found (e.g. 2.12 instead of 2.12.34); and (6) Unify different names that refer to the same OS (e.g. "sunos5.8/solaris8", "redhatlinux/redhat", "windows2k/windows2000). Every normalized name that occurs three or more times forms a separate class. Such classes

represent the more popular and frequently occurring OSes and make it possible to use cross-validation testing to estimate classification accuracy. Three exemplars were required to form a class because some ettercap exemplars appear to have been duplicated by setting the WSO feature to "missing" in an attempt to improve performance. When only two exemplars were available for a unique OS name, this practice would invalidate cross-validation testing results because the exemplars were not sampled independently. Exemplars for operating systems that occurred fewer than three times and that were clearly not members of the already labeled classes were placed in a class named "other". These are primarily network devices and rare OSes.

The last column in Table 2 shows the types of packets that are analyzed by each tool. Siphon analyzes the initiating SYN packet or following ACK packets from the originating source of a TCP connection, p0f analyzes only initiating SYN packets, and ettercap analyzes both the initiating SYN packet from the connection source and the initial SYN-ACK response from the destination. These choices affect the utility of these tools. Siphon and p0f can only analyze connections from the originating source of TCP connections. These include web, mail, ftp and other clients. Ettercap, however, can analyze both packets from client programs and responses from servers.

The three tools in Table 2 use different features. Siphon uses the first three features in Table 1, p0f uses the first eight features, and ettercap uses all features. The additional features used by p0f and ettercap are primarily related to TCP options set only on the initiating SYN and the SYN-ACK response that starts a TCP connection. Features that occur only in these packets are marked using asterisks in the first column of Table 1. The use of these options limits the application of these tools to initiating SYN and SYN-ACK packets, but also adds new features that can potentially help discriminate between different classifiers. Two other features are the packet size and a flag for ettercap to indicate whether the packet analyzed was a SYN or SYN-ACK packet. This flag doesn't indicate OS type, but would be useful if options were used differently for the two packet types.

All three open-source tools report an OS with confidence only if a perfect match is found between features extracted from packets and features in a stored exemplar. Although the more recent tools will find inexact matches, they deprecate the OS reported in this situation and indicate that these names are not to be trusted. In addition, although there may be multiple exemplars in a database that match perfectly, all tools report only the OS of the first exemplar found in the database with an exact match. This behavior is used as a benchmark to represent these tools in the remainder of this paper.

4. Assumptions and Caveats

Results presented in the remainder of this paper are purposefully limited in scope and need to be interpreted with some important caveats in mind. The most important is that, as noted in [15], many features used to classify OSes are default values in packet headers. Some of these default values such as TTL can easily be changed to allow one OS to masquerade as another. Although this is currently uncommon, it is possible and sometimes recommended to defeat both active and passive OS identification. In the remainder of this paper it is assumed that default header values have not been altered.

Another concern is that network devices such as proxy firewalls and the types of traffic normalization suggested in [8] modify some packet header values used for OS identification. Again, it will be assumed that packets captured to form exemplars for OS identification have not been modified (except for the normal decrement in TTL at each router) by network devices. It will also be assumed that the OS labels and feature vectors that make up exemplars in ettercap are correct. These assumptions will be explored using new unseen test data with carefully captured packets and direct confirmation of OS names.

Finally, this work focuses solely on features extracted from single packet headers. Other information that could potentially be used for passive OS identification is not considered. Some of this other information is not always available and can only be used opportunistically. For example, the network interface card manufacturer indicated by the ethernet MAC address is only available on a host's local network and the content of FTP, Web, and SNMP server banners is only available from servers and if another host happens to query the server. In addition, statistical multi-packet features such as those used in [3] that require observing from 10 to 100 consecutive TCP connections per host are not considered because the initial focus is on single-packet performance.

Table 3. Numbers of SYN and SYN-ACK exemplars in ettercap and LL-test data.

Database	SYN	SYN-ACK	TOTAL
Ettercap	355	738	1093
LL-test	95	104	199

5. Evaluation Approach

All evaluations used the two databases shown in Table 3. Ettercap data contains 355 exemplars extracted from SYN packets and 738 exemplars extracted from SYN-

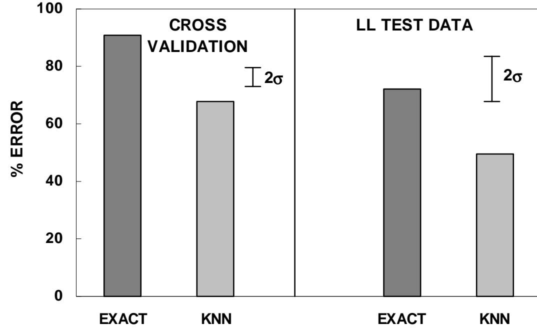

Figure 1. Baseline 75-class error rates for exact-match and k-nearest neighbor classifiers.

ACK packets. P0f and siphon data is not used because few exemplars are provided and some appear to be already included in ettercap data. The LL-test database (labeled LL for Lincoln Laboratory) was created for this study. SYN packets were received by a mail and SSH server on a local class-C network over two days. SYN-ACK packets were received in response to an nmap port scan performed on the same network. Features were extracted using a modified version of ettercap. Operating systems in the LL database were determined by logging directly into each host on this network because nmap active scanning was found to be inaccurate for some hosts. All unique exemplars were kept for each of 83 hosts. Care was taken to make sure OS names were correct by verifying the MAC address for hosts that use the DHCP protocol to obtain IP addresses and by verifying the status of dual-boot hosts and portable laptops when packets were captured. Ettercap exemplars were used for training classifiers and for cross-validation testing. LL-test exemplars were never used to train or tune classifiers. They were only used to estimate generalization error on previously unseen data.

All evaluations used all features listed in Table 1 or subsets of these features. Features were pre-processed to make it possible to explore alternate classifiers using LNKnet pattern classification software [9]. As noted above, if the TTL was not a power of two, it was rounded up to the next power of two. This feature, WS, MSS, and PS all are large numbers that typically vary by powers of two across different OSes. Instead of using these numbers directly as inputs to classifiers, the log base 2 of these numbers was used to make the typical changes observed in these features similar to those of binary features. All other features were binary. They indicated the presence or absence of a TCP option (DF, WSO, SOK, NOP, TS) or whether the packet analyzed to produce the exemplar was a SYN or SYN-ACK packet (SYN). The value used for the window scaling option (WSO) was not used as a feature because values were present infrequently and they appear to depend more on the channel than the host. Only a binary WSO feature was used that specified the presence of this option. The features WSO, PS, and MSS were sometimes missing in ettercap exemplars. Missing features were replaced with the most common value of that feature.

6. Baseline 75-Class Experiments

Initial 10-fold cross-validation experiments were performed using the 1093 ettercap exemplars grouped into 75 classes including the "other" class as described above. In this analysis, all ettercap exemplars are divided into 10 splits with roughly equal numbers of exemplars in each split. Ten evaluations are performed using exemplars from 9 of the 10 splits as training data and exemplars from the remaining split for testing. Errors from the 10 evaluations are then combined. This can be thought of as a retrospective analysis of the performance of ettercap. Since exemplars are being added to ettercap over time, its performance changes and cross-validation performance can be thought of as the performance expected for new exemplars not seen before.

After combining OS names into classes, it was found that the top 20 classes with the most exemplars contain roughly 50% of the exemplars not in the "other" class. Labels for these classes are bsdi, freebsd2.2, bsdi, freebsd2.2, linux, linux2.0, linux2.1, linux2.2, macosx, solaris2.62, solaris7, solaris8, windows, windows2000, windows2000pro, windows2000server, windows98, windowsme, windowsnt, windowsnt4, windowsxp, and windowsxppro. Note that ettercap attempts to separately identify operating systems that are very similar to each other such as windowsxp and windowsxppro. In addition, ettercap allows generic OS names such as linux even when versions of the same OS are included with specific version numbers. It was also found that the "other" class contains 224 exemplars from 191 different operating systems. Examples of OS names placed in the "other" class are: 3com812adslrouter, acornriscos3.6, amigaos3.1, beos5.0, crayunicos9.0, debian3.0, gauntlet4.0, hplaserjet2100series, hpux11.00, novellnetware4.0, scounixware7.0, suse8.0, ultrixv4.5, and vms. These include rare operating systems with only one or two exemplars, printers, and network devices.

Performance of the baseline ettercap classification approach is poor. This classifier rejects patterns that don't match exemplars exactly and reports the class of the first exemplar that matches exactly as the classification result. The baseline classifier rejected 84% (796/948) of the test exemplars and the error rate for the accepted patterns was 44% (67/152). Two problems contribute to this poor

Table 4. Examples of feature values for ettercap exemplars.

CLASS	WS	TTL	DF	MSS	WSO	SOK	NOP	PS	TS	SYN
windowsxp	14.0	7.0	1	15.49	0	1	1	5.58	0	0
windowsxp	14.0	7.0	1	15.49	0	1	1	5.58	0	0
windows2k	14.0	7.0	1	15.49	0	1	1	5.58	0	0
windows2k	9.9	6.0	1	10.40	1	1	1	5.58	1	0
windows2k	10.0	5.0	0	10.00	0	0	0	5.45	0	1
windows2k	10.4	7.0	1	10.43	0	1	1	5.58	0	1
windows2k	12.5	7.0	1	9.06	0	1	1	5.58	0	1
windows2k	13.0	5.0	1	15.49	0	0	0	5.58	0	0

performance. First, substitution errors occur because exemplars in different classes (e.g. windowsxp and windows2000) have exactly the same feature values. Such errors can only be eliminated by combining classes. This is illustrated by the first three exemplars in Table 4. These three exemplars for the windows2k and the windowsXP class are identical. This table shows log base 2 of the WS, TTL, MSS, and PS features. A second reason for poor performance is the high rejection rate caused by differences between feature values within an OS class. The bottom part of Table 4 shows feature values for the first five exemplars of the windows2k class ordered by the value of the WS feature. All the five exemplars in this table differ in at least one feature.

These examples of exemplars suggest that performance of the baseline classifier could be improved by using a k-nearest-neighbor (KNN) classifier where small differences in feature values would not lead to rejected patterns as in an exact-match classifier. Figure 1 compares the overall error rate for an exact match and a KNN classifier for 10-fold cross-validation testing on all ettercap exemplars and for testing on LL test data. These and following results use a KNN classifier with k=3 instead of 1 to prevent a nearest-neighbor classifier from selecting classes randomly when multiple classes share identical exemplars. Two binomial standard deviations in Figure 1 are roughly 3 percentage points on the left half and 7 on the right. These results demonstrate large reductions in the overall error rate with a KNN classifier for both cross-validation testing and testing on unseen LL test data. They also show the predicted overall error rate for the exact-match baseline classifier using LL test data is roughly 72%. The rejection rate is 60% (107/177) and the substitution error rate is 30% (21/70). The overall error drops to roughly 50% with no rejections using a KNN classifier. These error rates are all too high to create a useful passive OS identification system.

These results suggest that the "fingerprint" concept does not apply to the large numbers of OSes that ettercap attempts to resolve. Even with more than 1000 training exemplars, more than 60% of new unseen exemplars do not match any training exemplar. Confusions for cross-validation testing of the baseline knn classifier, however, suggest that combining OS names into fewer classes could reduce the error rate and still provide useful OS identification.

7. Grouping OS Names into Fewer Classes

Fewer classes were created by combining or eliminating the original 75 classes using the results of a series of eight 10-fold KNN cross-validation experiments and all ettercap exemplars. At each stage after the first, from one to five classes with the highest overall error rate were eliminated or combined with another class. An OS was combined with another if it was confused often with the other OS at that stage (e.g freebsd and macosx; win9x and winnt) or if domain knowledge suggests that the OSes are related (e.g. win95, win98, and win98secondedition). If the error rate was high and could not be reduced by combining classes, then the OS was eliminated and exemplars from that OS were not used in the following experiments.

The first stage of the analysis reduced the number of classes from 75 to 24. This was accomplished by grouping the following into single classes: (1) All versions of HP Laser Jet printers; (2) All devices using any Cisco IOS; (3) All versions of FreeBSD UNIX; (4) All versions of IRIX; (5) Linux 2.2 and 2.3; (5) All versions of MacOS9; (6) Solaris 2.3 through 2.5; (7) All versions of Windows2000; (8) All versions of Windows9; (9) All versions of WindowsNT; (10) All versions of WindowsXP; (11) All versions of Novell Netware; and (12) All versions of SCO UNIX. In addition Redhat, Slackware, Suse, Mandrake, and yellowdog versions of Linux were mapped into the underlying common Linux kernel version 2.2 or 2.4 because this kernel determines how the OS responds to TCP connections. Finally, all classes where no exemplars were identified correctly were eliminated. These class names included aix, hpux, macos7, macos8, openbsd, netbsd, solaris8, and vms.

Figure 2 shows the class names for the seven experiments where the number of classes was reduced

24	19	15	13	12	10	9
baynet	**baynet**					
bsdi	bsdi	bsdi	bsdi	bsdi	bsdi	bsdi
cisco	cisco	cisco	cisco	cisco	**cisco**	
hpjet						
ibmos	**ibmos**					
irix	irix	irix	irix	**irix**		
lexmark	**lexmark**					
linux2.0	linux2.0	linux2.0	linux2.0	linux2.0	linux2.0	linux2.0
linux2.1	linux2.1	linux2.1	linux2.1	linux2.1	linux2.1	linux2.1
linux2.2	linux2.2	linux2.2	linux2.2	linux2.2	linux2.2	linux2.2
linux2.4	linux2.4	linux2.4	linux2.4	linux2.4	linux2.4	linux2.4
macos9	macos9	macos9	macos9	**macos9**		
macosx	macosx	macosx	**macosx**	**mac-bsd**	mac-bsd	mac-bsd
freebsd	freebsd	freebsd	**freebsd**			
netbsd1.4						
netware	**netware**					
solaris2.3	solaris2.3	**solaris2.3**				
solaris2.6	**solaris6-7**	solaris6-7	solaris6-7	solaris6-7	solaris6-7	solaris6-7
solaris7						
win9x	**win9-nt**	**win9-nt**	**win-all**	win-all	win-all	win-all
winnt						
win2k	**win2k-xp**	**win2k-xp**				
winxp						
other	other	other	other	other	other	other

Figure 2. Class names for experiments with from 24 to 9 classes.

from 24 to 9. Bold names indicate when classes were combined or eliminated and brackets indicate where classes were combined. As can be seen, it was not possible to accurately differentiate between all versions of Windows. The number of classes for the Windows OS collapses from four on the left to only one on the right. It was also not possible to differentiate between Solaris 2.6 and Solaris 7 or between MacOSX and FreeBSD. The MacOSX and FreeBSD confusion might be expected because MaxOSX is based on various versions of BSD UNIX including FreeBSD. Confusions between successive versions of different OSes is also expected because kernel code that handles TCP interactions is often not changed between versions.

Figure 3 shows that the cross-validation error rate

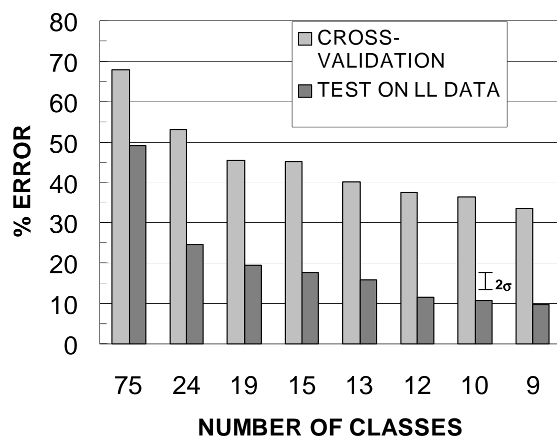

Figure 3. KNN 10-fold cross-validation and on LL test data error for from 75 to 9 classes

measured on ettercap exemplars drops slowly from 68% to 33% wrong as the number of classes is reduced from 75 to 9. It also shows that the generalization error, measured on LL test data, drops from roughly 49% to 10% wrong. The error rate on LL test data is roughly 10% and two binomial standard deviations are roughly 4 percentage points for from 9 to 12 classes. This low error rate is acceptable for many applications of passive OS identification.

A number of factors suggest that the low LL test data error rate might be more representative of the generalization error of this classifier than the ettercap exemplar cross-validation results. Table 5 shows the per-class error rates for the KNN classifier using 10-fold cross validation testing on ettercap exemplars and using the LL test data.

This table shows that with nine classes, the overall cross-validation error rate is 33.4% with cross-validation testing and 9.8% when testing on the LL data. The high cross-validation error rate is not caused by any one class. Per-class error rates on the LL data are much lower than cross-validation error rates except for the "other" class that contained only 9 examples (a disk server and Tektronix printers). For example, the error rate for the win-all class is 25.2% with cross-validation testing and only 9.8% with LL data. The error rate for the solaris6-7 class is 37.3% with cross-validation testing and only 4% with LL data. Both of these differences are well beyond two binomial standard deviations of these per-class error rates.

8. The Effect of Eliminating Ettercap Outliers

The high cross-validation error rates appear to be caused by the method used to collect ettercap exemplars. Documentation on the ettercap web site states "*If a*

fingerprint is not found in the current database it is shown as UNKNOWN and you can contribute to the database by adding the new fingerprint and the respective OS." This suggests that the relative frequency of ettercap exemplars across classes does not represent the true prior distribution of OSes and that the distribution of individual exemplars within a class also is not representative of the underlying prior distribution. In fact, the data collection method might be more prone to collecting misleading outlier exemplars than to collecting representative exemplars. Reasons for outliers could include incorrect labeling of the true OS, network devices between the OS and the network traffic collection site that change packet header contents, or OSes where packet header default values have been changed for security purposes.

Table 5. Nine-class KNN error rate.

	CROSS-VALIDATION		LL TEST DATA	
CLASS	**Exemplars**	**% Err**	**Exemplars**	**% Err**
bsdi	13	23.1%	0	-
linux2.0	22	63.6%	11	27.3%
linux2.1	20	35%	0	-
linux2.2	57	45.6%	0	-
linux2.4	76	46.1%	12	16.7%
mac-bsd	95	28.4%	12	0%
solaris6-7	51	37.3%	99	4.0%
win-all	270	25.2%	41	9.8%
other	153	35.3%	9	55.6%
TOTAL	757	33.4%	184	9.8%

To partially assess the effect of outliers in ettercap exemplars, outliers were detected by comparing only the first three features which prior analysis [15] suggests are important for OS identification. Outliers were eliminated for each class by keeping only exemplars where the first three feature values occurred in at least 1,2, or 3 other exemplars for the same class. This eliminates exemplars where the first three feature values are rare, that are presumably outliers. All features were used to assess performance of a KNN classifier with 9 classes as above. Table 6 shows the results.

Table 6 shows that more than half of the "outlier" ettercap exemplars can be eliminated with no statistically significant increase in the error rate measured on LL data. In addition, as more outlier exemplars are eliminated, the cross-validation error rate drops and comes closer to the error rate measured on LL data. These results suggest that more data should be collected to develop a training set where exemplars are labeled correctly and also where they span a wide range of OSes. They also suggest that ettercap data should be used with caution. The good KNN performance provided with ettercap exemplars despite the inclusion of outliers might be due to: (1) The addition of a new signature for a new OS name, even for only minor differences in names and (2) The addition of a new signature when any feature value in the signature value differs.

Table 6. Nine-class KNN error rates after eliminating different numbers of outlying ettercap exemplars.

Duplicate Matches to Keep	Training Exemplars	Cross-Validation Error	LL-Test Data Error
0 (Normal)	757	33.4%	9.8%
1	560	30.5%	12.5%
2	420	19.1%	10.9%
3	297	8.8%	10.3%

9. Other Classifiers and Rejections

Cross-validation experiments with the 9-class ettercap data were performed to set parameters for other types of classifiers followed by tests with LL test data to measure generalization error. Good performance similar to that obtained with the KNN classifier could be obtained using binary tree classifiers similar to those described in [2], Gaussian kernel support vector machine classifiers [4], and multi-layer perceptron classifiers [9]. All classifiers were versions included in the current version of LNKnet [9]. The binary tree classifier with 40 nodes provided good cross-validation performance that was no more than two standard deviations worse than the error provided by larger trees. A multi-layer-perceptron classifier with 10 inputs, 40 hidden nodes, and 9 output nodes provided good cross-validation performance when trained with stochastic gradient descent training and 50 epochs. It performed better than similar classifiers with from 10 to 60 nodes and with 25 or 75 epochs of training. Finally, a Gaussian kernel support vector machine classifier with a Gaussian sigma of 2.0 and an upper bound on Lagrange multipliers of 5.0 provided better cross-validation performance than linear or quadratic support vector machine classifiers. It also performed better than Gaussian kernel support vector machine classifiers with a sigma of 1.0 or 4.0 or with different upper bounds of 1 or 10. Generalization error rates on LL-data for all classifiers including the KNN classifier (k=3) are shown in Table 7.

Table 7 shows that all four classifiers provide good performance on the LL-data patterns with an error rate of roughly 10%. With the 184 test patterns, two binomial standard deviations is roughly 4.4 percentage points and the spread of error rates without rejection across patterns is less than this range. Cross-validation error rates were much

higher, as with the KNN classifier, and ranged from 27.1% (binary tree) to 35.4% (support vector machine classifier).

Table 7. Nine-class error rate on LL test data for four classifiers.

Classifier	No Rejections		Reject 10% of Input Patterns	
	# Errors	% Error	# Errors	% Error
KNN	18	9.8%	12	7.0%
Binary Tree	16	8.7%	1	0.9%
MLP	23	12.5%	13	8.2%
SVM	20	10.9%	8	5.2%

Table 7 also shows the error rate after 10% of the input test patterns are rejected. Each classifier provides an estimate of the posterior probability for every class. Patterns are rejected when the estimate for the selected class is below a threshold and the threshold is adjusted to reject 10% of the input patterns. As can be seen, the error rate for all classifiers improves when patterns are rejected. Best performance with rejection was provided by the binary tree classifier. It misclassified only 1 out of 165 test patterns. For comparison, the exact-match algorithm used in ettercap has a low substitution error rate of 3% (4/128) but rejects 30% (56/184) of the test patterns.

10. The Effect Of Feature Selection And Using Only SYN Or SYN-ACK Packets

The earliest proposed passive OSID system [13] used only the first three features shown in Table 1 (WS, TTL, DF) and other systems added the remaining features. No experiments, however, analyzed the benefits provided by different feature combinations. Forward-backward feature selection was performed using cross-validation experiments on the 9-class KNN and binary tree classifiers described above. These experiments found the smallest set of features that provided performance no more than two binomial standard deviations worse than the best performance found with any feature subset. With the KNN classifier, five features (WS, TTL, DF, SOK, PS) were selected. With the binary tree classifier only three features (WS, TTL, MSS) were selected. Error rates with the LL test data were then measured after this feature selection was performed. It was found that these error rates were statistically identical (within 1.2 percentage points) to error rates obtained using all features. These result demonstrate that all the features shown in Table 1 are not required for good performance. They show that WS and TTL, which are often suggested as good features, are important, but that other features such as one that indicates whether the input packet was a SYN or SYN-ACK packet are not required.

Some OSID systems such as p0f [14] only analyze SYN packets while others such as ettercap use both SYN and SYN-ACK packets. These two approaches were compared using the 9-class error rate for the KNN classifier. In a new condition, only ettercap SYN packets were used for training and only LL SYN packets were used for testing. This was compared to the normal classifier where both SYN and SYN-ACK packets were used for training and testing. The error rate on LL test data was 6.9% (7/102) for the new condition and 9.85 (18/184) for the normal condition. Both error rates are similar and low. Two binomial standard deviations for these error rates are roughly five percentage points and the difference between these error rates is thus not statistically significant. A second new condition was created where only SYN-ACK packets were used for training and testing. The error rate under this condition was 12.2% (10/82). This error rate is also low. Although it is 5.3 percentage points above the error rate with SYN packets alone, this difference is again not statistically significant and is primarily cause by fewer patterns in the more difficult “other” class with SYN packets. These results suggest that SYN and SYN-ACK packet headers are equally effective when performing passive OS identification.

11. Discussion and Summary

Passive operating system (OS) identification from packet header information is possible, but low error rates can only be obtained using a small number of classes that is much less than the more than 100 different OS names found in the most recent open-source tool. Machine learning evaluations demonstrated that many of the rules-of-thumb used to develop open-source tools may not be valid. For example, best performance is not provided using WS, TTL, and DF features. Other feature combinations provide best performance, especially for the binary tree classifier. Best performance is also not provided using only SYN packets. Similar low error rates are provided with SYN packets, with SYN-ACK packets or with both types of packets. In addition, best performance is not obtained by adding exemplars to a training data only when they differ from existing exemplars. This may lead to the inclusion of many outlier patterns in training data. Better performance would be provided by sampling packet headers generated at many sites from a wide range of OSes and correctly identifying each OS. Finally, the concept of OS “fingerprints” is misleading. Each OS does not have a unique “fingerprint”. Instead, feature values extracted from packet headers vary within and between classes and

machine-learning classifiers can account for this variability.

Experiments led to new nearest neighbor and binary tree classifiers that provide nine-class error rates of roughly 10% without rejection. Other classifiers including multi-layer perceptron and support vector machine classifiers provide similar low error rates. When 10% of the patterns are rejected, the error rate is lowest and near zero with the binary tree classifier. The binary tree classifier requires only three features to provide the lowest error rate (WS, TTL, MSS). The KNN classifier requires five features (WS, TTL, DF, SOK, PS). Nine-class OS identification is accurate and similar for SYN packets alone, for SYN-ACK packets alone, or for both packet types combined in one classifier.

Further work could improve passive OS identification performance by collecting more accurately labeled patterns for classifier development. The frequency of occurrence of OS names in these patterns should reflect the prior distribution of OS names and the frequency of occurrence of patterns in each class should reflect the true distribution of patterns. In addition, information concerning protocols, contents of server banners, email headers, and other content could be extracted to improve passive OS identification in some situations. Finally, when multiple TCP connections are observed for a host, statistical multi-packet features might improve OS identification accuracy. These could include features used in [3] such as averages of features in Table 1 and statistics on how source ports are incremented and TCP sequence numbers are changed between TCP connections. Use of these additional features may make it possible to detect OS masquerading when some feature values are changed to make one OS appear to be another.

Bibliography

[1] Boardman, Bruce, "Middle Managers," Network Computing, February 6 2003, 37-46, http://img.cmpnet.com/nc/1402/graphics/1402f2_file.pdf.

[2] Breiman, L., J.H. Friedman, R.A. Olshen, and C.J. Stone, Classification and Regression Trees, 1984: Belmont, CA: Wadsworth International Group.

[3] Carter, P. and A. Berger, "Passive Operating System Identification: What can be inferred from TCP Syn Packet Headers?" 2002, Presented at the Workshop on Statistical and Machine Learning Techniques in Computer Intrusion Detection, Johns Hopkins University, http://www.mts.jhu.edu/~cidwkshop/Presentations2002.html.

[4] Cristianini, N. and J. Shawe-Taylor, An Introduction to Support Vector Machines, 2000: Cambridge University Press.

[5] Dayioglu, B. and A. Ozgit, "Use of Passive Network Mapping to Enhance Signature Quality of Misuse Network Intrusion Detection Systems," in Proceedings of the Sixteenth International Symposium on Computer and Information Sciences, 2001, http://www.dayioglu.net/publications/iscis2001.pdf.

[6] Fyodor, "Remote OS Detection Via TCP/IP Stack FingerPrinting," 2002, http://www.insecure.org/nmap/nmap-fingerprinting-article.html.

[7] Fyodor, "The Nmap Stealth Port Scanner," 2003, http://www.insecure.org/nmap.

[8] Handley, Mark and Vern Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," Proceedings 10th USENIX Security Symposium, USENIX Association Washington, D.C., 2001, http://www.usenix.org/publications/library/proceedings/sec01/handley.html.

[9] Lippmann, R.P., Kukolich, L., and Singer, E. "LNKnet: Neural Network, Machine-Learning, and Statistical Software for Pattern Classification," Lincoln Laboratory Journal, 6(2) p 249-268, 1993.

[10] Lippmann, Richard, Seth Webster, and Douglas Stetson, "The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection," in A. Wespi, G. Vigna, and L. Deri (Eds.): RAID 2002, Lecture Notes in Computer Science 2516, pp. 307–326, Springer-Verlag Berlin, 2002, http://www.informatik.uni-trier.de/~ley/db/conf/raid/raid2002.html.

[11] Ornaghi, Alberto and Marco Valleri, "Ettercap," 2003, http://ettercap.sourceforge.net/index.php?s=home.

[12] Roesch, M., "Snort - Lightweight Intrusion Detection for Networks," in USENIX 13th Systems Administration Conference - LISA '99. 1999: Seattle, Washington, http://www.usenix.org/publications/library/proceedings/lisa99/roesch.html.

[13] Subterrain Security Group, "Siphon Project," 2000, http://siphon.datanerds.net/.

[14] Stearns, William and Michael Zalewski, "p0f – Passive OS Fingerprinting Tool," 2003, http://www.stearns.org/p0f/.

[15] Spitzner, Lance, "Know Your Enemy: Passive Fingerprinting," 2000, http://project.honeynet.org/papers/finger/.

Boundary Detection in Tokenizing Network Application Payload for Anomaly Detection

Rachna Vargiya and Philip Chan
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901
rvargiya@fit.edu and pkc@cs.fit.edu

Abstract

Most of the current anomaly detection methods for network traffic rely on the packet header for studying network traffic behavior. We believe that significant information lies in the payload of the packet and hence it is important to model the payload as well. Since many protocols exist and new protocols are frequently introduced, parsing the payload based on the protocol specification is time-consuming. Instead of relying on the specification, we propose four different characteristics of streams of bytes, which can help us to develop algorithms for parsing the payload into tokens. We feed the extracted tokens from the payload to an anomaly detection algorithm. Our empirical results indicated that our parsing techniques can extract tokens that can improve the detection rate.

1. Introduction

Motivation: Traditional intrusion detection systems use *misuse/signature detection*, which models known attacks, and generally cannot detect novel attacks. *Anomaly detection* models normalcy and identifies deviations, which potentially can be novel attacks. During training, network anomaly detection models the normal patterns of network traffic. During detection, scores are assigned to anomalous events and significant anomalies cause alerts indicating possible attacks. Existing anomaly detection techniques usually rely on information derived only from the packet headers; however, this is not sufficient since more sophisticated attacks involve the application payload. Parsing packet headers is relatively simple as there are few commonly used protocols such as IP, TCP, UDP, and ICMP. However, for application payloads, parsing is more challenging due to the large number of application protocols available and relatively frequent introduction of new protocols. *Hard coding* the parser for each application protocol could be time consuming, particularly when the protocols are complicated. Furthermore, updates to existing protocols or introduction of new protocols will require additional efforts.

Problem statement: We desire to parse application payload into tokens without explicit knowledge of the application protocols. Given a set of exemplar payloads, an algorithm learns a model that can parse the payloads into "meaningful" tokens. Furthermore, the algorithm needs to be independent of the protocols. The extracted tokens can then be used as attributes for modeling normal traffic for anomaly detection (the same techniques can also be used to identify tokens for misuse detection as well, but anomaly detection is the focus of this paper).

Approach: We propose four characteristics of relevant tokens in a continuous stream of bytes, and based on them, design algorithms that propose possible boundaries for tokens. We use these characteristics individually and in combination to estimated boundaries. The sequence of bytes between the two successive boundaries is considered a token which can be used to model the behavior of the payload. The characteristics are based on Boundary Entropy, Frequency, Augmented Expected Mutual Information, and Minimum Description Length. These characteristics do not depend on any particular property of a protocol.

Contributions:

- We describe four algorithms based on the characteristics mentioned above, and apply them to parse the payload to extract tokens from network traffic.
- We also explore techniques using more than one such characteristic in combination.
- We discuss four evaluation techniques to evaluate such tokens.
- We demonstrate that the tokens found by our algorithm can improve the detection rate of the LERAD anomaly detection algorithm.
- Our algorithms would work on encrypted data as well, since they are domain independent.

Organization: The next section, Section 2, discusses the related work. Section 3 details the four characteristics and the associated algorithms. In section 4 we discuss results obtained from our experimental evaluation, and finally we conclude in Section 5.

2. Related work

A variety of approaches have been adopted for the boundary detection problem. Some of them are unique and achieve interesting results.

One of the early studies include that described in Forrest et al. [1] They used fixed length patterns to represent the process model and used it for intrusion detection purpose. However, a main limitation of this approach is that there is no rationale for selecting the optimal pattern length, which has a major influence on the detection capabilities of the intrusion detection system. In addition, it uses fixed length patterns, which makes it a difficult task to select the optimal pattern length. Long patterns are expected to be more process specific than short patterns. Our approach is independent of such a parameter like length and hence overcomes this problem.

Wespi et al. [14] use Teiresias algorithm [12] in combination with a pattern reduction algorithm to construct patterns. All maximal variable length patterns contained in the set of training sequences are determined and a reduction algorithm is applied to prune the entries in the pattern table. Their pattern-matching algorithm returns the groups of consecutive uncovered events and the length of each of these groups. The greater the length, the more likely it is that an intrusion is observed.

Liao et al. [6] use a k-Nearest Neighbor classifier to characterize program behavior as normal or intrusive depending on the short sequences of system calls. Even though the computation required for this technique is reduced, it is unable to detect attacks that consist of abuse of a legal attack, e.g. Process table attack. Some text categorization work is also done by Dumais et. al [3].

Jiang et al. [5] consider both Intra pattern and Inter pattern anomalies. They provide a pattern extraction algorithm to identify maximal patterns. Then they use a Pattern overlap relationship module where adjacency lists are formed from patterns in which overlap relationship between patterns is stored. Pattern adjacency lists are then traversed at real time to identify both intra pattern and inter pattern mismatches. Significant deviations from the normal behavior cause the module to raise alerts.

Valdes [13] proposes a system that maintains a library of patterns that may be initially empty. When a pattern is observed, its similarity with respect to other patterns in the library is observed. If it matches one or more stored patterns above a configurable threshold, then the new pattern is considered to belong to the class of the best matching. However, their approach works only with a alphabet size and small number of actual observed patterns.

Michael [11] uses suffix trees of a fixed height to find frequent occurring sequences of system calls. Very frequent sequences are replaced with meta-symbols, resulting in a more compact representation of the system calls. Based on the revised vocabulary, a regular language is learned to represent the normal behavior of system call traces. One of the algorithms, SEQUITUR proposed by Manning and Witten [10] provides a technique for parsing the text, which is our first step. It is based on the principle that phrases, which appear more than once, can be replaced by a grammatical rule that generates that phrase. The rule generated is different from conventional grammar since the rules are not generalized and they generate only one token.

Another algorithm proposed by Cohen et al. [2], called VOTING EXPERTS consists of experts that evaluate the features of the episodes, namely Boundary Entropy and Frequency, and votes for boundaries in the corpus based on these features. A window is passed through the corpus and each location garners 0 or 1 vote from each expert. The location with the least boundary entropy and highest

frequency receive votes from the two experts respectively. The drawback of their technique is that their votes are binary; the confidence in a particular boundary cannot be indicated.

3. Approach

Our approach consists of parsing the payload and extracting tokens providing some information about the payload, and using these tokens to model the network behavior for anomaly detection. This approach demands a good algorithm to parse the payload. There are characteristics that can categorize bytes belonging to some relevant token. Hence, these characteristics can be exploited to detect boundaries in a continuous stream of characters. By extracting the token between two boundaries, we can derive a set of bytes belonging to the same token.

Our approach is inspired by VOTING EXPERTS [3] as in features are used to detect boundaries. However, there are many differences in the details of the approaches. Not only have we created more experts for casting votes and combined those experts, we also intend to make a system that allows certain feature to cast multiple votes, depending on how strongly that feature believes that a boundary exists at that location. In VOTING EXPERTS, each feature was independent and could cast only binary votes, whether the feature strongly suggested a boundary or there was just a slight indication of the same. Since the features we use to assess potential boundaries are statistical, our approach is independent of the language or in our case, independent of the protocol of the application layer. Hence, our technique is domain independent. Two sample records from port #21 are:

^@USER anonymous^M^ ^JPASS chiaraa@delta.peach.mil^M^ ^JSYST^M^ ^JPORT 194,7,248,153,4,241^M^ ^JLIST^M^ ^JCWD mailing_list^M^ ^JPORT 194,7,248,153,4,242^M^ ^JLIST^M^ ^JCWD archive^M^ ^JPORT 194,7,248,153,4,243^M^ ^JLIST^M^ ^JCWD music^M^ ^JPORT 194,7,248,153,4,244^M^ ^JLIST^M^ ^JTYPE I^M^ ^JPORT 194,7,248,153,4,245^M^ ^JRETR 0016.html^M^ ^JQUIT^M^ ^J

^@GET anonymous^M^ ^JPASS pablot@delta.peach.mil^M^ ^JSYST^M^ ^JPORT 194,7,248,153,4,255^M^ ^JLIST^M^ ^JCWD man^M^ ^JPORT 194,7,248,153,5,0^M^ ^JLIST^M^ ^JCWD man3^M^ ^JPORT 194,7,248,153,5,1^M^ ^JLIST^M^ ^JTYPE I^M^ ^JPORT 194,7,248,153,5,2^M^ ^JRETR cpp.1^M^ ^JQUIT^M^ ^J

The first record shows a normal connection record for port #21. However, the second connection record shows an anomaly. The first keyword in a FTP connection record is usually "USER". The keyword "GET" is inappropriate and suggests malicious data.

The general working of the algorithm includes a window of arbitrary size (given as an input), say *w*, which is slid through the corpus to be segmented. At each instant, *w* bytes from the corpus are observed. Each feature evaluates the value for each possible boundary within the window, and decides whether the value is good enough for a boundary or not. If yes, a vote is cast, otherwise the window simply slides one character forward, examining again the token of length *w*, differing in one byte from the previous token. Two parses are required for this approach on the corpus, first to evaluate the feature value for each possible token, second to compare the various possible boundary locations based on the evaluated feature value and to assign votes.

There are four features used to cast votes in our model. Two of them are similar to the ones in VOTING EXPERTS: Boundary Entropy and Frequency. The other two are Augmented Expected Mutual Information (AEMI) and Minimum Description length. Finally, we propose techniques by combining some or all of them. Each one is discussed in some detail below. We note again that we adopt weighted voting according to the confidence of each expert, which is different from the original VOTING EXPERTS.

3.1 Boundary Entropy

The entropy in patterns exhibits a trend that is exploited in this technique. It starts with a relatively high value, then drops as we go further, and peaks at the end of a valid word. This is because entropy gives us the uncertainty or degree of randomness in a system. When we see the first few characters of a word, it is difficult to predict what the word is. E.g., given the character 'W', it is difficult to say what the word is. It could be 'What', 'Where' or any other such word. Hence, the entropy after 'W' is relatively high. However, as we move further, the uncertainty drops. E.g., if we have the token 'Wha' then we know that the word probably is 'What'. At the end of the meaningful token, the entropy peaks. This is because it becomes very uncertain what the following word is going to be, and hence what the next character should be. In our example, any word could follow 'What', so it is difficult to say what the next character will be.

We exploit this property to create an expert to detect boundaries. The entropy at each possible location is

calculated using the formula, similar to voting experts, i.e.

$$-\sum P(x)\log P(x) \quad (1)$$

$P(x)$ is the probability of the byte x following the current window. (More precisely, $P(x)$ is actually $P(x|s)$, where s is the sequence of bytes in the window of size w.) The entire expression gives us the uncertainty of the byte following the token in the window. During the first parse, the window is moved across the file and the byte following the window is noted and $P(x)$ is estimated. In the second parse, the window is moved again and Boundary entropy at the end of each window is calculated using the formula in Equation (1).

The positions that have the maximum entropy get a vote from the expert. However, since we desire to signal only boundaries with reasonable confidence, we introduce a threshold that suppresses votes from boundaries with low entropy values. We use the average boundary entropy of the corpus as the threshold. To allow fair voting among experts, the boundary entropies are normalized before votes are cast. The votes cast are proportional to the number of standard deviations away from the mean value.

3.2 Frequency

The second method computes the frequency of each token that occurs in the corpus. The most frequent set of tokens are assumed to be valid tokens and boundaries are assigned at the ends of such tokens. As the window moves forward, the frequency of each possible token of length 1 to the window size, within the window, is calculated. E.g. if the window consists of "examsare", then the frequency of 'e', 'ex', 'exa' and so on is calculated. Boundary is voted at the end of the most frequent token. The votes given are proportional to the number of standard deviations that the frequency of the token is away from the mean.

The window is moved through the corpus and each token formed is counted. Hence, at the end of the parse, we have a list of all possible words, which the window may consist of, and their frequency. Generally, in most domains, there is a relationship between the length and frequency of patterns. Short patterns tend to be more common than the long ones. E.g.,'t' would be more common than 'the' even though 'the' is a valid word and 't' is not. We want to compare how unusual a pattern is, not just how frequent it is. Therefore, comparing the frequencies of short patterns with that of long patterns would not be appropriate. To accommodate this, we normalize the frequencies of the tokens for tokens of the same length. We subtract the sample mean from the value and divide by the sample standard deviation.

3.3 Augmented Expected Mutual Information (AEMI)

A lot of information can be gathered about a character based on the context it appears in. Generally, the concept of mutual information is used to evaluate the relationship between two events. Mutual Information can estimate the likelihood of the occurrence of a token given some other token. E.g. talking about food, given that we have seen 'POP' it is very likely that the next word would be 'CORN'. Hence, this approach is based on co-occurrence of tokens: if two tokens appear together frequently, they are probably part of the same word. Mutual information is given by:

$$MI(a,b) = \lg[P(a,b)/(P(a)P(b))] \quad (2)$$

In other words, MI gives us the reduction of uncertainty in presence of 'b' in the window if presence of 'a' is known (or vice versa). However, this metric only considers the presence of both the words but not the absence of either of them. That is, it does not consider what the probability of seeing one token in the absence of the other. This leads to misinterpretations if the token whose occurrence is being measured is highly frequent. E.g., we would expect that 'pop-corn' is more correlated than 'is in', however since 'is' is relatively more common. This would lead to a high MI value. The presence of one token without the other one counts for adverse correlation and proves to be counter evidence. Augmented Expected Mutual Information [1] incorporates the idea of independent existence of 'a' and 'b' as well, which appropriately incorporates the counter evidence. It is defined as:

$$AEMI(A,B) = P(a,b)MI(a,b) - P(\neg a,b)MI(\neg a,b) - P(a,\neg b)MI(a,\neg b) \quad (3)$$

Equation 3 sums the supporting evidence and subtracts the counter evidence. a is defined as the

event of the first token, and *b* is the event of the token following the first one. Higher values of AEMI indicate that *a* and *b* are probably part of the same word. We only consider three cases with each pair of tokens, occurrences when both the tokens appear together, when token *a* appears without token *b*, and token *b* appears without token *a*. The case when neither *a* nor *b* appears is disregarded since it does not really present much information about whether *a* and *b* co-occur or not. The window is again moved across for each possible boundary, the left and right sub tokens are considered. E.g. If a window contains "abcdef" we consider left and right sub tokens 'a' and 'bcdef', then 'ab' and 'cdef', then 'abc' and 'def' and so on. Then for each set of left and right sub tokens within a window, AEMI value is computed and compared. For each window, the location with the minimum AEMI value suggests a boundary, and the expert gives votes proportional to the standard deviations from the average AEMI.

3.4 Minimum Description Length

In coding theory, tokens that are more frequent are assigned a shorter code so that the overall coding length is minimized for a message with multiple tokens. Minimum Description Length (MDL) assumes a perfect encoding and measures the fewest number of bits necessary to encode a message. We calculate the description length per byte of a token by:

$$MDL = \sum_{i \in \{left, right\}} - \lg P(t_i) / | t_i | \qquad (4)$$

Where t_i denotes the two tokens on the left and right of the possible boundaries, P (t_i) is the probability of t_i and $| t_i |$ is the length of t_i in bytes. The assumption is that if we were to compress the file, we would assign minimum number of bits to the most frequently occurring token; hence, it would have the minimum length. –lg [P (t_i)] which gives us the number of bits used for t_i, dividing it by the length of t_i, $| t_i |$, which, gives us the number of bits per byte of the token or its description length.

The preprocessing is similar to that in AEMI. The first parse is used to look at all the left and right sub tokens and compute the probability of seeing those two tokens. This probability is then used in the second phase, which computes the sum of the description lengths of left and right sub tokens. As the window slides over the data, the boundary that yields the shortest coding length is voted as the boundary and the number of votes is again proportional to the number of standard deviations from the average coding length.

3.5 Combined Approach with Weighted Voting

All the approaches discussed so far use a single expert to suggest boundaries. We desire to design a model that combines the opinions from all the experts and then decides upon the boundary. In this method, we allow each expert to run a scan on the file and decide where to vote, and how much to vote. Votes from each expert are normalized, as some approaches may tend to assign more votes than the others do. These votes are then combined to locate positions most strongly suggested as the boundary after consideration by all the experts. A list of votes from all the experts is gathered. This list is normalized so that the votes from each expert indicate the confidence of the expert and are on the same scale. In order to normalize the list, the standard deviation of the votes is computed and each value in the list is divided by the standard deviation. This scales the value with respect to the other values in the list. In order to give more weight to a particular expert, the votes from this expert can be increased by a certain factor. For each boundary, the final votes from each expert, after normalization, are summed. A threshold is set computed depending on the final set of votes. Once again, it is the average of the votes assigned to each boundary. A boundary is placed at a certain position if and only if the votes at that position exceed the threshold.

We tried combining all the algorithms and then combining the strongest algorithms, frequency and minimum description length.

3.6 Anomaly Detection

Once we have placed the boundaries according to our experts, we can easily extract the meaningful tokens from the file. Our anomaly detection system, LERAD [9], forms rules based on attributes picked from the network data (including header and payload). Currently, it uses tokens from the payload that are "space" separated. Instead, we modify it to use tokens that are separated by boundaries identified by the algorithms discussed above.

4. Experimental Evaluation

4.1 Evaluation Criteria

We present four different types of evaluations depending on various attributes that would indicate the “meaningfulness” of the tokens retrieved in the output file.

The first evaluation, Evaluation A, is based on how many words, present in the input file, were we able to retrieve in the output file with the boundaries placed at positions suggested by the expert. All space or punctuation separated tokens are assumed to be “meaningful” tokens. This evaluation works for text-based protocols only. It doesn’t work for non-text based protocols because bytes that represent “spaces” usually do not exist. Moreover, Evaluation A only approximates how "tokens" are defined. E.g. a file name could consist of ‘/’ to denote the path of the file. The entire path should be considered as one token, however since our evaluation would consider ‘/’ as space, it will consider each directory a unique token. Also, for hyphenated words, even though they would logically be the same token, this evaluation would evaluate them as being separate tokens. Based on the space-separated words, we report the percentage of words recovered by our methods.

The second evaluation, Evaluation B, is similar to the first evaluation except that it looks for certain keywords that are characteristic of the particular application protocol. These keywords are collected from the specification of each protocol (Request for Comments or RFC). However, this evaluation is limited to text-based protocols for the reasons mentioned above and is an approximation since tokens between two keywords are not specified. Based on the known keywords, we report the percentage of keywords recovered by our methods.

The third evaluation, Evaluation C, calculates the entropies of the output files. The motivation for this evaluation is that if the expert was successful and it found most of meaningful tokens, then the tokens should be repeated often in the output file, leading to less randomness in the output file and therefore to lower entropy values for the file. Thus, in our evaluation, the lower the entropy value of the output file, the better is the feature or expert. This evaluation is independent of any text-based assumptions and hence can be used for all kinds of ports. It gives a good estimate of the output file. It can be used to compare the performance of an approach on any kind of protocol.

The fourth evaluation, Evaluation D, is the detection rate evaluation, which is the most important evaluation, while Evaluations A and C are intermediate approximations. We measure the number of detections at various false alarm rates and compare the performance of the original LERAD with LERAD using tokens extracted by our proposed methods.

4.2 Evaluation Data and Procedures

The proposed methods were evaluated using the 1999 DARPA Intrusion Detection Evaluation Data Set [7]. The test bed involved a simulation of an air force base that has machines that are under frequent attack. These machines comprise of Linux, SunOS, Sun Solaris and Windows NT. Various intrusion detection systems have been evaluated using this test bed. It comprises of three weeks of training data obtained from network sniffers, audit logs, nightly file system dumps and BSM logs from Solaris machine that trace system calls and two weeks of testing data. Weeks 1 and 3 of the data are attack free while various attacks are present in Weeks 4 and 5 of the data.

To our knowledge, the DARPA 99 data set is the most comprehensive publicly available data set for evaluation of intrusion detection. We are aware of the simulation artifacts in the data set as discussed in [8].The focus of this paper is comparing tokenization techniques for extracting features to enrich the representation of the training dataset for anomaly detection algorithms. We are not comparing anomaly detection algorithms and fixed the algorithm to be LERAD. We plan to extend our investigation to datasets that contain collected traffic from real-life networks.

For our first three evaluations, where we compute the number of words retrieved, number of keywords retrieved, and the entropy of the output file, we use only the data from Week 3. The reason for using week 3 for evaluations A, B, and C is that Weeks 4 and 5 are for testing only and we do not want to have the advance knowledge of which tokenization methods work better in Weeks 4 and 5. Weeks 4 and 5 contain an evidence of 146 simulated attacks. However these attacks are across all the ports. We have tested only

some of the ports from the entire data. We used the first four days of Week 3 for training and the last three days for testing. This gives us an estimate of how predictive the approaches are, and how well they would perform on unseen data in the network traffic. We studied the ports with the most traffic and results from these ports are reported. The window size was a parameter set to six, which was experimentally observed to be the best value.

The anomaly detection system LERAD [9] works in three phases. In the first phase, it samples training pairs to suggest rules. In the second and third phases, it removes redundant rules and rules that generate alarms on attack free traffic respectively. LERAD learns rules based on 23 attributes taken from the TCP header and the payload. First 15 attributes are picked from the packet header and the remaining eight are picked from the payload. LERAD, originally picks the first eight space separated tokens from the payload-- space as boundary is not applicable to non-text protocols. We replace these eight space separated tokens with the more intelligently found boundary separated words from our approaches. For Evaluation D, we use Week 3 for training and Weeks 4 and 5 for testing.

4.3 Experimental Results and analysis

For each evaluation criterion, we compare results from six different approaches, four approaches being the results of the four algorithms independently, fifth being the combination of all the algorithms and sixth being the combination of two of the strongest algorithms, Frequency and MDL. The reason for combining Frequency and MDL is that, from our experience with this data set, they provide maximum coverage and complement each other.

4.3.1 Evaluation A: Space Separated Tokens

Table 4.3.1 reports the results of all the approaches on popular ports with text-based protocols, SMTP (25), HTTP (80), FTP (21) and Finger (79), based on Evaluation A. For all these ports, Boundary Entropy gives the poorest results. Frequency performs the best for SMTP and Finger, however Freq + MDL performs best for HTTP and FTP. On qualitative analysis, Freq + MDL seems to give a more consistent output with long relevant tokens. Hence, we suggest that Freq+MDL together give the best results followed by the single approach of Frequency. The model of all the algorithms combined follows these two techniques. MDL performs better than Frequency when trained and tested on the same set; however it is not very predictive. Frequency on the other hand, is highly predictive. Hence, these two algorithms tend to find different kinds of words. When combined they give maximum coverage and hence best results.

Table 4.3.1 Evaluation A: % of Space-Separated Tokens Recovered

Method	Port #25	Port #80	Port #21	Port #79
Frequency	15	16	13	99
Min Desc. Length	6	7	3	25
AEMI	5	9	4	32
Boundary Entropy	3	3	1	9
All 4 experts	21	14	5	12
Freq + MDL	52	26	21	81

4.3.2 Evaluation B: Keywords in RFCs

Table 4.3.2 Evaluation B: % of Keywords in RFCs Recovered

Method	Port #25	Port #80	Port #21
Frequency	31	28	40
Min Desc. Length	7	6	1
AEMI	9	5	2
Boundary Entropy	3	2	2
All 4 experts	12	13	21
Freq + MDL	40	36	59

Table 4.3.2 reports the results for all the methods based on Evaluation B. Results for port #79 are absent since no keywords were available for port #79. Here again Frequency + MDL performs the best for ports #80 and #21. The ranking of the algorithms remains

the same and reinforces our conclusions from the previous evaluation.

4.3.3 Evaluation C: Entropy

Table 4.3.3 Evaluation C: Entropy of Output

Method	Port #25	Port #80	Port #21	Port #79	Port #1023	Port #22
Frequency	9.19	5.11	5.17	3.78	0.86	5.79
Min Desc. Length	8.61	5.26	5.50	1.43	0.77	8.61
AEMI	8.66	5.74	9.23	6.27	1.10	7.32
Boundary Entropy	7.89	5.36	6.79	2.63	0.96	7.75
All 4 experts	9.52	5.07	5.36	6.32	1.39	5.74
Freq + MDL	7.94	4.98	9.04	4.31	1.91	8.32

Table 4.3.3 reports the results of all the approaches based on Evaluation C on four text based and two non text based ports, Smtp (25), Http (80), Ftp (21), Finger (79), SSH (22), and TCP Reserved (1023). This evaluation compares the schemes on both text based as well as non-text based ports and allows us to compare the techniques without any bias. The relative values vary for different ports. For port #25 Boundary Entropy gives the best results, however for ports #79 and #1023, Minimum Description Length gives lowest entropy. Frequency gives lowest entropy values for port #21, Freq + MDL and the combination of all four methods achieve the lowest entropy for #80 and #22 respectively. Since all techniques are very close in this evaluation, it is difficult to say which technique is best for all ports based on this evaluation only. However we can make port specific conclusions like for port #80, Freq + MDL is the best technique.

4.3.4 Evaluations on Combined models

Since Frequency + MDL and the model combined of all algorithms have the potential of giving better boundaries indicated by evaluations A-B and evaluation C respectively, we performed experiments on the remaining ports with these two techniques.

Table 4.3.4 Results from Additional Ports for Freq + MDL and ALL

Port #	Evaluation A % Words Found		Evaluation B % Keywords Found		Evaluation C Entropy	
	Frq + MDL	ALL	Frq + MDL	ALL	Frq + MDL	ALL
23	13	7	5	3	7.88	8.08
113	43	20	--	--	4.45	5.18
515	38	14	--	--	7.66	7.27

Table 4.3.4 reports results of the two models, Frequency + MDL and the combination of all algorithms on additional ports, for all three evaluations. Based on these results, it is evident that Frq+MDL performs better than the model combining all four approaches. Even though Frq+MDL performs very well, the inclusion of the other two techniques weakens the model. This could be attributed to the probability that with the inclusion of AEMI and BE the model gets confused and results deteriorate. Boundary Entropy in particular attempts to vote at too many positions and lowers the performance.

4.3.4 Evaluation D: Detection Rate

Table 4.3.5 Detection Rate for Space Separated LERAD and Boundary Separated LERAD using Freq + MDL tokenization

PORT#	10 FP/day		100 FP/day	
	Space-Separated	Boundary-Separated	Space-Separated	Boundary-Separated
20	2	2	4	5
21	14	16	14	17
22	3	3	3	3
23	13	14	13	14
25	15	16	16	16
79	3	3	3	3
80	10	10	11	13
113	2	2	2	2
Overall	59	62	63	68

Based on our first three evaluations, we picked the most promising technique for our fourth and most important evaluation. From the previous evaluations, it was obvious that certain techniques may be better depending on the port. However, the model consisting of Frequency and Minimum Description Length gave a good performance consistently. Thus, we decided to perform our final evaluation on this technique.

LERAD forms conditional rules that are used to test tuples from test data. The alarms generated were evaluated for two different allowed false alarm rates – 10 and 100 per day respectively. The results, reported in *Table 4.3.5,* indicate some improvements in the total number of detections for both text based and non-text based protocols. Port #20 shows an improvement of one detection when the false alarm rate is set to 100 per day. Considerable improvement for port #s 21, 23 is observed for both false alarm rates. Port #25 and #80 also show an improvement of one attack each at false alarm rates of 10 per day and 100 per day respectively. For other ports where the results are comparable, we suggest two possible reasons. Firstly, the training data for these ports was not sufficient for the experts to cast vote during the testing phase. In addition, for certain ports, it never generated any rules based on the tokens from the payload—LERAD did not find the payload tokens to be indicative of normal behavior. In such cases, even if tokens that are more meaningful were extracted by our algorithms, the results would not be affected.

We also present a set of "Overall" results which indicate the total number of attacks detected over all the ports, excluding the duplicate detections. This means that there are several attacks which occur across ports, multiple detections of the same attack in different ports are discarded. Even the overall detection shows an improvement of 3 attacks when false alarm rate is set to 10; and an improvement of 5 when the false alarm rate is set to 100. That is an improvement of 5% and 8% respectively.

We also performed experiments using a combined model of all the ports instead of using port specific data. Even then LERAD with space-separated tokens finds 36 attacks in week 4 data as compared to 38 attacks detected if boundary separated tokens are considered. The false alarm rate was 10 per day for this result. On increasing this rate to 100 per day, the former still detects 36 attacks while the latter detects 39. Data for week 5 was not used for these results and experiments are still being conducted to evaluate this technique on week 5 data of the DARPA data set.

5. Concluding Remarks

In this paper, we present the four algorithms based on characteristics mentioned above, and apply them to parse the payload to extract more information about the traffic. The results of each of those techniques applied independently and then applied in various combinations based on these evaluations are given. According to the experimental results obtained from the DARPA 99 dataset, we observed that Frequency and MDL are two strong experts individually and achieve good results. MDL works even better when training and testing sets are more similar. Frequency is highly predictive and does well on different training and testing sets. When combined, the model formed by combining Frequency and MDL is found to be the strongest. Combining all four methods does not do as well as Frequency + MDL. This payload parsing method, when applied to the LERAD anomaly detection algorithm, leads to an increase in the detection rate in two configurations: individual LERAD model per port or single LERAD model for all ports. The overall detection rate also showed a significant improvement of 5% and 8% at the rate of 10 and 100 False-alarms/day respectively. One significant contribution we would like to bring forth is that we have made use of information from the payload while most IDS concentrate on the header information. Also our payload parsing technique is such that it can be applied to any protocol. Our parsing techniques also use weighted voting, which is different from the original VOTING EXPERTS.

Our goal is to use these approaches to improve the features used by the anomaly detection algorithm LERAD [7]. One may also point that payloads may be encrypted. However, the payload has to be decrypted somewhere; that is, the detection algorithm can be placed after the payload is decrypted.

Our algorithm is offline. Adapting to protocol changes would require retraining, but retraining is far less labor intensive than changing hand-coded parser. Furthermore, our offline algorithm can be applied in a semi-online manner. For example, learn a model using data from one day, and then learn a model using data from two days and so on. That is, the model is updated each day.

The algorithms do show improvement over the original LERAD. Moreover, the techniques are subject to further investigation that can improve the results further. Another improvement can be made by using the tokens which are likely to give maximum information instead of the first eight boundary separated tokens. This property of the tokens can be measured by again looking at features like frequency, AEMI and so on. Of the words that are retrieved in the output, the ones with maximum feature value are likely to give us maximum information. We will also try to integrate our technique, i.e. incorporating information from the payload, to more intrusion detection systems. We will also be apply the technique to real data in the near future.

6. Acknowledgement

This work is partially funded by DARPA (F30602-00-1-0603). We thank the LLR members for their help on ideas and the anonymous reviewers for their comments.

References

[1] Philip K. Chan, Constructing web user profiles: A non-invasive learning approach, In *Web Usage Analysis and User Profiling*, LNAI 1836, Springer-Verlag, p39-55, 2000.

[2] Paul Cohen, Brent Heeringa, and Niall Adams. An unsupervised algorithm for segmenting categorical time series into episodes, *IEEE International Conference on Data Mining,* 2002.

[3] Susan Dumais, John Platt, David Heckerman, Inductive Learning Algorithms and Representations for Text Categorization, In *Proc. of ACM-CIKM98,* 1998.

[4] Steven A. Hofmeyr, Stephanie Forrest and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security,* 1998.

[5] Jiang N., Hua K., and Sheu S. Considering Both Intra-pattern and Inter-pattern Anomalies in Intrusion Detection, In *Proc. of International Conference on Data Mining,* 2002.

[6] Yihua Liao, V. Rao Vemuri. Using text categorization techniques for intrusion detection, In *Proc. 11th USENIX Security Symposium,* 2002.

[7] R. Lippmann, J. Haines, D. Fried, J. Korba & K. Das. The 1999 DARPA Off-Line Intrusion Detection Evaluation, *Computer Networks*, 34(4), p579-595, 2000.

[8] Matthew V. Mahoney, Philip K. Chan, An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection, *Proc. 6th Intl. Symp. Recent Advances in Intrusion Detection*, p. 220-237, 2003.

[9] Matthew V. Mahoney, Philip K. Chan, Learning Models of Network Traffic for Detecting Novel Attacks, In *Proc. of the Third IEEE International Conference on Data Mining, 2003 (to appear).*

[10] Nevill Manning, C.G., Witten, I.H. Identifying hierarchical structure in sequences: A linear time algorithm, *Journal of Artificial Intelligence Research*, 7, 67-82.

[11] C. C. Michael, Finding the Vocabulary of Program Behavior Data for Anomaly Detection, In *DISCEX,* 2003.

[12] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences, *Bioinformatics, 14(1):55–67*, 1998.

[13] Alfonso Valdes, Detecting Novel Scans Through Pattern Anomaly Detection, In *Proc. DISCEX, 2003*

[14] Andreas Wespi, Marc Dacier, and Herve Debar. Intrusion detection using variable length audit trail patterns, In *Proc. of Recent Advances in Intrusion Detection,* 2000.

Detecting Privilege-Escalating Executable Exploits

Jesse C. Rabek, Robert K. Cunningham, and Roger I. Khazan
MIT Lincoln Laboratory
rkc@ll.mit.edu, rkh@ll.mit.edu

Abstract[1]

The Lincoln Laboratory Malicious Code Detector (LIMACODE) is a system for statically detecting privilege-escalating exploits in data streams, such as files and network traffic. LIMACODE operates as follows: it scans data streams, identifies the language of the stream, then extracts language-specific features for input to a feed-forward neural network classifier which labels the stream as either malicious or benign. LIMACODE is designed to be a relatively lightweight system that can classify a large number of streams quickly so as to be deployed at sites where new data streams (e.g., software) appear frequently. This paper describes a part of LIMACODE that detects privilege-escalating exploits embedded in UNIX Executable and Linking Format (ELF) files; the detectors for C and shell code exploits were described earlier elsewhere.

1. Introduction

There are many routes an attacker may take when attempting to compromise a computer system, including employing social engineering, exploiting a vulnerability in a network or local service, or tampering with a physically accessible computer system. However, the damage caused, information gained, resources obtained, etc. is limited by the privileges held by the attacker. On UNIX-based systems, such privileges allow a user to access only those resources that have been specifically granted to the user, to the user's groups, or to the programs that the user is allowed to use [1].

Often attackers attempt to increase the privileges that they hold in order to obtain access to resources that are otherwise unavailable to them. In order to increase their privileges, attackers must either cause the operating system to grant them unauthorized privileges or cause a process with a different set of privileges to perform unauthorized actions on their behalf. Unauthorized privileges can be granted if a system is configured in an insecure manner or if users select weak passwords; attacks against these vulnerability classes are not the focus of this paper. Unauthorized actions can be performed if an attacker can inject code and cause the privileged process to run it. Our focus is this latter case. Specifically, we wish to detect exploits used to perform code injection via out-of-bounds writes, also known as buffer overflows. Buffer overflow vulnerabilities have been and remain one of the most common: between July 2002 and July 2003, 231 out of 712 (approximately 32%) of high severity vulnerabilities published by NIST were from buffer overflows [2].

This paper describes The Lincoln Laboratory Malicious Code Detector (LIMACODE). LIMACODE can detect privilege-escalating C, shell, and executable code. Detectors for source code analysis of attacks that exploit buffer overflows and time-of-check-to-time-of-use errors [3] in privilege escalating C and Shell code are described elsewhere [4]. This paper describes a detector for attacks that exploit buffer overflows in privilege escalating code appearing in Executable and Linking Format (ELF) files compiled for the x86 architecture. ELF is the most common binary executable format used in Linux and Solaris OSs, and x86 processors are the most prevalent. However, there is nothing inherent in our approach that would prevent it from being used on other file system formats, operating systems, or architectures.

2. Background and Related Work

Buffer overflow attacks can be detected using dynamic or static analysis. Dynamic analysis monitors software that is executing, and therefore requires an appropriate environment for running the exploit and vulnerable software, and obtaining information about their interactions. The requisite environment includes the correct operating system, libraries and external programs and an audit system to report on the executing software. In contrast, static techniques examine code without running it. Such techniques are useful in a network in which the ingress of all software is to be monitored, but

[1] This research was sponsored by the Defense Advanced Research Project Agency (DARPA) under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

the software is arriving through various methods (e.g. downloaded in a web browser or pulled from the local ftp server to which outside users upload files) for different operating environments. While the software can be examined as it arrives, the host on which it is arriving or passing through may not be able to run the software for various reasons (e.g. wrong operating system, architecture, or available libraries). Static analysis is also useful for forensic analysis. It is important for our system to be able to detect old attacks, variations on old attacks, and novel attacks. There are several different static analysis approaches that we considered pursuing to meet these requirements. A signature scanner [5] is fast, but does not reliably handle variations on old attacks even when the signature language is highly expressive. Some commercial virus detection systems using this approach are unable to handle even modest code obfuscation [6]. Neither a policy enforcement approach [7] nor an emulation approach were selected because of the computational overhead [8].

Instead, we pursued a machine learning approach to achieve accurate detection, initially examining source code because it was easier to perform feature extraction [9]. Others have pursued similar strategies with binary data, although most examined detection of viruses on DOS and Windows systems. Feature extraction must be performed to achieve fast, accurate classification of malicious software. The amount of intelligence built into the preprocessing step ranges from none, where software is treated as byte sequences with different likelihoods of being in malicious software [10], to some, where a feature is either a byte sequence, a string or a dynamically linked library [11], to substantial, where software is converted into an abstraction pattern prior to matching [6]. In our approach, a few instructions are interpreted and machine learning combines these to create an accurate system that is moderately robust to obfuscating transformations, but which can quickly process new files.

3. System Overview

LIMACODE is a static analysis system that detects old attacks, variations on old attacks, and novel attacks. It uses a language-specific static feature extractor with a feed-forward neural network classifier since this type of system is able to define general classes of privilege escalating attacks and is therefore more robust in determining both novel attacks and variations on old attacks.

The remainder of this section provides a system overview to LIMACODE. A high level flow diagram of the detection process appears in Figure 1.

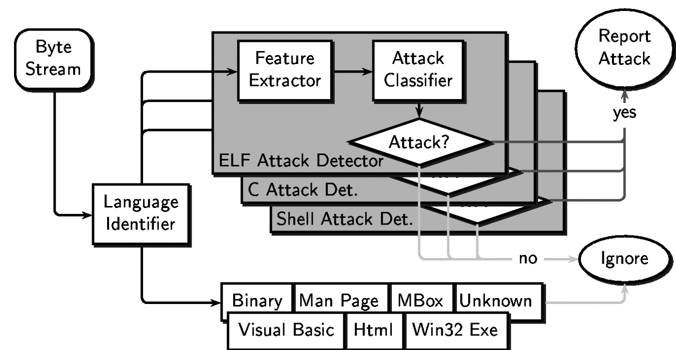

Figure 1. LIMACODE flow chart

The diagram depicts a byte stream (e.g. from a file or network packets) being fed into a language identifier. The language identifier determines which detector (language-specific feature extractor and attack classifier), if any, should be used to analyze the byte stream. The chosen feature extractor examines the byte stream for interesting features and reports them in the form of a vector of integers and real numbers; the vector is then fed into the attack classifier. The output of the attack classifier is the posterior probability that the byte stream does or does not contain privilege-escalating code.

Using this approach, our system is easily extensible to new attack vectors, such as those that employ scripting language formats, by adding more feature extractors and attack classifiers, and then updating the language identifier to include the new byte stream type.

The ELF Attack Detector discussed in this paper operates on ELF files; hence, in the remainder of the paper the byte stream source is assumed to be a file.

3.1. Language Identification

The first step in processing a given file is to identify its language type so that the appropriate feature extractor and attack classifier are used. The language classifier allows each file to only be processed by one detector, thereby speeding the overall detection process at the possible cost of a missed detection. LIMACODE uses a rule-based system that exploits a language’s defined structure and syntax to determine a file’s type. For ELF files, the language identifier looks for the presence of the ELF magic number at a fixed offset into the file. This simple approach has correctly identifying every ELF file we have encountered. For C and shell source code, the language identifier parses the contents of the file; however, this was not required for ELF files [4].

3.2. Attack Feature Extraction

In order to understand the privilege-escalating attack features and the extraction process, some background on ELF binaries is necessary. ELF binaries consist of a required header section and one or more optional sections [12]. The header section contains information such as the version, target architecture, and the virtual memory address at which the ELF file is loaded. The remaining sections are optional although there are some, such as the *.text* and *.data* sections, that are almost always present. The *.text* section contains the actual executable code, the *.data* section contains initialized data, and the *.rodata* section contains the read-only data.

The ELF feature extractor parses an ELF file into its *.text* and .(*ro*)*data* sections, analyzes these sections, and produces statistics that could identify a given file as malicious. Attack feature statistics represent the steps necessary for injecting code into one or more buffer overflow vulnerabilities. To achieve the best performance on the widest variety of attack code, each feature should encode all possible ways to accomplish a particular part of the attack.

We started with a large number of features in various groups, some inspired from the best features in the LIMACODE C and shell classifiers and some based on our knowledge of how privilege-escalating attacks work. We then used a forward-and-backward, leave-one-out selection process [13] to select those sets of features that best divided the sample space. The selected features can be categorized as a kernel call, instructions in data, or other, and are shown in Table 1.

Table 1: Features used to classify privilege-escalating code.

Group	Name	Count Type	Description
Kernel Call	Exec	N	*Exec* family of functions
	System	N	*System* family of functions
Instructions in Data	Payload	M	Instruction sequences or combinations typically found in injectable buffers
Other	Code in Strings	P	C or shell source code present in strings in the *.data* sections
	Size Data	M	Size of the largest *.data* section

Count Legend: (P)resent, (N)ormalized, (M)aximum.

There are several ways the features statistics are measured: *Present, Normalized, and Maximum*. *Present* indicates if a sample does or does not have a feature. *Normalized* is the number of times a feature appears in a sample normalized by an appropriate divisor to obtain a notion of the density of the feature. *Maximum* records the value of the window with the highest score. All possible combinations were considered during feature selection and the best method for counting a feature is presented in the *Count* column in Table 1.

The remainder of this section describes the observations that led to creating these features and the specifics of our implementations.

Kernel Calls

Observation: Privilege escalating code needs to pass information to a higher privilege process in order to exploit it. Sometimes this is done by starting a vulnerable program with carefully selected arguments. Empirical tests indicate that most benign binaries have a low density of calls to the program initiating services *exec* and *system*. Shorter exploit programs that launch a vulnerable, higher privilege application have a higher density of these calls. We imagine the same is true for inter-process communication calls as well, although we were unable to gather enough training and test samples to verify this.

Implementation: Independently decode and count the occurrences of process creation using the `exec` family of functions[1] that fully specify program path and input, and also the `system` call that uses the shell to specify the environment and determine the absolute program path.

Extension: Interprocess communication (`pipe/signal/shmat/connect`) should also be counted.

Instructions in Data

Observation: Certain types of executable actions rarely appear in localized parts of non-exploit *data* sections unless constructed specifically for injection and execution in a higher privilege process. Among these are software that isolates code location, zeros out registers, and includes control flow.

Implementation: The feature extractor examines a fixed-size sliding window of decoded instructions looking for particular instructions and instruction sequences that accomplish actions that are commonly performed by injected code, adding one point for each type found. Each action is counted only once, even if multiple examples of the action occur within the sliding window. The count thus encodes the number of diverse actions present in a window. The window is used to require locality of actions; window sizes of 20, 30, 40…100 were used with the training data and a window size of 50 consecutive

[1] `execl, execle, execlp, execv` and `execvp`

instructions was found to give the best discrimination. Large non-attack binaries may also contain many of these actions in the *.data* section, but they are distributed over the entire section, whereas dedicated exploit code has tended to have more densely packed features. The following paragraphs detail what each action does and why it is important to look for it.

The first action determines the location of code so that once the exploit has occurred and the buffer is executing, it can pass the address of local data as arguments both to functions provided by standard libraries and to the kernel. This is necessary because the exact location where the payload is injected can depend on the version of the victim application and late-bound library load order, among other factors. An example instruction sequence that will accomplish this is a relative `call` to a `pop` instruction. In this sequence, the call instruction causes the address of the next function to be placed on the stack. The immediate `pop` will place that address into a register, which can then be used as a reference point for its local data. If there is a `jmp` instruction that redirects control flow to the initial `call` instruction in this sequence, then two points are added to the total score for this window since this sequence is prevalent in our training data and is more indicative of an injectable buffer.

The next action looks for instructions that cause control flow changes, since injected code makes decisions based on return codes, and often loops to perform various actions (e.g. file scanning or denying access to some service). LIMACODE looks for local calls and jumps that direct control flow to somewhere within the instruction window. One point is awarded if one or more control flow actions are found.

Another common action is setting a register to zero. Empirical tests indicate that instructions and instruction sequences that zero registers are common in the part of the *.data* sections of malicious ELF files that contain an injectable buffer. There are many ways to do this; for this system, we only include common, single-instruction instructions that accomplish this: *Xor register register; mov register, 0;* and *imul register, 0.* One point is awarded if one or more instructions are found that set a register to zero, regardless of the register used.

The final code-in-data action identifies the presence of one or more kernel calls since injected code frequently needs to interact with the operating system. On x86 based Linux operating systems, this instruction is INT 0x80 (interrupt 0x80). This instruction causes a transition into the kernel from where the call is handled.

In regards to the *Payload* feature as a whole, we found that such a heuristic was necessary since simpler schemes, such as looking for runs of valid instructions, did not work because the IA-32 instruction set is very dense and a random sequence of bytes has a high probability of being a valid sequence of instructions. One of the features that was eliminated as a result of feature selection involved the detection of a payload by identifying valid sequences of instructions in the *.data* section.

Extensions: Other instructions that should also be counted include multiple-instruction sequences that zero registers as well as instructions that reference environment string memory locations.

Other

This final group of features also indicates that the sample attempts to increase privileges, but did not fit into the other two classes.

Observation: It is rare for non-exploit code to embed C or shell code in the *.data* section of an executable; in contrast, exploit code often includes shell code as part of its launching or exploitation process. Also, sometimes C source of an exploit is included in attack software so that it can be compiled differently based on the exact details of the victim application (e.g. version or configuration).

Implementation: In order to detect both C and shell code appearing in strings, all strings appearing in the *.data* section are used as byte stream inputs to a modified versions of the C and shell classifiers respectively [9].

Observation: The vast majority of our training samples had small *.data* sections. This is due to the fact that the exploit code we used for training contained an injectable buffer, the name of the vulnerable program, and little else. Compiler options may dramatically affect the size of the *.text* sections (e.g. static versus dynamically linked), but not the *.data* sections. While not a good indicator on its own, when combined with the other features it significantly improves the ability of the classifier.

Implementation: During the ELF parsing process, the size of the largest *.data* section is recorded and included as part of the feature vector.

3.3. Attack Classification

The attack classifier's neural network is a multi-layer perceptron classifier with a single hidden layer trained using back-propagation of errors [14]. A gradient descent method with a squared error cost function is used for training in which the new weights propagating backward through the network. Training time is negligible using these techniques on the feature vectors and sample sizes used here. Other machine learning techniques were informally explored, but this approach gave the best results for the cases considered.

Prior probabilities of the attack and training classes were equalized to maximize detection rate at the cost of an increased error rate, since more files are benign than malicious. LIMACODE is therefore more likely to

misclassify a benign file as malicious than to misclassify a malicious file as benign.

4. Data Sources

Benign and privilege escalating samples were collected at two different times in order to test the ability of the system to detect new, unseen attacks.

For the malicious samples, C source code was obtained from various websites. Only code that had privilege escalating intent was collected. There are 221 training samples that were collected between July 2001 and January 2002 and included attacks that were released from before that time period. The 68 testing samples were collected between January 2002 and September 2002 and only included samples released in that time period. Test samples were verified as distinct from training samples by performing a byte level comparison of each compiled test sample against the existing compiled training samples. The malicious C files were compiled in as similar a method as possible to the benign samples so as not to leave compilation artifacts that could easily identify the malicious samples.

The benign samples were taken from the /usr/bin directory of a default Red Hat 7.1 installation. The 1280 total benign files were portioned into training and test sets: 979 were randomly chosen to be in the training set and the remaining 301 were used in the test set, to match the ratio of the training and test set of the malicious data.

5. Evaluation

This section presents the ability of LIMACODE to detect new, unseen privilege-escalating ELF binaries and its data processing speed.

5.1. Detection

Figure 2 displays the accuracy of LIMACODE in the form of a detection error tradeoff (DET) curve [15]. In the figure, the false alarm percentage is plotted against the miss percentage for various operating points (i.e. thresholds applied to the output of the classifier). The axes are scaled by normal probability deviates to magnify the target zone.

Unlike fixed-heuristic or signature-based systems, LIMACODE can be operated over a range of operating values, with the operating point selected by a user who specifies the relative value of misses and false alarms. Three points are of particular interest for different uses: the point where the false alarm rate approaches zero, the equal error rate, and the point where the miss rate approaches zero. The first point is interesting for virus detection-like applications where the user wants some protection but mostly does not want other applications to be erroneously labeled. Here, the false alarm rate approaches zero when the miss rate is approximately 30%. Next, the equal error rate describes the point at which miss and false alarm rates are equally important; for LIMACODE's Malicious ELF detector the rate is 4.65%. The final point of interest is one which might be used to scan a captured disk, and for which missing an exploit is much worse then spending the time to examine a false alarm. The miss rate approaches zero when the false alarm rate is about 35%.

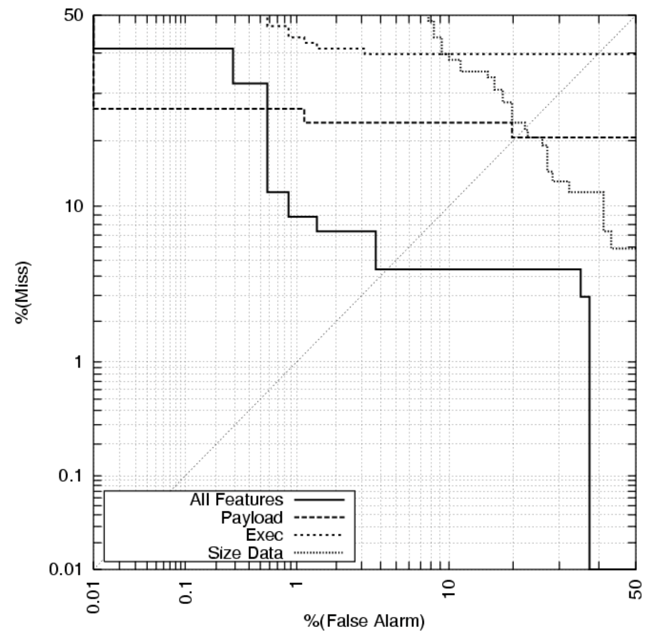

Figure 2: LIMACODE Performance

The results of training and testing the classifier on three different single features are also presented alongside the primary result, since it is conceivable that single features dominate the output. As is clear from the figure, combining multiple features significantly improves the accuracy of the system over most of its operating range. The curves for the accuracy of the isolated System and Code in Strings features do not appear on the graph as they lie outside the region in view. Although individually inaccurate, the integrated system accuracy improves when these features are included.

For false alarm rates less than about 0.6%, the single payload feature is a better discriminator than the classifier that uses all the features. At these low rates, features other than the payload feature introduce a significant amount of noise.

5.2. Throughput

In evaluating the throughput of the system, LIMACODE was used to classify the contents of the /usr/bin directory on a RedHat 9.0 installation. On an Intel Pentium III running at 800 MHz it classified 2,336 files with a total size of 170 MB in 142 seconds or 1.17 MB/s. During execution, LIMACODE spends the vast majority of its time in the feature extraction phase.

6. Discussion

The LIMACODE system does not use signature matching, and is therefore able to detect attacks that it has not seen before. Instead of looking for specific sequences of instructions, features representing actions that are required to exploit a vulnerability in another process encode the fact that there are multiple ways to accomplish the same goal. Our implementation requires locality of multiple required actions. For an attacker to hide from the system, he must obfuscate multiple actions. This is harder to do than to change an isolated signature.

Our approach is resilient to many common obfuscation techniques [6]. It is not affected by code transposition (in which instruction order of an attack is altered), because there is no explicit model of instruction sequences. It is not affected by register reassignment (in which the specific registers used by an attack is changed), because there is no explicit model of a given register for a given attack. It is relatively insensitive to instruction substitution, because in the case where instruction classes are used, all equivalent single-instruction cases are included. (It remains possible to use multi-instruction code to accomplish similar ends, and this is not yet addressed.) Finally, the system is insensitive to dead-code insertion, provided the amount of dead code injected does not cause the exploit to become longer than the scanning window.

There are a number of features that could be added to LIMACODE to increase its accuracy. First, the *Payload* feature could be extended to include more injected buffer actions such as identifying references to environment variable locations, decryptor blocks for obfuscated payloads, and typical API usage sequences. Also, the C source feature set identified calls to *link* as part of the *System Call* group. This feature helped detect exploits of race conditions. However, due to an insufficient number of samples in this newer data set, it was not included in the executable feature set. Collecting a sufficient number of such samples with which to train and test would allow LIMACODE to be tuned to detect this class of privilege escalating attacks.

It is, however, possible to bypass LIMACODE, by causing the counts of the features to change. An attacker could increase the feature count by adding unexecuted dead code to an exploit. Other techniques will work equally well. To respond to this, our system would need to either remove dead code or, equivalently, increase the window size in the presence of dead code. Alternatively, an attacker can decrease the feature count, perhaps by encrypting or obfuscating actions, encoding an action using an instruction or instruction sequence that we don't count, or by spreading out the actions in the *.data* section so that they fall outside of the code window.

Finally, an attacker can avoid the system altogether. Most modern UNIX-like operating systems have compatibility modes, and can execute the older a.out file format (as well as several others). While there is nothing to prevent us from adding support for these file formats, we have not done so.

7. Summary of Results

The most important result from this paper is that it is possible to build an accurate detector of unobfuscated ELF attack code by identifying the specific actions that privilege-escalating code must take in order to accomplish its goal and then detecting code which accomplishes these actions.

LIMACODE raises the skill level required for creating and transmitting an exploit into an enclave. We found that much of the easily obtainable privilege escalating code does not attempt to hide its intent. Therefore, in order to compromise a system protected by LIMACODE, an attacker would have to find or develop intentionally obfuscated attack code.

8. Acknowledgements

We would like to thank Craig Stevenson for his work on the C and Shell code part of the LIMACODE project [4] and his contributions to an early version of the malicious ELF file detector.

References

[1] S. Garfinkel and E. H. Spafford, *Practical Unix and Internet Security*, 2nd ed: O'Reilly & Associates, Inc., 1996.
[2] NIST, "ICAT Metabase," 2000.
[3] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, pp. 131-152, 1996.
[4] C. S. Stevenson and R. K. Cunningham, "Accurately Detecting Source Code of Attacks that Increase Privilege," presented at Recent Advances in Intrusion Detection, Davis, CA, 2001.
[5] S. Kumar and E. H. Spafford, "A Generic Virus Scanner in C++," Purdue University, West Lafayette, IN, Technical Report September 17 1992.

[6] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," presented at 12th USENIX Security Symposium, Washington, D.C., 2003.
[7] M. D. J. Bergeron, J. Desharnais, M. M. Erhioui, Y. Lavoie and N. Tawbi, "Static Detection of Malicious Code in Executable Programs," presented at Symposium on Requirements Engineering for Information Security, Indianapolis, Indiana, USA, 2001.
[8] F. Leitold, "Reductions of the general virus detection problem," presented at EICAR International Conference, 2001.
[9] R. K. Cunningham and C. Stevenson, "Accurately Detecting Source Code of Attacks that Increase Privilege," presented at Recent Advances in Intrusion Detection, Davis, CA, 2001.
[10] J. O. Kephart and W. C. Arnold, "Automatic Extraction of Computer Virus Signatures," presented at 4th Annual Virus Bulletin International Conference, Abingdon, England, 1994.
[11] M. G. Schultz, E. Eskin, E. Zadok, M. Bhattacharyya, and S. J. Stolfo, "MEF: Malicious Email Filter," presented at 2001 USENIX Annual Technical Conference, Boston, MA, 2001.
[12] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Standard, Version 1.2," 1995.
[13] *Feature Extraction Construction and Selection: A Data Mining Perspective*: Kluwer International, 1998.
[14] R. P. Lippmann, L. C. Kukolich, and E. Singer, "LNKnet: Neural Network, Machine Learning, and Statistical Software for Pattern Classification," *Lincoln Laboratory Journal*, vol. 6, pp. 249-268, 1993.
[15] A. Martin, G. Doddington, T. Kamm, M. Ordowski, and M. Przybocki, "The DET Curve in Assessment of Detection Task Performance," presented at Eurospeech97, Rhodes, Greece, 1997.

A Prototype Tool for Visual Data Mining of Network Traffic for Intrusion Detection*

William Yurcik Kiran Lakkaraju James Barlow Jeff Rosendale
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign
{byurcik,kiran,jbarlow,jeffr}@ncsa.uiuc.edu

Abstract

Human comprehension of the overall security of large and complex networks of machines is currently limited since security staff use multiple applications, each with limited scope, most without visual output. We have developed a new tool called NVisionIP for allowing an operator to interactively assess the security situational awareness of an entire network using visualizations derived from NetFlow log data that is continuously collected. This tool is a novel contribution because for the first time it shows the macro/micro relationships between individual machine events, subnet events, and network-wide events on a single screen, specifically a color-coded grid with drill-down views representing an entire Class B IP address. We provide examples of experimental output results showcasing tool utility for managing security on large and complex networks, concluding with plans for deployment in production environments.

1. Introduction

The current state of computer security on most networked systems is dangerous and by most metrics getting worse. There are many unpatched software vulnerabilities as well as point-and-click software that will exploit these vulnerabilities allowing intrusions and disruptive attacks. While the Internet has enabled impressive productivity gains due to connectivity, this same connectivity also allows malicious attackers worldwide direct access to your network perimeter. In addition, corporate security incident surveys report that insider attacks, staff with privileged access and knowledge, are an even greater threat.

We propose that the solution for security begins with human awareness and subsequent understanding of exactly what is occurring on a network – *Know thy network!* Ignorance is bliss, but it is also very risky; unfortunately this is still where most organizations find themselves due to the lack of satisfactory tools. While situational awareness of computational security has evolved from “Is there a problem?” to “Where is the problem?” to “What is the problem?” - state-of-the-art tools still do not facilitate assessing entire networks as a whole. Identifying and disabling individual compromised machines, scanning for known vulnerabilities that are unpatched, and filtering specific perimeter traffic may produce short-term gains, however, the ability of a human operator to efficiently, clearly, and continuously assess the security posture of an entire network at any instant in time are long-term requirements currently not being addressed.

We would like to briefly highlight two of the stated requirements: (1) *monitoring an entire network* and (2) *monitoring continuously*. *Monitoring an entire network* as a holistic system is important because a malicious software foothold (no matter how small) anywhere inside the perimeter will endanger all machines [13]. It is vital to be able to assess the security posture of a network at both the **macro** and **micro** scales simultaneously in order to comprehend the relationship between individual events (an intrusion on an individual machine) and network-wide events (disruptions and/or attacks on multiple machines across the network). We will show how the macro/micro views has enabled detection of attacks that otherwise would not have been detected. *Monitoring continuously* is important because there is a need for vigilance over time – technology changes over time will necessitate adaptation by both offense and defense that should be considered and attackers are both intelligent and persistent, they will circumvent static protection unless monitoring can dynamically track and be poised to react to new attacks (so called “zero-day” exploits). We are not expecting a human to sit at a screen 24X7 but will also show how animated visualizations can leverage human cognitive abilities to understand events in the time domain efficiently and effectively.

We have developed a visual data mining software tool that meets these requirements. Our unique tool, NVisionIP, allows an operator to interactively monitor the security status of an entire network on one screen using a

* This research is supported in part by a grant from the Office of Naval Research (ONR) within the National Center for Advanced Secure Systems Research (NCASSR) <www.ncassr.org>

visualization derived from processing audit log data that is continuously collected. NVisionIP is novel because no other tool allows an operator to visually assess situational awareness of an entire network in one screen (actually a Class B IP address space of 65K hosts with each host having 130K (TCP/UDP ports) based on time-series events. Typically security staff can view only small, highly aggregated portions of the network and are forced to use multiple applications because each individual application only provides limited information (signature matches, network/machine performance status, traffic filter statistics, worm/virus detection).

Since *clear* interpretation by a human operator is another requirement, we intentionally made the output of NVisionIP visual for reasons of human cognitive processing: (1) it is estimated that humans can process visual information at 150Mb/s [11], (2) human vision is especially tuned for discriminating tiny but high contrast visual effects (referred to in psychology as the just-noticeable-difference), and (3) humans perform well at recognizing visual patterns especially when intuition can be used (ecological design).

The remainder of this paper is organized as follows: Section 2 provides background on network visualizations and the NetFlow application (NetFlow is our *efficient* data source satisfying our last requirement). Section 3 presents the architecture of NVisionIP and output results from different experiments. Section 4 discusses both the significance and limitations of this new tool. In Section 5 we close with a summary, conclusions, and directions for future work.

2. Background

To put our research in context, we summarize previous work in visualizing networks and security, as well as introducing the application we utilize for source data. This background highlights the unique contribution of our new tool to current capabilities since previous work is primarily focused on network overlays to geographical maps or logical network configurations, both of which convey few insights about security.

2.1. Visualizing Networks and Security

[3] provides a comprehensive overview of network visualizations. Low-dimension visualizations include networks mapped onto geography, logical diagrams of equipment (including network management tools based on SNMP), traffic level representations in x-y diagrams/pie charts/histograms, connectivity diagrams with links sized/colored corresponding to bandwidth capacity, and packet-level animation of network simulations (as best exemplified in OPNET[1] and Nam)[4]. High-dimension visualizations include the peacock diagrams of Lumeta[2] which show the Internet in its own space independent of geography and the SKITTER diagrams of CAIDA[3] which show peer interconnections projected on a polar-projected longitude graph.

There has been a small amount of work combining network visualization and security that we now describe (in chronological order). [5] presents a prototype design tool from the Harris Corporation named the Network Vulnerability Tool (NVT) which visually depicts the network topology under study (using HP's Openview SNMP product) and generates a vulnerability assessment window with results from proactive scans and a vulnerability database. [6] proposes visual symbols to better communicate security events to users. [12] states visualization should be the next focus of intrusion detection systems (IDSs) since it can convert the essentially *serial* IDS alarm process to the *parallel* process of visual perception. [10] presented a visualization of network routing information that can detect inter-domain routing attacks and routing misconfigurations. The most relevant work is a rapid visual feedback system originally developed by the NASA Jet Propulsion Laboratory for tracking the status of spacecraft components that has now been adapted for network security as a commercial tool called TowerView Security [7]. To our knowledge, [7] and [10] are the only working examples of computer network security visualizations and they are both significantly different that what we present in this paper.

2.2. Source Data: NetFlow Audit Logs

We utilize NetFlow audit logs as the data source for our tool. Although there are a number of tools that process NetFlow data[4], the only NetFlow visualization tool we are aware of is FlowScan that produces near real-time x-y utilization diagrams of network traffic levels geared toward bandwidth management [9]. NetFlow data is derived from routers caching recent flows for lookup efficiency. For the NetFlow application, a distinct flow is defined as either a unidirectional TCP connection (where a sequence of packets take the same path) or individual unidirectional UDP datagrams. As shown in Table 1, an individual record within a NetFlow log file consists of some or all of the following: IP address pairs (source/destination), port pairs (source/destination), protocol (TCP/UDP), packets per second, timestamps (start/end and/or time duration), and byte counts. While

[1] http://www.opnet.com/
[2] http://www.lumeta.com/
[3] http://www.caida.org/
[4] http://www.splintered.net/sw/flow-tools/

Table 1. NetFlow Record Contents

	src_ip	dst_ip	bytes	start_time	end_time	src_port	dst_port	ip_prot	tos
0	2374907906	2374904584	120	1583469988	1583472984	1964	4321	6	0

NetFlow logs can grow large over time, depending on the size of the network, amount of data transferred, and duration of analysis, NetFlow logs are efficient since they are typically much smaller in size than logs that capture raw packets. NetFlow was initially introduced in Cisco routers as a proprietary tool but has since become a defacto feature across the majority of router vendors, with the IETF Realtime Flow Management (RTFM) working group preparing to standardize its implementation.[5]

NetFlow data is difficult to spoof and has been used to identify security compromises based on suspicious traffic patterns between source/destination IP addresses and/or ports. NetFlow also exhibits a strong multiplicative effect in that once a single clue is found, an operator can subsequently use NetFlow to monitor traffic to specific IP addresses and ports leading to other compromised machines. Examples of the use of NetFlow for computational security include identifying the source and destination of denial-of-service attacks as well as identifying compromised machines involved in:

- uploading/downloading unapproved software (high traffic levels from a non-server machine)
- hosting Internet relay chat (IRC) servers (a large number of unexpected flows from multiple source IP addresses to a single destination IP address which is a non-server machine)
- worm/virus propagation (a large number of flows from a single source IP address to multiple destination IP addresses or ports)
- network and host probing[6] - small scale preattack reconnaissance to identify machine platforms and port services that may be vulnerable
- network and host scanning[7] - large scale preattack reconnaissance to identify machine platforms and port services that may be vulnerable, also used to virtually map a network for malicious navigation, a host scan can be detected as a source IP "touching" more than a preset threshold number or ports on a single machine while a network scan can be detected as a source IP "touching" more than a preset threshold number of destination IPs.
- hosting remotely installed "bots" that are remote controlled (traffic patterns on unusual ports)

While NetFlow logs efficiently provide a rich set of network data, its size over time and streaming nature makes finding useful security information difficult (such as identifying the security situations just listed). Thus the original scope of our investigation was to transform unmanageable network data, from a source like NetFlow, into something manageable for security purposes without losing information. While our border router operating at Gigabit/second second speeds represents a challenge for NetFlow I/O interfaces for data management that can only be solved using sampling techniques (with current technology), we have been able to merge NetFlows from multiple internal routers operating at slower speeds without losing any information.

3. NVisionIP

NVisionIP is designed to meet three objectives. The first objective is to accurately and concisely visualize status information of an entire IP address space on one screen. The second objective is to provide more detailed information about specific machines. The third objective is the ability to process different sources of input data. NVisionIP achieves all three of these objectives, the first by representing an entire class B IP address space as a 255X255 grid, the second by allowing an operator multiple views of traffic activity to/from specific machines within this address space, and the third by making the architecture independent of source data.

[5] The RFTM IETF Working Group home page can be found here: <http://www2.auckland.ac.nz/net//Internet/rtfm/>. RTFM concerns itself with current issues in traffic flow measurement including security issues relating to both traffic measuring devices and the data they produce and existing work in traffic flow measurement.

[6] a host probe is a single connection request from a single source IP address to a single destination IP address, a port probe is a single connection request from a single source IP address to a single destination port on a single IP address, in the same category as operating system fingerprinting

[7] exhaustive one-dimensional host range scans – connection requests from a single source IP address to a *sequential range of multiple destination IP addresses*, exhaustive one-dimensional port range scans – connection requests from a single source IP address to a *sequential range of multiple destination ports on a single destination IP address*, exhaustive two-dimensional host/port range scans – connection requests from a single source IP address to a *sequential range of multiple destination ports on a sequential range of multiple destination IP addresses*, distributed scans – connection requests from *multiple source IP addresses* to multiple/single destination IP addresses or multiple/single destination ports on multiple/single destination IP addresses, temporal scans – connection requests from multiple/single source IP addresses to multiple/single destination IP addresses and/or multiple/single destination ports on multiple/single destination addresses *over a period of time*

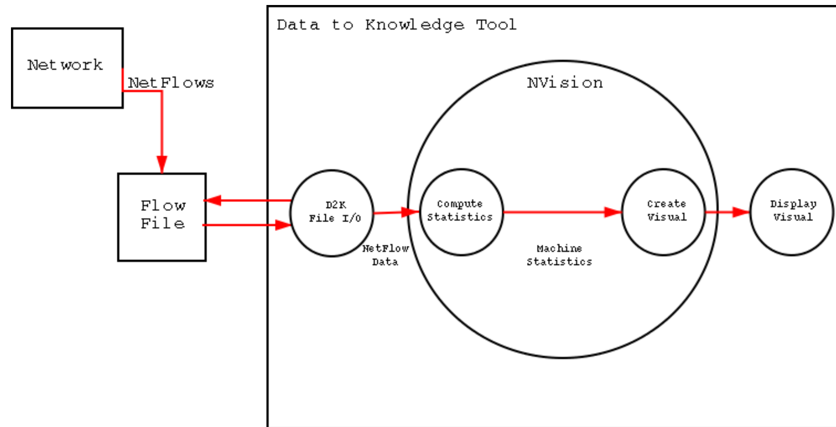

Figure 1. The NVisionIP Architecture

The NetFlow logs that serve as our initial input to NVisionIP are created by multiple routers and stored by a "collector" data server in one unified file every five minutes. These unified files can then be cumulatively loaded (automatically or manually) to analyze a specific time period. Once the NetFlow data is loaded, NVisionIP generates the following statistics for each machine corresponding to an IP address (note not all IP addresses have a corresponding physical machine):

- Count of all connection flows to and from
- Count of external source IP flows
- Count of external destination IP flows
- Byte count of all traffic to/from
- Byte count of all outbound source traffic binned by port number
- Byte count of all inbound destination traffic binned by port number

We are developing animations of events over time by incorporating two save features: (1) an applet that contains current visualization environment variables with file information for further query ability and (2) a single frame .jpg visualization file that can grouped with other single frames in an application like MacromediaMX Flash to create a multi-frame animation.

3.1. System Architecture

Figure 1 highlights the organization of NVisionIP and its relationship to the Data-to-Knowledge (D2K) data mining software package. D2K is a rapid, flexible machine learning system that effectively integrates different data mining methods. It offers a visual programming environment that allows users to connect software components using drag-and-drop. D2K also supplies a standard set of software modules and application templates with a standard API for component development. Other advantages of building within the D2K environment are: fast file I/O, efficient internal data representation, and multiple visualization options [1].

Integrating information from heterogeneous sources is an overarching goal of this research. The use of D2K to create an internal representation of the data creates a layer of abstraction between NVisionIP and the input such that new audit log sources can be incorporated. Leveraging the D2K framework to split NVisionIP into two independent parts, the first to compute statistics and the second to display visualizations makes NVisionIP easily extensible.

3.2. Experimental Results

Before reporting our experimental results that are in the form of visual output, we would first like to give an example of our statistic generation capability and its relevance to security. Given that the instrumented network has approximately two thousand machines, Table 2 shows a concentration of flow connections into a relatively small number of machines. This concentration is indicative of a scale-free network where connectivity is unevenly distributed such that a focused attack on hub machines can have devastating consequences [2].[8] This statistic provides a way to identify and thus focus protection on hub machines running approved services, however some hubs may turn out to be unexpected machines with suspicious traffic. This statistic can be generated for each of many different services (e.g., http, ftp, telnet) since different machines will generally be hubs

[8] In a random network, the distribution of the number of links from one node to other nodes is a normal (Bell Curve) distribution. Scale-Free networks are characterized as having an uneven distribution of connectedness with "very connected" hub nodes that shape the way the network operates (including security and survivability). The term scale-free refers to the ratio of the number of hub nodes to the number of nodes in the rest of the network remaining constant as the network changes in size (scale). Particularly devastating attacks on a hub include denial-of-service attacks for disruption and targeted worm/virus infections for speeding propagation via cascading outbreaks.

for different services.

Table 2. Flow Connection Statistics

Concentrated IP Addresses	Sum Count All Flow Connections	Flow Count - Internal IP listed as Destination Address (Ingress)	Flow Count - Internal IP listed as Source Address (Egress)
Top 5 IP addresses	28.8%	27.7%	32.4%
Top 10 IP addresses	37.3%	35.6%	40.6%
Top 15 IP addresses	44.6%	42.6%	48.1%
Top 20 IP addresses	50.3%	47.7%	54.4%

Figure 2 shows the visual input/output GUI of NVisionIP as it displays an entire network as a color-coded grid allowing users to search for patterns that provide knowledge about the state of the computational security. The instrumented Class B network being investigated is represented as a grid of 255 X 255 boxes (each box is a 2 pixel by 2 pixel) in order to provide 65,025 possible IP addresses.[9] According to Tufte, humans are able to distinguish up to 625 points in one square inch, a density higher than we have created (please note figures in this paper are reduced, the actual screen images are larger) [11]. Each 2 X 2 pixel box in the grid represents one IP address. The subnets within the instrumented network are listed on the X-axis and the hosts within the subnet are listed on the Y-axis. Each pixel box within the grid represents an attribute of the corresponding IP address classified by colors with a legend shown in the output window. For the purposes of this black and white publication, the color range has been mapped to a gray scale for later figures. In practice, the use of contrasting colors (green, yellow, red) greatly enhances discrimination.

Although a grid provides assessment of the overall state of security for an entire address space, it is still necessary to "drill-down" to specific IP addresses for more information. NVisionIP allows two levels of interactive zoom capabilities: (1) to a subset of the network and (2) to the port activity within a specific IP address. At each stage of magnification, the user can select which attribute to view such as flow connection or byte counts.

The GUI provides a 2-level interactive filtering ability for all possible query combinations: (1) IP addresses (all/source/destination/subset); (2) ports (all/source/destination/subset); (3) protocols (all/subset); and (4) activity type (flow connections/byte count). The motivation behind this comprehensive capability is that it is hard to anticipate which information may be useful in discovering future security events.

In Figure 3 we show a visualization of flow connection counts for a network highlighting these zoom capabilities. Figure 3A shows *the Galaxy View* of an entire Class B IP address space where the color of each grid point maps to user-specified bins corresponding to count ranges and represents the number of times a specific machine has appeared as a source or destination IP address in the flow file. Figure 3A clearly shows the high-traffic subnets as vertical line patterns corresponding to the instrumented network. After using a mouse to select a subset of machines to investigate further, an operator can view more detailed information. Figure 3B shows an inset displaying port traffic histograms of a subset of IP addresses (*the Small Multiple View*). Figure 3C shows an inset of information about one specific machine selected by the mouse input, in this case port byte traffic information (*the Machine View*).

The NVisionIP GUI, shown most clearly in Figures 2 and 3A, is split into three sections: (1) the top left contains statistical information about the corresponding flow file; (2) the bottom allows operators to dynamically select which statistic to visualize; and (3) the right-hand side (60% of the GUI) contains the main content - a Galaxy View of a Class B IP address space. In addition to a mouse-over event-handler that displays a small pop-up IP address adjacent to each dot under the pointer, we have implemented a linear magnification widget that can be dragged across the screen to highlight areas of interest. Under development is a fisheye capability that will provide a third dimension to further highlight areas of interest by user-controllable distortion.

Figure 4 is a Galaxy View of byte counts (to and from aggregate) for all active machines found in a particular flow file. An unusually large traffic volume may indicate a compromised machine subverted as a server of non-approved software. As a real example, we recently had an intrusion on the instrumented network where large files were subdivided into smaller files (all with the exact same byte count) for transfer to and from compromised machines. In this case we are able to add a new attribute to NVisionIP that highlights the machines with this particular byte count signature.

Figures 5 and 6 are Galaxy View of ingress (inbound) and egress (outbound) IP flow connections for all active hosts within a NetFlow file respectively. For each

[9] Note one minor detail about the mapping of all possible IP addresses to actual machines: not all possible IP address within an address space are valid to map to actual machines, some IP addresses are reserved for other purposes (the same also holds for port numbers).

machine within the internal instrumented network, we measured the number of times it appeared as a destination/ingress (source/egress) IP address in a flow connection with external IP address. While such traffic may be normal, it can also indicate a compromised machine that has been widely advertised in the underground for downloading unapproved software.

Figure 7 shows the difference in the number of aggregate (ingress and egress) flow connections on the instrumented network between two points in time. This comparison output can be used to further pinpoint suspicious machines we alluded to in Figures 5 and 6 (machines that have been compromised in the interim period of time and are now exhibiting different flow connection patterns as a result). To enhance human cognitive abilities for discovery, we use color processing to emphasize. It should be noted that this Galaxy View has a reversed color scheme (not a gray scale) to indicate a difference-file with darker colors (black, green) indicating little or no change and contrasting colors (red, yellow) indicating large change. More detailed change information can be found by zooming to either the Small Multiple View or the Machine View.

4. Discussion

It has become more difficult to characterize application activity on a network due to: (1) increases in the number of different applications, (2) applications (malicious and otherwise) whose underlying protocol does not depend on registered well-known port numbers, and (3) dynamic changes in application mix over time [8]. This has made it challenging to profile normal network traffic activity in order to distinguish suspiciously abnormal network traffic activity as a sign of potential security events (anomaly detection).

NVisionIP based on NetFlow data facilitates characterization of network traffic since it provides: (1) an overall view of an entire IP address space for selected traffic dimensions (bytes, flow connections) at a specific instant in time; (2) an interactive specific view for selected port traffic dimensions (bytes, flow connections) on individual machines at specific instants in time; and (3) the ability to interactively contrast views from different instances in time or between different machines. Changes in the IP address space as represented in the Galaxy View using visual cues of spot location, color, and geometric patterns have the capacity to transmit on the order of Mbytes of information to a user when considering all the possible permutations of these cues. For example, we have used the Galaxy View and Small Multiple View to visually determine patterns of fast and slow network scans (over time) and small-scale DoS attacks that otherwise would not have been detected using multiple IDS alerts.

In the Machine View, comparing histograms of the traffic byte counts or flow connection counts to/from different ports on different hosts or on the same host over time also has the capacity to transmit on the order of Mbytes of information to a user when considering the number of potential ports per host and all the possible count levels. For example, we have used the Machine View to visually determine patterns of fast and slow host scans (over time) and single machine sources of network-wide events that otherwise would not have been detected.

The primary limitation we faced with NetFlow data is generating statistics. For the instrumented network under investigation, NetFlow generated on the order of 500 Mbytes daily which makes generating statistics a lengthy process. Although the time to generate statistics cannot be significantly reduced, we found caching statistics such that calculations do not need to be repeated can reduce time processing.

5. Summary

We present a visual data mining tool, NVisionIP, which allows a human operator to interactively visualize the security status of an entire IP address space of networked machines in one screen. We report results from experiments based on NetFlow source data that convey how NVisionIP can be used to assess the situational awareness of a network for security. NVisionIP has been designed to accept multiple data sources so our initial success with NetFlow source data is especially encouraging since the NetFlow application is not specifically designed for security analysis. We are in the process of creating a website for distribution of NVisionIP including installation instructions, version announcements, hot fixes, and licensing restrictions (it is hoped we can pursue open source).

NVisionIP is a novel advance for managing security because it is the only extensible tool that currently provides a simultaneous view of events on individual machines, subnets, and across an entire IP address space (in the case of NCSA a Class B IP address space of 65K hosts with each host having 65K ports). The unique contribution of this tool is that it provides a new visualization capability to detect attacks with macro/micro relationships that otherwise would not be identified. Future work is focused on gaining experience with NVisionIP in different production environments. We plan to consider usability feedback from security experts; test different input data sources, develop GUI enhancements and automated pattern recognition algorithms, and last, but not least, evaluate impact on actual security incident detection, prediction, and response.

6. Acknowledgments

We especially thank Jennifer Rexford and Carsten Lund of AT&T Labs-Research for their insightful feedback and detailed suggestions from their own work with NetFlows for network management. We received important input and encouragement from Jiawei Han and Yuanyuan Zhou from the Department of Computer Science at the University of Illinois at Urbana-Champaign and Vipin Kumar, Jaideep Srivastava, Yongdae Kim, and Paul Dokas from the Department of Computer Science at the University of Minnesota. We would like to acknowledge the following members of the NCSA Security Research team who made significant indirect contributions to this paper (in alphabetical order): Loretta Auvil, Ratna Bearavolu, Randy Butler, Dora Cai, David Clutter, Yifan Li, Doru Marcusiu, Duane Searsmith, David Tcheng, Michael Welge, and Xiaoxin Yin. Lastly we would like to thank the anonymous reviewers for their insightful comments most of which we have been able to incorporate here in this paper with the rest to be addressed in future papers.

7. References

[1] L. Auvil, *D2K Reference Manual*, National Center for Supercomputing Applications, 2001. <http://alg.ncsa.uiuc.edu>

[2] A-L. Barabasi, *Linked: The New Science of Networks*, Perseus Publishing, 2002.

[3] M. Dodge, R. Kitchin, *Atlas of Cyberspace*, Addison-Wesley, 2001.

[4] D. Estrin et al., “Network Visualization with Nam, the VINT Network Animator,” *IEEE Computer*, Nov. 2000, pp. 63-68.

[5] R. Henning, K. Fox, “The Network Vulnerability Tool (NVT) – A System Vulnerability Visualization Architecture,” *National Information Systems Security Conference (NISSC),* 1999.

[6] H. Hosmer, “Visualizing Risks: Icons for Information Attack Scenarios,” *National Information Systems Security Conference (NISSC),* 2000.

[7] W. Jackson, “NASA Software Finds Cybersecurity Niche,” *Government Computer News*, Sept 9, 2002, p. 46.

[8] D. Liu, F. Huebner, “Application Profiling of IP Traffic,” *IEEE Local Computer Networks (LCN)*, 2002.

[9] D. Plonka, “A Network Traffic Flow Reporting and Visualization Tool,” *USENIX LISA XIV*, 2000.

[10] S-T. Teoh et al. “ELISHA: A Visual-Based Anomaly Detection System,” *RAID 2002*.

[11] E. Tufte, *Visual Display of Quantitative Information* 2nd edition, Graphics Press, 2001.

[12] P. Varner and J. Knight, “Security Monitoring, Visualization, and System Survivability,” *Information Survivability Workshop (ISW),* 2001.

[13] W. Yurcik and D. Doss, “A Survivability-Over-Security (SOS) Approach to Holistic Cyber-Ecosystem Assurance," *IEEE Workshop on Information Assurance*, 2002.

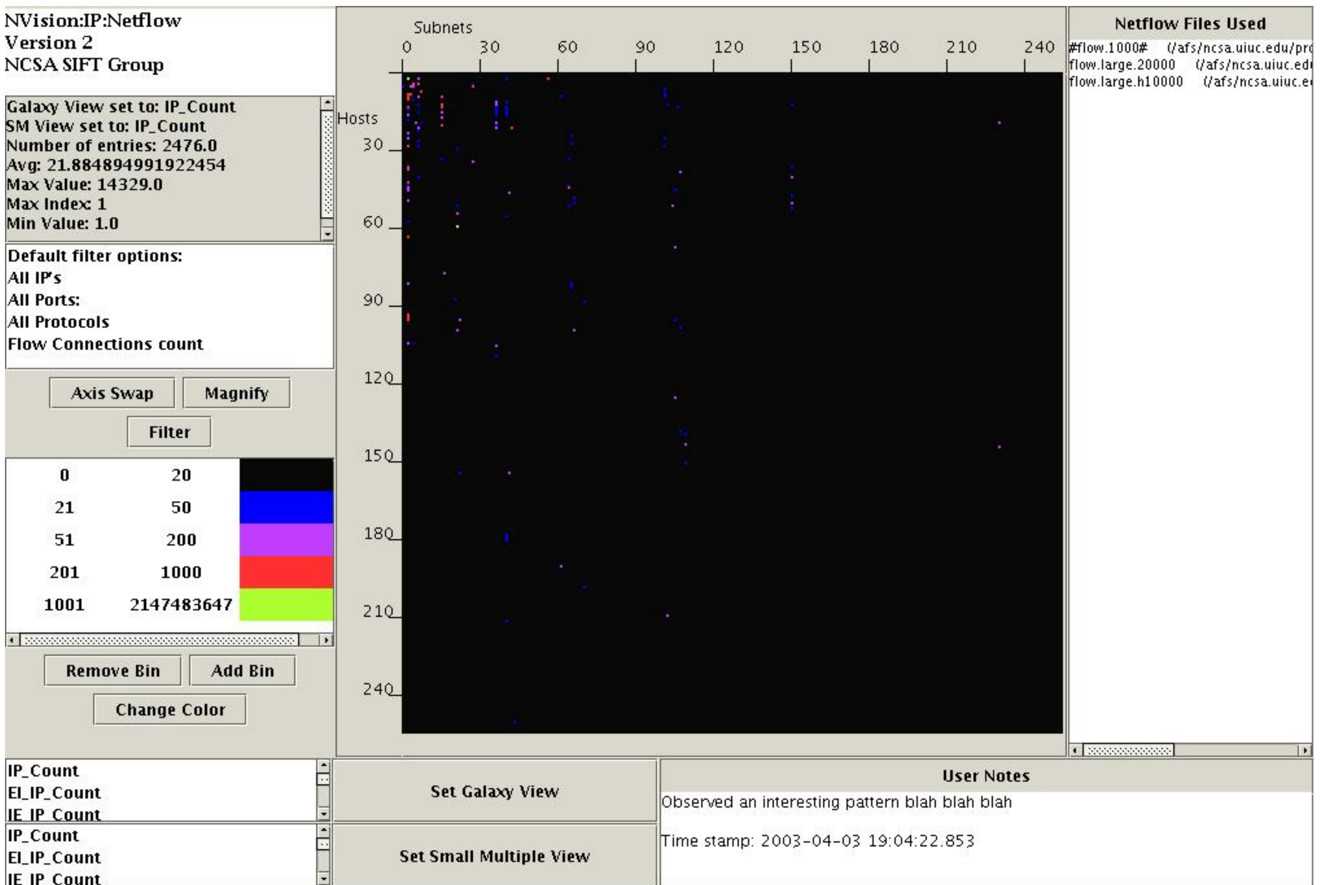

Figure 2. The NVisionIP GUI

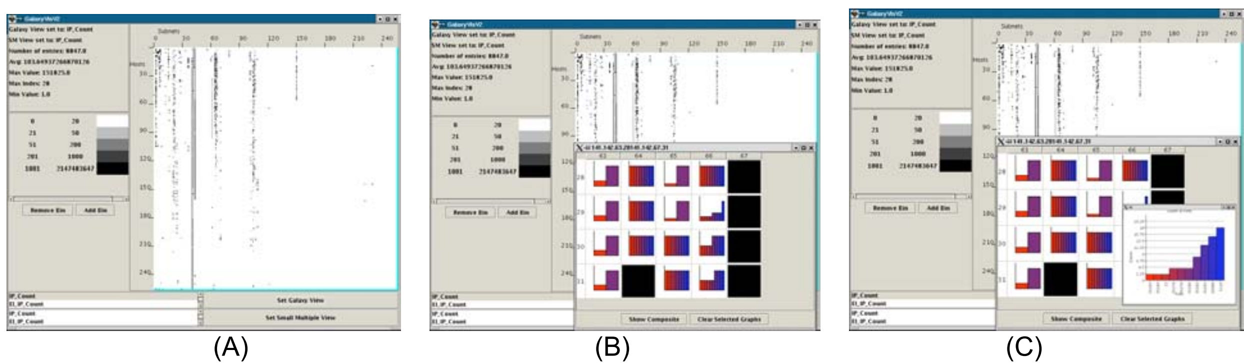

Figure 3. NVisionIP Connection Count Output - (A) "Galaxy View"; (B) "Small Multiple View" inset within the Galaxy View (C) "Machine View" inset within the Small Multiple View inset within the Galaxy View

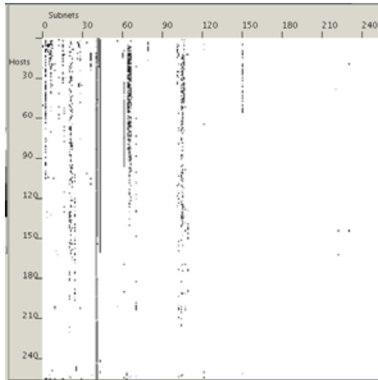

Figure 4: NVisionIP Byte Count Output

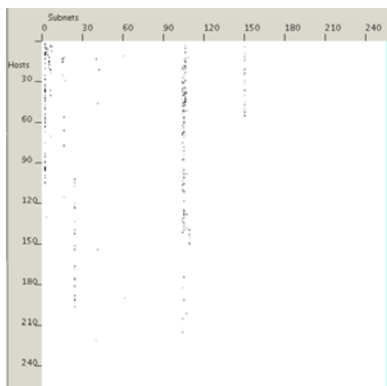

Figure 5: Ingress Flow Count

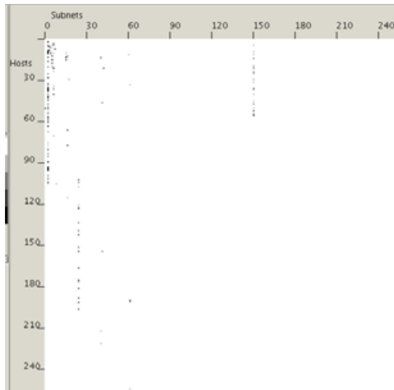

Figure 6: Egress Flow Count

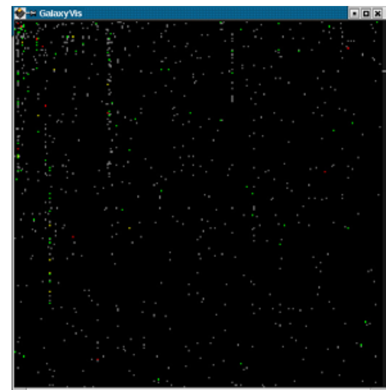

Figure 7: Flow Count Difference