Detecting Anomalous and Unknown Intrusions Against Programs*

Anup K. Ghosh, James Wanken, & Frank Charron
Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
poc:aghosh@rstcorp.com
www.rstcorp.com

Abstract

The ubiquity of the Internet connection to desktops has been both boon to business as well as cause for concern for the security of digital assets that may be unknowingly exposed. Firewalls have been the most commonly deployed solution to secure corporate assets against intrusions, but firewalls are vulnerable to errors in configuration, ambiguous security policies, data-driven attacks through allowed services, and insider attacks. The failure of firewalls to adequately protect digital assets from computer-based attacks has been boon to commercial intrusion detection tools. Two general approaches to detecting computer security intrusions in real-time are misuse detection and anomaly detection. Misuse detection attempts to detect known attacks against computer systems. Anomaly detection uses knowledge of users' normal behavior to detect attempted attacks. The primary advantage of anomaly detection over misuse detection methods is the ability to detect novel and unknown intrusions. This paper presents a study in employing neural networks to detect the existence of anomalous and unknown intrusions against a software system using the anomaly detection approach.

1 Introduction

The connectivity of the Internet to corporate, academic, and home users' desktop machines has become ubiquitous. The Internet has enabled nearly seamless connectivity of users via email and World Wide Web pages among other services. While the Internet has enabled the boon of electronic commerce, it has also been an enabling medium for computer-based attacks against corporate and university assets exposed to the Internet.

Firewalls were initially touted as the panacea to computer security problems. By restricting access to computer systems through known ports, firewalls served to eliminate computer exploitation attacks through network services that run often unbeknownst to the host site. Now, it is commonly understood that firewalls merely reduce exposure rather than eliminate vulnerabilities in computer systems. Computer-based attacks can compromise digital assets in spite of firewalls by exploiting errors in firewall configuration, exploiting ambiguities in security policies, obtaining access around firewalls through backdoors and computer modems, attacking network services allowed through the firewall, and using insider privilege to gain unauthorized access to sensitive assets. Furthermore, firewalls often engender a false sense of security that results in relaxed security at the individual host machines. As a result, once a penetration through or around a firewall is successful, internal webs of trust between computers can be exploited to obtain increasing levels of access to sensitive assets.

The failure of firewalls to adequately secure computer systems has led to the growth of the intrusion detection software industry. Intrusion detection software attempts to detect possible attacks against software systems in real time before critical assets are compromised. While no one purports intrusion detection software to be a silver bullet to computer security problems, combining intrusion detection with other security mechanisms such as firewalls, strong access controls, one-time passwords for remote authentication, secure shells, and regular auditing can provide defense in depth that can stymie the majority of computer-based attacks. Without the assistance of intrusion detection systems, most organizations are not aware

*This work was funded by the Defense Advanced Research Projects Agency (DARPA) under Contract DAAH01-97-C-R095. THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

of the number of attacks they experience on a daily basis against their intrusion detection systems. Intrusion detection systems often have two components: (1) detection of possible attacks, and (2) response to attacks. This paper addresses the first component, *i.e.* how to effectively detect attacks against computer systems. The second component involves implementing an organization's policy for either blocking an attack in progress or entrapping the attacker in a snare.

Intrusion detection techniques can be partitioned into two main approaches: misuse detection and anomaly detection. Misuse detection methods attempt to model attacks on a system as specific patterns, then systematically scan the system for occurrences of these patterns [Kumar and Spafford, 1996, Lunt, 1993, Garvey and Lunt, 1991, Porras and Kemmerer, 1992, Ilgun, 1992, Monrose and Rubin, 1997]. This process involves a specific encoding of previous behaviors and actions that were deemed intrusive or malicious. Anomaly detection assumes that intrusions are highly correlated to abnormal behavior exhibited by either a user or an application. The basic idea is to baseline normal behavior of the object being monitored and then flag behaviors that are significantly different from this baseline as abnormalities, or possible intrusions. See [Lunt, 1993, Lunt and Jagannathan, 1988, Lunt, 1990, Lunt et al., 1992, D'haeseleer et al., 1996, Porras and Neumann, 1997] for sources on anomaly detection approaches.

The most significant disadvantage of misuse detection approaches is that they will only detect the attacks for which they are trained to detect. Novel attacks or even variants of common attacks often go undetected. In a time when new security vulnerabilities in software are discovered and exploited every day, the reactive approach embodied by misuse detection methods is not feasible for defeating malicious attacks.

The main advantage of anomaly detection approaches is the ability to detect novel attacks against software systems, variants of known attacks, and deviations from normal usage of programs regardless of whether the source is a privileged internal user or an unauthorized external user. One drawback of anomaly detection approaches is that well-known attacks may not be detected, particularly if they fit the established profile of the user. Once detected, it is often difficult to characterize the nature of the attack for forensic purposes. Another drawback of many anomaly detection approaches is that a malicious user who knows he or she is being profiled can change his or her profile slowly over time to essentially train the anomaly detection method to learn his or her malicious behavior as normal. Finally, a high false positive rate may result for a narrowly trained detection algorithm, or conversely, a high false negative rate may result for a broadly trained anomaly detection approach.

In this paper, a novel anomaly detection approach to intrusion detection is described. Because new vulnerabilities in software are reported on a daily basis in forums such as Bugtraq[1], the challenge of intrusion detection research is to create techniques that can detect novel attacks against computer systems. The status quo in commercial intrusion detection software is represented by misuse detection approaches that scan for known attacks. These techniques cannot detect novel attacks against computer systems, and often times they do not detect simple variations on known attacks.

A key distinction of the work presented here is that intrusion detection is performed on software programs. Most intrusion detection systems analyze either network traffic or host system logs. This work focuses on system processes because attacks against computer systems are in fact attacks against specific software programs. By analyzing the usage or mis-usage of specific software programs, computer-based intrusions can be tracked at a finer grain of resolution. The work of Forrest et al. also examines system processes for anomalous behavior, however, their approach captures system calls from programs and uses a table look-up algorithm for detecting potential intrusions [Forrest et al., 1997, D'haeseleer et al., 1996]. An application of machine learning to intrusion detection has been developed elsewhere as well [Lane and Brodley, 1997]. Lane and Brodley's work is similar in that machine learning is used to distinguish between normal and anomalous behavior. However, their work is different in that they build *user* profiles based on sequences of each individual's normal user commands and attempt to detect intruders based on deviations from the established user profile. Rather than building profiles on a per-user basis, our work builds profiles of *software behavior* and attempts to distinguish between normal software behavior and malicious software behavior. The advantages of our approach are that vagaries of individual behavior are abstracted because program behavior rather than individual usage is studied.

The approach described here applies machine learning techniques to learn the normal behavior of a particular program in order to detect aberrations. By providing detection at the software process level, mul-

[1] `Bugtraq` can be viewed on-line at http://www.netspace.org/lsv-archive/bugtraq.html

tiple, diverse, and overlapping detectors can be embedded within the software infrastructure to provide system-wide coverage.

2 Employing Anomaly Detection Using Neural Networks

Desiring the ability to detect novel or unknown intrusions, the anomaly detection approach was employed in this research. Neural networks are used to learn the normal behavior of the monitored process and to detect potential intrusions against the software. The defining aspect of this approach is that anomaly detection is performed at the software process level using machine learning techniques.

Monitoring at the process level adds a layer of abstraction such that abnormal process behavior can be detected irrespective of individual users' behavior. This layer abstracts out users' individual behavior and allows anomaly detection against the set of all users' behavior. The approach taken in this research enables both observable external states such as program input/output as well as internal states that can be captured via instrumentation to be employed in training a neural network for detection of misuse. Thus, if the source code of the monitored program is not readily available, it does not preclude the use of neural networks for intrusion detection. The advantage of the black-box approach is that it can be used to detect malicious attacks against commercial software where the source code is unavailable. The added advantage of using a white-box–capable approach is that it allows access to internal states of a program which provides additional information in detecting anomalous program behavior. Differentiation between anomalous and normal behavior will occur by creating expectations about a program's usage and behavior and then attempting to detect deviations from the normal behavior.

Expectations of normal program behavior are created by dynamic analysis of the process under normal operational conditions. This approach differs from model-based approaches that specify either correct usage or, conversely, misusage of programs [Kumar and Spafford, 1996].

An architecture of the system for analyzing programs for malicious behavior is shown in Figure 1. The system was designed to enable different applications, neural networks, and test vectors to be used interchangeably. Thus, the architecture was constructed such that any of these components can be extracted and replaced with a different version without affecting the other components.

The approach begins with training neural networks with supervision; *i.e.*, feedback is applied to the network during training to indicate whether the input is normal or anomalous. Training with supervision requires labeled input, so it is not feasible for on-line detection of intrusions. However, in cases where historical or archival data exists, it can be useful for training a network to learn normal behavior and in some cases for recognizing non-normal behavior. This approach to anomaly detection permits programmatic detection of intrusions based on deviations of normal behavior without the need for a detailed specification model of correct behavior.

During recall, or during the on-line operation mode of the tool, the neural network classifies inputs and internal states as either anomalous or normal. Anomalous inputs and anomalous internal states are assume to be indicators of potentially dangerous behavior. The approach has proven to be easily modifiable to monitor other processes than those already tested.

One drawback of this approach is that the training period of the neural network may take on the order of hours or days to complete. However, other approaches that build up user profiles suffer from this same problem. Another pitfall to be aware of is that the network is only as good as the data on which it has been trained. It not known what effect different training sets would have on the results presented in this paper. The final drawback of our implementation is that if new data is added to the training set, the neural network has to be re-trained over the entire training set, instead of just the data that was added to the training set.

2.1 Backpropagation network

The backpropagation network, or backprop, is probably the most commonly used neural network. The standard architecture consists of an input layer, at least one hidden layer (neurons that are not directly connected to the input or output nodes), and an output layer. Typically, there are no connections between neurons in the same layer or to a previous layer.

Backprops have generalized capability. As such, they can produce nearly correct outputs for inputs that were not used in the training set. In theory, no more than two hidden layers are needed in a neural network since the network can generate arbitrarily complex regions in the state space [Lippmann, 1991]. One of the backprop's main drawbacks, though, is that it tends to be very computationally complex and is time consuming to train. Backprops are well-suited for applications in classification, function approximation, and prediction [Jain et al., 1996]. For the purposes of intrusion detection, the backprop is well-suited for

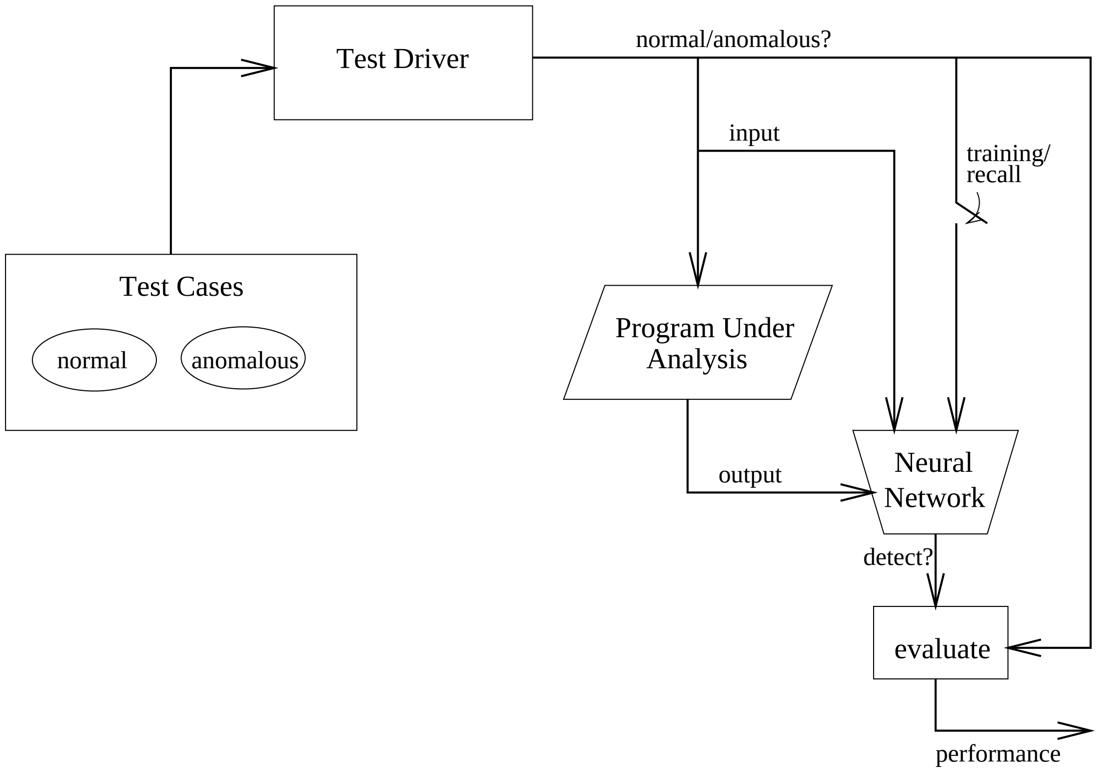

Figure 1: An architecture for analyzing programs for anomalous or malicious behavior using neural networks.

classifying normal behavior and flagging anomalous behavior.

The training cycle of a backprop proceeds in two distinct phases. First, the input is submitted to the network and the activations for each level of neurons are cascaded forward. In the training phase, the desired output is compared with the network's output. For instance, in training with supervision, the desired output for anomalous data is an intrusion detection. If the vectors do not agree, the network updates the weights starting at the output neurons. Then, the change in weights is calculated for the previous layer. This process continues to cascade through the layers of neurons toward the input neurons, hence the name backpropagation.

3 Neural Network Implementation

The backprop implementation provided many advantages in this work. Backprop networks are very good at classifying complex relationships, which in the case of anomaly detection, is useful for classifying normal and anomalous states.

The generalized backprop neural network is shown in Figure 2. The input layer of the network governs the number of inputs and internal states that the network uses in classification. Likewise, the output nodes govern the total number of classes the network is classifying. The backprop is trained with supervision; thus, the desired outputs for each input pattern is supplied to the network during the training phase.

4 Experimental Analysis

The objective of the experimentation described in this section is to determine how effective the implemented neural network is at detecting misuse of programs. The approach examines program inputs as well as internal states to determine if the program is being mis-used.

The experiment was designed to detect potential misuse of system programs, such as `lpr`. The Linux `lpr` program can be subverted on certain platforms using a buffer overflow attack. It should also be noted that this experiment was performed using both black-box as well as white-box analysis because access to the source code for the `lpr` program was freely available. This experiment was run repeatedly using different initial weightings of the neural network in order to account for potential statistical out-liers in the results.

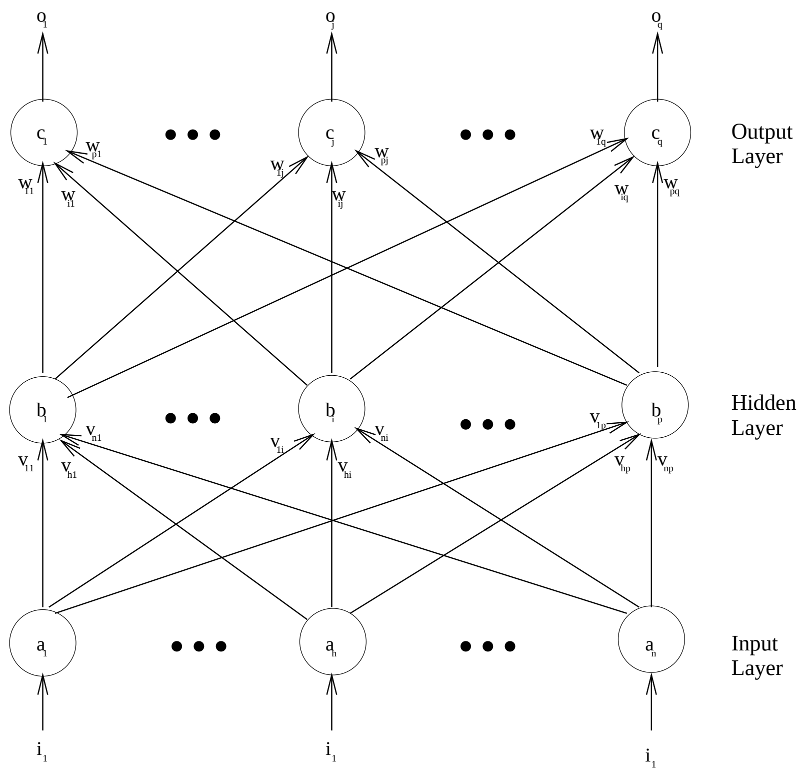

Figure 2: Topology of the elementary backpropagation network architecture with one hidden layer, **n** input nodes, and **q** output nodes.

5 Linux `lpr` Exploitation Experimental Results

5.1 Introduction

This set of experiments explores the use of a neural network to detect anomalous use of the standard Linux `lpr` program. The linux `lpr` program is known to have a buffer overflow vulnerability which can be exploited through input. This exploit uses a buffer overflow attack on specific `lpr` flags to enable the user to execute a root shell or perform other root-privileged commands. The goal of this experimentation is to determine how effective the neural network is in detecting anomalous use of system programs. To this end, the neural network was trained with malicious, normal, and in some cases, random input. This set of experiments also examines the use of internal states in the process of anomaly detection.

5.2 General description

The `lpr` program is exploited in the following manner. An auxiliary program was written which filled a large character buffer. At the end of this buffer, specific character strings were added to overwrite the return address of the stack frame with the address of a new instruction to be executed, which was placed elsewhere in the buffer (this instruction was usually a /bin/sh). The program would then `exec` an `lpr` child process with this buffer as one of the flag arguments. During the experimentation, both this auxiliary program and the `lpr` program were modified to write specific inputs and internal states to a file to be used as inputs to the neural network.

The neural network's input patterns consisted of a combination of inputs from two distinct sources. The first source was the character buffer passed to the `lpr` program as a flag argument. To encode the character buffer, only the last 75 positions were used. This was done in an attempt to reduce the difference between a normal flag input, typically on the order of 10-30 characters, and an overflow buffer of approximately 4,000 characters. Thus, for the overflow attempts, only the last 75 of the 4,096 characters were used as inputs, and for non-overflow attempts, the entire string was used (unless of course it exceeded 75 characters). To encode these 75 input characters, they were given the value: (integer value of character) / 128.0, and 0.0 was used to pad any of the unused inputs.

The second source of input was the internal `len` variable in the lpr card() function. The card() function was chosen because most of the lpr flag inputs are funneled through this particular procedure. The len variable is interesting because it represents the length of the input parameter. The len variable from card() constituted an additional 8 inputs (since the procedure could be executed multiple times), and was encoded by: (len value)/1032, where 0.0 was used to pad any of the unused inputs.

5.3 Training sets

The input patterns for the neural networks' training sets were constructed from the following sets: malicious inputs, normal lpr inputs, and random inputs. Malicious inputs were generated using the exploit program mentioned above. The malicious inputs did not necessarily generate /bin/sh, but were generated to look very similar. The normal lpr inputs were generated by printing valid files, with various flag options. The random inputs were generated using the fuzz program [Miller et al., 1995] and were limited to a length of 80 (only 75 characters could be used).

5.4 Recall set

To test the performance of each of the neural networks trained below, one recall set was created for all lpr experiments. The recall set consisted of 150 normal lpr inputs, 50 malicious inputs, and 50 random inputs. *All of these input patterns were unique from those used in the training sets.* The recall sets for the different networks were only modified in the experiments to include the specific subset of input patterns that were necessary for a particular network. Hence, if the internal state was not used to train the network, then the inputs corresponding to the len variable would also not be clamped to the network during the recall phase.

5.5 Experiments

The neural network used in the lpr exploit experiments was a 3 layer backpropagation network whose architecture consisted of a hidden layer, an input layer, and an output layer. There were 125 nodes in the hidden layer, 1 output node, and a variable number of input nodes.

Table 1 summarizes the experimental setup for the six experiments. The inputs to the neural network (NN) are distinguished between the experiments by whether the inputs were external lpr inputs or internal lpr states. External inputs represent standard input to the lpr program. If the len variable was used as an input to the neural network, then the *internal* column is checked in the table. The number of input nodes, hidden nodes, and output nodes for each of the neural network experiments is also given in the table. The other distinguishing parameter between the experiments is whether random input was used to train the network. If random data was used in training the network, then this column is checked in the table. The neural network was trained to classify random data as anomalous, since the network attempts to distinguish between normal use and anomalous use of a program. The goal of the experimentation is to determine which of these parameters are most useful (or conversely least useful) for detection of misuse of a program.

5.5.1 Discussion of results

Table 2 shows the results from all six experiments. All experiments were run 30 times using 30 different initial weights of the network. The performance of the network is evaluated based on the percentage of the inputs it classified correctly, the percentage of false positives, and the percentage of false negatives. A false positive is defined as a normal input that was classified as anomalous by the network. A false negative is defined as an anomalous input that was classified as a normal by the network. Either of these two distinctions would constitute an error in classification by the network. The results are presented as averages, minima, and maxima over these runs in these categories.

To summarize the results, the best results were obtained in Experiments 3 and 4 when the neural network was trained with random data generation. Recall, that the neural network was trained to classify random data as anomalous. Including the internal state variable len in addition to the external lpr inputs did not impact the results significantly. In fact, in Experiments 5 and 6, where the neural network was trained and tested on the internal len variable exclusively, the results were the weakest. Arguably, training with the internal len variable exclusively was too narrow for detection of a range of anomalous and malicious usage.

The neural network in Experiment 6 did not converge to an acceptable mean squared error, so its results were omitted. In Experiment 5, the error rate was approximately 20%—exclusively false negatives. In none of the experiments were false positives detected. Since the training was heavily biased with normal inputs, in no cases did the neural network incorrectly classify a normal input as an anomalous input (the false positive case).

As a simple benchmark for comparison, consider a monkey instead of a neural network that is choosing whether a given input is normal or anomalous. The monkey uses a simple algorithm for determining

	NN Inputs		NN Layers			
Num	**external**	**internal**	**Inputs**	**Hidden**	**Outputs**	**Random Data Included**
1	x		75	125	1	
2	x	x	83	125	1	
3	x		75	125	1	x
4	x	x	83	125	1	x
5		x	8	125	1	
6		x	8	125	1	x

Table 1: Experimental setup for detection of anomalous use of `lpr` program.

whether the input is normal or anomalous—he flips a coin. The likelihood that the monkey will classify an input as normal is 50% as is the likelihood that an input will be classified as anomalous. The actual inputs sent to the program have the following prior probabilities: 40% were anomalous while 60% were normal. The probability that the monkey commits a false negative is the probability that an anomalous input was sent to the program and that the monkey said it was normal. Since these are independent events, the false negative probability is 20%. The probability that the monkey commits a false positive is the probability that a normal input is sent and that the monkey said it was anomalous. The false positive probability is calculated similarly and is 30%. The results from using a neural network on the whole do better than the monkey. No false positives were detected (compared to 30% for the monkey), while false negatives for the neural network ranged from 0.4% to 19.9% (compared to 20% for the monkey). For some experiments (*e.g.*, Experiments 1, 2, and 5), the neural network did as bad as the monkey for false negatives (these are explained in the discussion below), while in others it did extremely well compared to the monkey (*e.g.*, Experiments 3 and 4).

In intrusion detection systems, reducing the false positive rate is perceived as a greater responsibility than reducing the false negative rate. The reasoning is that an excessive false positive rate can lead to a "cry wolf" syndrome where nobody pays attention to the intrusion detection software after awhile. On the other hand, provided multiple, overlapping detectors, a small false negative rate is tolerable as missed attempts by one detection unit may be correctly classified by another. A discussion of the results from each of the experiments is presented next.

Experiment 1 This experiment can be considered the baseline for the `lpr` exploit experiments (see Table 1). It demonstrates how well the neural network performed when training only on the accessible `lpr` inputs, a reasonably sized test set, and without training randomly generated input. From Table 2 we see that the baseline error rate was 20%, exclusively composed of false negatives.

Experiment 2 This experiment extends the baseline experiment, (Experiment 1), by not only using the accessible inputs to the `lpr` program, but also the internal state (represented by the `len` variable in the `card()` function). This experiment attempts to determine if by using an internal program state whether the performance of detection can be improved. The results do not indicate any significant impact, let alone improvement. The results are too similar to Experiment 1 to be statistically significant.

Experiment 3 The network was trained with random inputs to further diversify its notion of anomalous inputs. The idea is to expose the network to a number of different types of anomalous input, rather then strictly well-known intrusion attempts. Training with randomly generated inputs may be one way of detecting novel intrusion attempts. The performance of the network was excellent in this experiment. The average error rate for this experiment was only 0.9%. We can conclude that including randomly generated patterns in the training set vastly increased the performance of the network allowing the network to correctly classify a wide range of input patterns with a very high degree of accuracy. The results indicates that an anomaly detection approach may be useful over misuse detection in detecting novel, unknown intrusion attempts—especially when trained with random data classified as anomalous.

Experiment 4 This experiment extends Experiment 3, by including the internal state variable `len`. Once again, the goal is to determine if by adding in internal state information, whether the performance of the neural network will be improved. As seen from the Table 2, adding the internal variable once again did not statistically impact the results.

	Classified Correct			False Positive			False Negative		
Num	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
1	82.5%	80.1%	88.4%	0.0%	0.0%	0.0%	17.5%	11.6%	19.9%
2	81.9%	80.1%	85.7%	0.0%	0.0%	0.0%	18.1%	14.3%	19.9%
3	99.1%	98.0%	99.6%	0.0%	0.0%	0.0%	0.9%	0.4%	2.0%
4	98.8%	97.6%	99.6%	0.0%	0.0%	0.0%	1.2%	0.4%	2.4%
5	80.2%	80.2%	80.2%	0.0%	0.0%	0.0%	19.8%	19.8%	19.8%
6									

Table 2: Results of detection from anomalous use of `lpr` program.

Experiment 5 This experiment uses the internal state exclusively as the input to the neural network during training and recall. The results were the worst of all the experiments, but not significantly worse than experiments 1 and 2. A closer look at the data revealed that on recall, all the anomalous data randomly generated were classified as normal by the network (the false negative case). This contributed to the large false negative rate for this experiment. The reason is that the randomly generated data had a length approximately close to that of the normal training set. On the other hand, the network did very well in correctly classifying all anomalous input that were buffer overflows. Using strictly the `len` variable is effective at detecting buffer overflows, but not in detecting other potential misuses of a program.

The problem stems from the fact that the single internal variable chosen does not provide sufficient information to classify the input patterns. Inspecting the training set reveals that all of the anomalous input patterns are exactly the same, when one is only looking at the internal variable `len`. Thus, it is no surprise that the network converged to a state in which only this input vector was classified as anomalous. To alleviate this problem, a few steps can be taken. The first is that additional internal variables could be used. A second is to pair this internal state with inputs, as was done in the earlier experiments. This experiment reveals one of the problematic areas encountered in this research, namely, identifying which information provides maximum utility for detection of anomalous behavior.

Experiment 6 The neural network in Experiment 6 did not converge to an acceptable mean squared error, so its results were omitted.

6 Conclusions

This paper describes novel work in using neural networks for detecting misuse of programs. Two important observations result from this work. First, the results demonstrate how misuse of programs can be detected using neural networks. The results indicate that training with randomly generated data lead to the best performance in detection of possible novel misuse attempts—an area in which most misuse detection approaches are weak. Furthermore, the results show the benefit of applying anomaly detection at the process level such that abnormal process behavior can be detected irrespective of individual users' behavior. This approach abstracts out users' individual behavior and allows anomaly detection against the set of all users' behavior.

The experiments also raise an interesting discussion about using internal program states in training the neural networks. Choosing useful internal states is in general a very difficult process. How a particular state is identified as being important is a time consuming task, usually relying heavily upon code inspection. The final experiment also provides a warning for not placing too much emphasis on a particular internal state, or a single input for that matter. The best approach seems to be to take a broad, yet representative sample of inputs and internal states. Further research will attempt to identify important internal states for anomalous use of programs. These results may be important in not only detecting attempted misuse of programs, but also erroneous program behavior.

References

[D'haeseleer et al., 1996] D'haeseleer, P., Forrest, S., and Helman, P. (1996). An immunological approach to change detection: Algorithms, analysis and implications. In *IEEE Symposium on Security and Privacy*.

[Forrest et al., 1997] Forrest, S., Hofmeyr, S., and Somayaji, A. (1997). Computer immunology. *Communications of the ACM*, 40(10):88–96.

[Garvey and Lunt, 1991] Garvey, T. and Lunt, T. (1991). Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*.

[Ilgun, 1992] Ilgun, K. (1992). Ustat: A real-time intrusion detection system for unix. Master's thesis, Computer Science Dept, UCSB.

[Jain et al., 1996] Jain, A., Mao, J., and Mohiuddin, K. M. (1996). Artificial neural networks: A tutorial. *IEEE Computer*, 29(3):31–33.

[Kumar and Spafford, 1996] Kumar, S. and Spafford, E. (1996). A pattern matching model for misuse intrusion detection. The COAST Project, Purdue University.

[Lane and Brodley, 1997] Lane, T. and Brodley, C. (1997). An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, pages 366–377.

[Lippmann, 1991] Lippmann, R. (1991). *An Introduction to Computing with Neural Nets*, chapter Part 1, pages 5–23. IEEE Press, Piscataway, NJ. in Neural Networks Theoretical Foundations and Analysis.

[Lunt, 1990] Lunt, T. (1990). Ides: an intelligent system for detecting intruders. In *Proceedings of the Symposium: Computer Security, Threat and Countermeasures*. Rome, Italy.

[Lunt, 1993] Lunt, T. (1993). A survey of intrusion detection techniques. *Computers and Security*, 12:405–418.

[Lunt and Jagannathan, 1988] Lunt, T. and Jagannathan, R. (1988). A prototype real-time intrusion-detection system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*.

[Lunt et al., 1992] Lunt, T., Tamaru, A., Gilham, F., Jagannthan, R., Jalali, C., Javitz, H., Valdos, A., Neumann, P., and Garvey, T. (1992). A real-time intrusion-detection expert system (ides). Technical Report, Computer Science Laboratory, SRI Internationnal.

[Miller et al., 1995] Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A., and Steidl, J. (1995). Fuzz revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept.

[Monrose and Rubin, 1997] Monrose, F. and Rubin, A. (1997). Authentication via keystroke dynamics. In *4th ACM Conference on Computer and Communications Security*.

[Porras and Kemmerer, 1992] Porras, P. and Kemmerer, R. (1992). Penetration state transition analysis - a rule-based intrusion detection approach. In *Eighth Annual Computer Security Applications Conference*, pages 220–229. IEEE Computer Society Press.

[Porras and Neumann, 1997] Porras, P. and Neumann, P. (1997). Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365.