A Real-Time Intrusion Detection System Based on Learning Program Behavior

Anup K. Ghosh, Christoph Michael, and Michael Schatz

Reliable Software Technologies
21351 Ridgetop Circle, #400
Dulles, VA 20166 USA
{aghosh, ccmich, mschatz}@rstcorp.com
http://www.rstcorp.com

Abstract. In practice, most computer intrusions begin by misusing programs in clever ways to obtain unauthorized higher levels of privilege. One effective way to detect intrusive activity before system damage is perpetrated is to detect misuse of privileged programs in real-time. In this paper, we describe three machine learning algorithms that learn the normal behavior of programs running on the Solaris platform in order to detect unusual uses or misuses of these programs. The performance of the three algorithms has been evaluated by an independent laboratory in an off-line controlled evaluation against a set of computer intrusions and normal usage to determine rates of correct detection and false alarms. A real-time system has since been developed that will enable deployment of a program-based intrusion detection system in a real installation.

1 Introduction

Today, most commercial intrusion detection systems monitor network packets for unusual patterns, or patterns of known suspicious actions. Recent advances in high bandwidth local area networks have presented significant challenges to performing network monitoring in real-time. In addition, as end-to-end encryption protocols are adopted enterprise wide, many of today's network-based intrusion detection systems will be rendered obsolete.

Host-based intrusion detection systems attempt to detect computer intrusions by monitoring audit trails created on host computer systems. Many modern day operating systems provide audit trails for processes that run on the machine. On the Solaris platform, the Basic Security Module (BSM) provides a configurable audit manager that facilitates recording system events requested by executing processes.

We leverage this audit reporting mechanism in this research. The motivation for our work is that a large class of computer intrusions involves program misuse. Most program misuse attacks exploit privileged programs in clever ways in order to gain unauthorized privileges that are subsequently used to commit malicious acts of sabotage or data theft. Buffer overrun attacks are the most frequent form of program misuse attacks. Other types of program misuse attacks include using

rarely used features (such as debug features), exploiting race conditions, and triggering Trojan horse functionality in order to gain higher privileges.

When a program is misused, its behavior will differ from its normal usage. Therefore, if the normal range of program behavior can be adequately and compactly represented, then behavioral features captured by audit mechanisms can be used for intrusion detection.

A well-recognized failing of today's commercial intrusion detection systems is that they cannot detect novel attacks against systems, and they often fail to detect variations of known attacks. The reason is that most commercial intrusion detection systems detect attacks by matching audit events against well-known patterns of attacks. This approach is known as signature-based detection. The problem with a signature-based detection approach is that it is reactive by nature. Once a new form of intrusion is developed, it is often perpetrated against many systems before its signature is captured, codified, and disseminated to individual detection sensors. In a worm-type of infection, millions of machines can potentially be compromised before a signature-based system can be upgraded with the appropriate signature.

To detect novel attacks against systems, we develop anomaly-based systems that report any unusual use of system programs as potential intrusions. The advantage of this approach is that both known attacks and novel attacks are detected. The disadvantage is that if the training mechanism for the detection sensor is not robust, a large number of false alarms may be reported. In other words, perfectly legitimate behavior may be reported as intrusions.

Another large challenge in intrusion detection is to generalize from previously observed behavior (normal or malicious) to recognize similar future behavior. This problem is acute for signature-based misuse detection approaches, but also plagues anomaly detection approaches that must be able to recognize future normal behavior that is not identical to past observed behavior, in order to reduce false positive rates.

In the research reported here, we address both challenges: detecting novel attacks as well as generalizing from previously observed behavior in order to reduce the false positive rate to acceptable levels from an administration standpoint.

We develop an anomaly detection system that uses machine learning automata to learn the normal behavior for programs. The trained automata are then used to detect possibly intrusive behavior by identifying significant anomalies in program behavior. The goal of these approaches is to be able to detect not only known attacks and but also detect future novel attacks using off-the-shelf auditing mechanisms provided by the operating system vendor.

We develop three algorithms for learning program behavior profiles and detecting significant deviations from these profiles. The algorithms were evaluated by an independent laboratory in a controlled off-line experiment to determine their effectiveness against program misuse attacks. The performance of the algorithms is presented as a measure of the probability of correct detection against the probability of false alarm.

Finally, in Section 5, we describe a real-time system that implements one of the learning algorithms to detect intrusions in real-time.

2 Related Work

Analyzing program behavior profiles for intrusion detection has recently emerged as a viable alternative to user-based approaches to intrusion detection (see [11, 18, 14, 6, 4, 7, 16, 2] for other program-based approaches). Program behavior profiles are typically built by capturing system calls made by the program under analysis under normal operational conditions. If the captured behavior represents a compact and adequate signature of normal behavior, then the profile can be used to detect deviations from normal behavior such as those that occur when a program is being misused for intrusion.

For a detailed comparison of our general approach to program-based intrusion detection with those of others in this area, please see [10].

3 Three Machine Learning Algorithms for Anomaly Detection

As described in the introduction, we are interested in detecting novel attacks against systems by detecting deviations from normal program behavior. To this end, we have developed three machine learning algorithms to train automata to learn a programs' normal behavior. The trained program automata are subsequently used to detect program misuse. The three algorithms are: an Elman recurrent artificial neural network, a string transducer, and a finite state tester. Each algorithm is described next.

3.1 Elman Recurrent Neural Network

The goal in using artificial neural networks (ANNs) for anomaly detection is to be able to generalize from incomplete data and to be able to classify online data as being normal or intrusive. An artificial neural network is composed of simple processing units, or *nodes*, and connections between them. The connection between any two units has some *weight*, which is used to determine how much one unit will affect the other. A subset of the units of the network acts as *input nodes*, and another subset acts as *output nodes*. By assigning a value, or *activation*, to each input node, and allowing the activations to propagate through the network, a neural network performs a functional mapping from one set of values (assigned to the input nodes) to another set of values (retrieved from the output nodes). The mapping itself is stored in the weights of the network.

We originally employed ANNs because of their ability to *learn* and *generalize*. Through the learning process, ANNs develop the ability to classify inputs from exposure to a set of *training inputs* and application of well defined *learning rules*, rather than through an explicit human-supplied enumeration of classification rules. Because of their ability to generalize, ANNs can produce reasonable

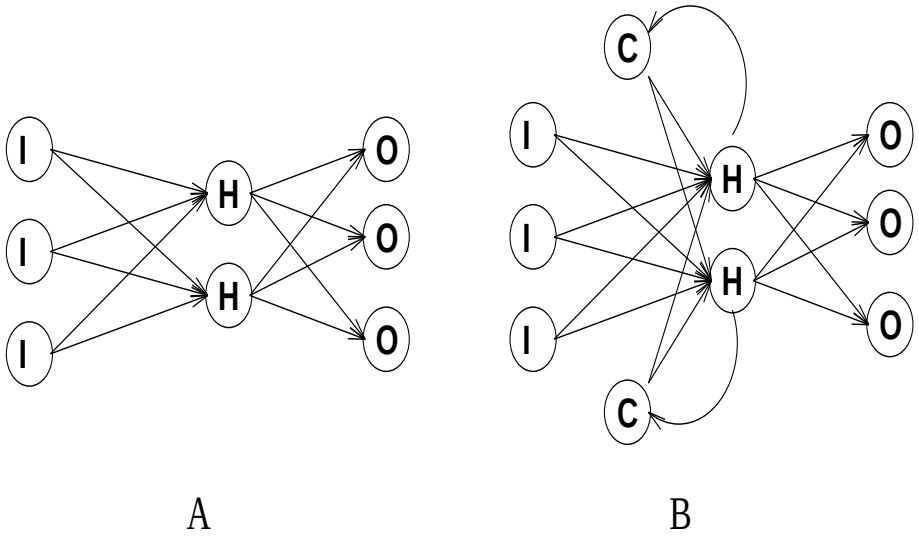


Fig. 1. In each of the examples above, the nodes of the ANNs are labeled as input nodes (*I*), hidden nodes (*H*), output nodes (*O*), or context nodes (*C*). Each arc is unidirectional, with direction indicated by the arrow at the end of the arc. A) A standard feed-forward topology. B) An Elman network

classifications for novel inputs (assuming the network has been trained well). Further, since the inputs to any node of the ANN used for this work could be any real-valued number, no sequence of BSM events could produce an encoding that would fall outside of the domain representable by the ANN.

In order to maintain state information between inputs, we require a recurrent ANN topology. A recurrent topology (as opposed to a purely feed-forward topology) is one in which cycles are formed by the connections. The cycles act as delay loops—causing information to be retained indefinitely. New input interacts with the cycles, affecting both the activations propagating through the network and the activations in the cycle. Thus, the input can affect the state, and the state can affect the classification of any input.

One well known recurrent topology is that of an Elman network, developed by Jeffrey Elman [5]. An Elman network is illustrated in Figure 1. The Elman topology is based on a feed-forward topology—it has an *input layer*, an *output layer*, and one or more *hidden layers*. Additionally, an Elman network has a set of *context nodes*. Each context node receives input from a single hidden node and sends its output to each node in the layer of its corresponding hidden node. Since the context nodes depend only on the activations of the hidden nodes from the previous input, the context nodes retain state information between inputs.

We employ Elman nets to perform classification of short sequences of events as they occur in a larger stream of events. Therefore, we train our Elman networks to *predict* the next sequence that will occur at any point in time. The nth

input, I_n, is presented to the network to produce some output, O_n. The output O_n is then compared to I_{n+1}. The difference between O_n and I_{n+1} (that is, the sum of the absolute values of the differences of the corresponding elements of O_n and I_{n+1}) is the measure of anomaly of each sequence of events. Or, in other words, the anomaly measure is the error in predicting the next input in sequence. The classification of a sequence of events will now be affected by events prior to the earliest event occurring within the sequence.

3.2 String Transducer

A string transducer is an algorithm that associates a sequence of input symbols with a series of output symbols. String transducers are most often used in computational biology and computational linguistics, where they are usually implemented using finite automata whose transitions or states are associated with output symbols. In the current context, we use automata as well, but the input sequence is a string of BSM events, and the output sequence is a prediction for the next several events.

Our use of string transducers as intrusion detectors is based on an examination of the *probabilities* of the output symbols at each state. During training, we estimate the probability distribution of the symbols at each state, and during testing, deviations from this probability distribution are taken as evidence of anomalous behavior.

Our implementation of this idea is relatively simple. We use a finite automaton whose states correspond to n-grams in the BSM data, and the output symbols associated with each state are also BSM ℓ-grams (for $\ell < n$). More specifically, the output symbol represents sets of ℓ BSM events that may be seen when the automaton is in a given state. During training, our goal is to gather statistics about these successor ℓ-grams; we estimate the probability of each ℓ-gram by counting.

During actual intrusion detection, the deviation of the successor ℓ-grams from their expected values are used for anomaly scores. Of course, the anomaly scores are usually non-zero, but if the program is behaving normally these deviations should average out over time.

In the ideal case, it can be shown that the anomaly scores are uncorrelated if the probability distributions have, in fact, been correctly estimated (this is due to the fact that the deviations are then an innovations process; see [1]). That means that if we subtract the mean anomaly score for each state from the actual anomaly scores generated there, the result is zero-mean white noise.

If these values are integrated over a sufficiently long period, the result should be close to zero if the program is behaving normally. However, if abnormal program behavior results in a significant deviation of the successor ℓ-grams from their expected values, then the resulting scores will not integrate to zero, and this fact can be used to detect anomalous behavior.

In practice, there are obviously a number of factors preventing the realization of this ideal case:

1. If the probabilities of the successor ℓ-grams have not been correctly estimated, then the deviations may not be uncorrelated.
2. During detection, n-grams may be encountered that do not correspond to any known state because they were not seen during training.
3. An intrusion may not result in a systematic deviation from the expected ℓ-gram values; in other words, the intrusion may look normal. Although this seems unlikely, we cannot prove that all intrusions really cause the necessary deviations.
4. The window of integration needed to get sufficiently low anomaly scores during normal behavior may be large. This delays the detection of anomalies (though if it prevented them from being detected we would arguably be in case 3).

The fourth is an intrinsic problem of change detection [15]; there is an inevitable tradeoff between the time to detection and the susceptibility to false positives. The third problem is also, in some sense, unavoidable; it seems unlikely that we could guarantee the detection of all intrusions without assuming something about the nature of those intrusions, which is contrary to our assumptions. (We may, of course, be able to make guarantees for certain classes of intrusions).

The second problem cited above is more directly related to our specific application. It results from having too little training data to characterize all states. It dictates that states should not be too highly specialized, since such specialization makes it less likely for all states to be seen during training.

The first problem dictates a wise choice of states. For example, it has been observed that programs go through different phases of behavior [3], so the probability of a given ℓ-gram may depend on how far along the program is in its execution. Thus, states should reflect the state of the program itself. Even if the distribution of ℓ-grams varies over time, the distribution *from a given state* should be constant. Unfortunately, this condition can be best achieved by using highly specialized states to avoid having two or more states of the underlying program represented by a single state of the automaton. Thus, the solutions to the first and second problems are in some sense at odds. This tradeoff between expressiveness and ease of training is also well-known in machine learning [19].

As we have said, the probability densities of the successor ℓ-grams in a given state are estimated by counting (that is, we simply count the number of occurrences of each ℓ-gram in the training data). This approach is feasible with BSM data because it tends to be fairly regular; the number of BSM ℓ-grams is much smaller than, say, the number of possible BSM events raised to the ℓth power.

We measure deviations from expected behavior by treating the estimated probability distribution as a vector, which we first normalize with respect to the L_k metric,

$$\left| \sum_i x_i^k \right|^{1/k},$$

for some k. When a given ℓ-gram occurs during detection, we treat it as a vector with a 1 in the position corresponding to the actual ℓ-grams that were seen,

and a 0 in the other positions. The deviation is proportional to the L_k distance between this vector and the normalized density vector. In other words, if $\hat{p}_i$ is the estimated probability of the ith ℓ-gram, according to some arbitrary ordering, then the elements of the normalized probability vector are given by

$$h_i = \frac{\hat{p}_i}{\left|\sum_j \hat{p}_j^k\right|^{1/k}},$$

and the deviation d_i, reported when the ith ℓ-gram is seen during detection, is given by

$$d_i = 2^{-1/k} \left(\sum_j c_{i,j}^k \right)^{1/k}$$

where

$$c_{i,j} = \begin{cases} 1 - h_j, & \text{if } i = j; \\ h_j, & \text{otherwise.} \end{cases}$$

We treat these as summations over all possible ℓ-grams, though the actual implementation only has to sum over those that were seen during training since p_j is zero for the others. But if a novel ℓ-gram is seen during testing, this convention assures that d_i is still defined, and, in fact, its value is just 1.

3.3 State Tester

The goal of the third algorithm, we call simply a state tester, is to automatically create finite automata to represent program behavior. Since data representing intrusive behavior is not used during training, the first goal is simply to build a finite automaton that accepts all audit sequences in the training data, but without being so generous that it accepts *all* data, or being so rigid that it rejects every novel audit sequence after training.

In [13], finite automata (FA) of this kind were generated largely by hand. First, the BSM data was pre-processed so that commonly occurring sequences of events could be combined into a single meta-event. Then, the meta-events were encoded as an FA. The combination of events into meta-events, called *macros* in that paper, was done manually, and though the paper does not say whether the FAs were then also created by hand, it is implied that they were.

Our approach is to automate the process of inferring finite automata. Something along these lines is done in [3], where training data is used to learn hidden Markov models of normal program behavior. This technique proved effective at the task of intrusion detection, but training (using the Baum-Welch algorithm, see [17]) was found to be expensive. This raises the question of whether simpler algorithms that only infer an FA, and not the transition probabilities associated with a Markov model, might also be effective without requiring as much training.

Below, we present an algorithm for automatically constructing finite automata from training data. In this context, it should be noted that the inference of finite automata is not intractable, although the automatic inference of finite

automata *is* intractable in a number of other settings (C.f., [12, 9]). What makes the problem tractable in the case of anomaly detection is that the requirements are simple. The finite automaton merely has to accept any training sequence that is not abnormal. Of course, it should also reject abnormal BSM sequences, but since there are no abnormal BSM sequences in the training data this requirement cannot be formalized within the learning algorithm itself. Rather, we will evaluate the performance of the FAs empirically.

By way of example, we could create an FA with a single state, where every BSM event results in a transition from that state back to itself. We could also create an FA with no cycles that accepts exactly the BSM sequences occurring in the training data.

The first approach is too weak because it tends to accept *any* sequence of BSM events, and thus fails to notice abnormal BSM sequences. The second approach is probably too strong, because it rejects any sequence as being abnormal unless exactly the same sequence was seen during training. Our goal is to create a reasonably expressive FA, but one that can still generalize. Of course, this is a qualitative requirement.

The first issue is how to define the states of the automaton. The technique reported in this paper associates each state with one or more n-grams of BSM data, where n is a parameter of the learning algorithm. For example, the FA might have a state corresponding to the event sequence `lstat, open, ioctl`, and enter that state whenever the sequence `lstat, open, ioctl` is seen. The idea, however, is to be parsimonious in the creation of new states, and not simply have one state in the FA for every n-gram of BSM events. Instead, we will have more than one n-gram assigned to most of the states.

During training, separate automata are created for the different programs whose audit data are available for training. As with the intrusion detection systems of [8], the training algorithm is presented with a series of n-grams taken from non-intrusive BSM data for a given program. Conceptually, the goal of the automaton is to predict the entire n-gram based on the automaton's current state and on the first ℓ audit events in the n-gram, $\ell < n$.

The FA's transitions correspond to specific sequences of ℓ audit events, and each state corresponds to one or more n-grams. We say that the FA predicts an n-gram G if there is a transition from the current state to the state corresponding to G, and if that transition is labeled with the first ℓ elements of G. Thus, the automaton predicts a set of states, and these states are simply the ones reachable by transitions labeled with the first ℓ elements of G. If this set is empty (*e.g.*, there is no transition labeled with the first ℓ elements of G) then we say that the FA makes no prediction at all. Otherwise, a *prediction error* occurs if the predicted set of states does not contain the one associated with G.

During training, an incorrect prediction results in the creation of a new transition and possibly a new state. The training algorithm starts with an FA having a single state and no transitions. We say that the FA is initially in this state. Whenever a new training n-gram is seen, there are three possibilities:

1. The current state has an outgoing edge that corresponds to the first ℓ events in the n-gram, and that edge leads to the correct state (the correct state is the state that is assigned to the newly obtained n-gram). In this case, the FA needs no modifications.
2. The current state has outgoing edges that correspond to the first ℓ events in the n-gram, but none of the edges lead to the correct state. In this case, the FA may contain a correct state (but no edge from the current state to the desired state), or else the FA may not even have any state assigned to the new n-gram.
   We simply create a state for the new n-gram if one doesn't already exist. In either case, we create a transition from the current state to the new state, and label that transition with the first ℓ events of the new n-gram (recall that we will use these ℓ events when trying to make future predictions).
3. The current state has no outgoing edges that correspond to the first ℓ events in the newly obtained n-gram. If there is already a state assigned to the newly obtained n-gram, then we simply create a transition to that state, and label it with the ℓ events as in the previous case.
   However, if the new n-gram doesn't have any state assigned to it, we can assign any one of the already existing states, or create a new state, without introducing any prediction errors. Currently, the algorithm just creates a transition back to the current state, and assigns the new n-gram to the current state (where it joins whatever n-grams were assigned to that state previously).

   In all three cases, the FA transitions to the state assigned to the new n-gram.

4 Performance of Algorithms

The three algorithms described in the preceding section were implemented and evaluated by an independent laboratory, Lincoln Laboratory of the Massachusetts Institute of Technology, in the 1999 U.S. Defense Advanced Research Projects Agency (DARPA) Intrusion Detection Evaluation. The full extent of the experimental setup, the data, the participants, system descriptions, full attack descriptions, raw scores, and results are available online at the Lincoln Laboratory's Intrusion Detection Evaluation page (http://ideval.ll.mit.edu). In this section, we summarize the results of our systems.

Lincoln established four categories of attacks: Denial of Service (DoS), probe, remote-to-local (R2L), and user-to-root (U2R). Within these categories they ran several select instances of attacks. Lincoln does not claim these attacks are comprehensive of the category of attacks. Rather, the attacks can be considered as samples from the attack space within a category. DoS and probe attacks were network-based attacks that leave traces in network packet data. Remote-to-local attacks involved network-based attacks again, but also included some attacks that attempted to misuse host-based programs. User-to-root attacks attempt to gain super user privileges on the host machine either by misusing programs or by running malicious software.

While our approach is not exclusive to any single category of attacks as partitioned by Lincoln, our approach is best suited to detect user-to-root attacks according to the Lincoln partitions. Our approach will detect program misuse attacks regardless of which of the four Lincoln categories the attacks falls in, as long as the attack leaves some trace in the audit data we use. In addition to the user-to-root attacks, a few instances of the remote-to-local attacks involved program misuse. So, we also include results from detecting remote-to-local attacks in this section.

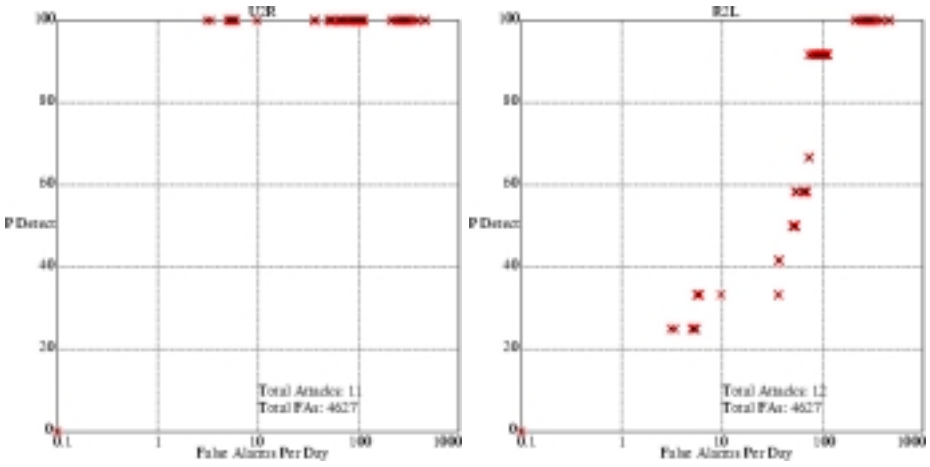
Table 1. List of programs monitored by intrusion detection automata

admintool	dhcpcd	kswapd	ping	sperl5.00404	wu.ftpd
allocate	dos	list_devices	procmail	ssh1	xlock
aspppd	eject	lockd	ps	sshd	xscreensaver
at	exrecover	login	pt_chmod	su	xterm
atd	fdformat	lpd	pwdb_chkpwd	suidperl	Xwrapper
atq	ff.core	lpq	rcp	syslogd	ypbind
atrm	ffbconfig	lpr	rdist	tcpd	yppasswd
auditd	fsflush	lprm	rdistd	timed	zgv
automountd	gpasswd	m64config	rlogin	traceroute	
cardctl	gpm	mingetty	routed	umount	
chage	hpnpd	mkdevalloc	rpcbind	uptime	
chfn	untd	mkdevmaps	rpciod	userhelper	
chkey	in.*	mount	rpld	usernetctl	
chsh	inetd	newgrp	rsh	utmp_update	
cron	kcms_calibrate	nispasswd	rusersd	utmpd	
crond	kcms_configure	nmbd	rwhod	uu.*	
crontab	kerbd	nscd	sacadm	volcheck	
ct	kerneld	nxterm	sadmind	vold	
cu	kflushd	pageout	sendmail	w	
deallocate	klogd	passwd	smbd	whodo	

Since our approach involves training program monitors, we must first choose which programs to monitor. Most attacks, in practice, are launched against privileged programs on network servers. So, our rule was to train program monitors on SUID root programs that run on Unix servers. Table 1 lists the programs we monitor for intrusions and also represents a superset of program monitors run against the Lincoln Laboratory data because not all programs in Table 1 are exercised by the Lincoln data.

4.1 Performance of Elman networks

Figure 2 shows the performance of the Elman networks on the BSM data against both U2R (Figure 2(a)) and R2L (Figure 2(b)) attacks. The plots are called Detection/False Alarm plots by Lincoln Laboratory. The plot shows the probability



(a) (b)

Fig. 2. Performance of Elman networks on BSM data against User-to-Root (U2R) and remote-to-local (R2L) attacks

of correct detection versus the false alarm rate per day. Examining the user-to-root attacks first, it becomes clear that the Elman networks performed very well against this class of attacks. The Elman networks achieved 100% detection of attacks very quickly at a false alarm rate of close to 3 per day. This false alarm rate is considered acceptable in an operational environment and is vastly superior to current commercial tools.

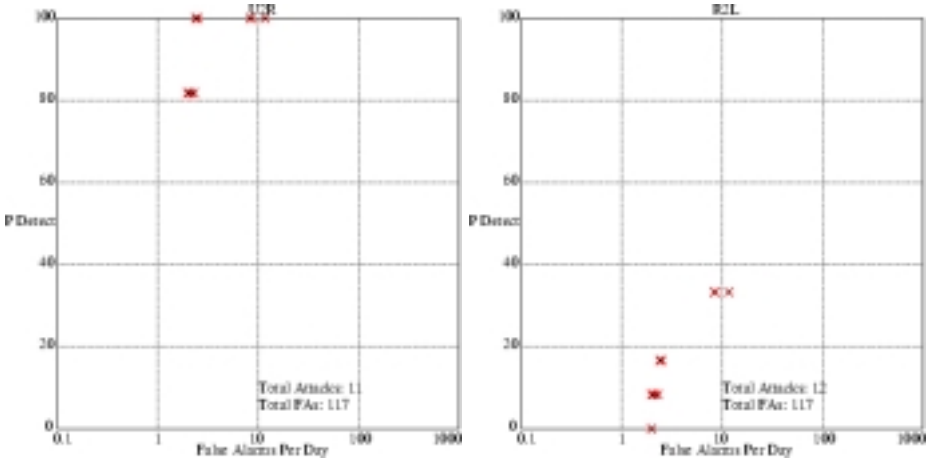
A closer examination of the attacks showed that the vast majority of them involved program misuse types of attacks such as buffer overrun attacks. However, our technique is not limited to buffer overrun attacks. Rather the approach is designed detect any program misuse attack. It turns out that the sample U2R attacks chosen by Lincoln were all buffer overrun attacks. As more different types of program misuse attacks are captured in evaluation sets, we will be able to verify this claim in the future.

The performance of the Elman networks against Lincoln's remote-to-local attacks was not nearly as good, as shown in Figure 2(b). At a rate of approximately 10 false alarms per day, we detected roughly 30 percent of R2L attacks. If you are willing to accept a false alarm rate of up to 100 per day, the correct detection rate goes over 90 percent. However, operationally speaking, that false alarm rate is not acceptable.

In the 1999 evaluation, the R2L attacks run by Lincoln by-and-large did not involve program misuse. Thus, most of these attacks fall outside the scope of our approach. For example, the `guessftp`, `ftpwrite`, and `guest` R2L attacks all involve using the legitimate protocol to either guess passwords or write files (when the program was configured to do so).

Other remote-to-local attacks involved malicious clients acting on behalf of an outside perpetrator. Since we only monitor programs we know about, we do not detect malicious programs. However, our technique can detect intrusions that may have been precursors to installing malicious clients. In summary, attacks that involve programs we do not monitor or attacks that involve normal uses of programs fall outside the scope of our detection mechanism. The reason we did end up detecting them at all (even at a high false alarm rate) is that side effects from the intrusion tend to show up in other programs we monitor, albeit at a high false alarm rate.

4.2 Performance of String Transducer



(a) (b)

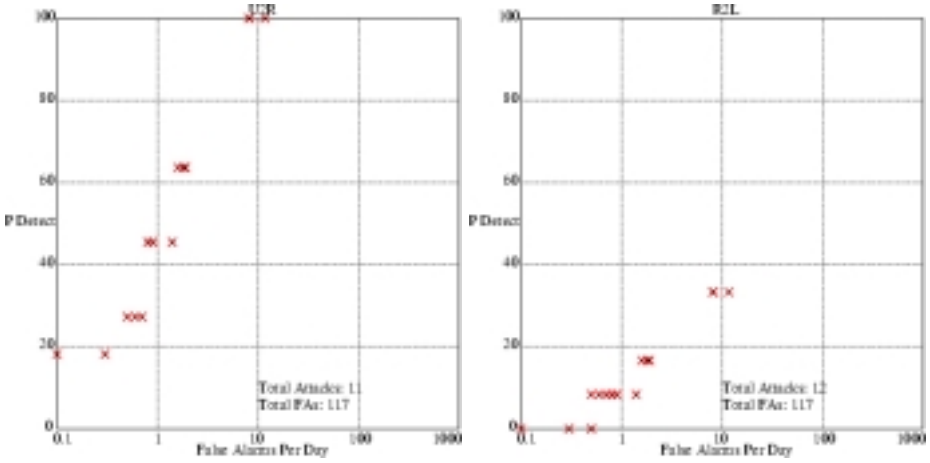
Fig. 3. Performance of string transducer on BSM data against User-to-Root (U2R) and remote-to-local (R2L) attacks

Figure 3 shows the performance of the string transducer against U2R and R2L attacks. The performance of the string transducer is very close to that of the Elman network. At a rate of about 3 false positives a day we detected 100% of the user-to-root attacks. What is most significant about this result, however, is that since the training time for the string transducer is orders of magnitude less than that of the Elman neural network, we can achieve comparable detection performance with significantly less training time. Where training the Elman nets takes on the order of thousands of minutes for all the programs monitored, training the string transducer and the state tester takes on the order of tens of minutes.

Again, the performance against the R2L attacks was not very good for the same reasons. At the same false positive rate we detected about 15% of the

remote-to-local attacks. If you raise the false positive rate to about 9 false positives a day we detected about 35% of the remote-to-local attacks. The reasons why our string transducer failed to detect many R2L attacks is the same as in the Elman network: most of the R2L attacks launched by Lincoln Laboratory did not misuse programs, or they involved malicious clients.

4.3 Performance of State Tester



(a) (b)

Fig. 4. Performance of state tester on BSM data against User-to-Root (U2R) and remote-to-local (R2L) attacks

The performance of the state tester is shown in Figure 4. At a rate of about 9 false positives a day we detected 100% of the user-to-root attacks. Figure 4a shows a more gradual progression towards 100 percent detection, compared to Figure 2a, whose progression looks more like a unit step function. The upshot is that with the state tester, one can tune the performance of the system more easily to meet the acceptable detection requirements within the organization's tolerance to false alarms. On the other hand, the performance of the Elman network indicates, more or less, all-or-nothing detection, which does not leave much to tune. However, it is important not to over generalize, as the results may vary from experiment to experiment depending on the attacks launched and the training data.

At a false alarm rate of about 9 per day, we detected about 35% of the remote-to-local attacks. While the state tester did not perform as well as the Elman networks or the string transducer, its performance is still good, nonetheless, by existing commercial standards. The reason we believe the state tester had a higher false alarm rate is because the likelihood of falling off the deterministic

automata is greater than for the string transducer or the Elman neural network. We believe, though, that its performance can be improved with more robust training data.

Overall, the performance of our systems on user-to-root attacks is good, roughly 100 percent detection at a rate of less than 10 false positives per day. Two of our systems, the Elman neural network and the string transducer, were able to detect all user-to-root attacks with fewer than four false alarms per day. This, combined with the fact that our systems can be trained in much less time than it takes to configure a rule-based intrusion detection system, makes our approach very promising.

Our systems did not fare as well on remote-to-local attacks, but this was because many of the remote-to-local attacks Lincoln launched did not involve program misuse. Thus, not all such attacks are in the scope of our approach. Conversely, our approach will be able to detect attacks that fall in other categories, so long as the attacks involve program misuse. Thus, the scope of our detection has more to do with how an attack affects program behavior then it has to do with other types of attributes. While we do not claim to detect all attacks, we do claim the scope of our detection mechanism to cover those attacks that misuse programs.

5 Implementing a Real-Time Intrusion Detection Tool

While, studying the performance of the algorithms off-line is a necessary step to understand the strengths and limitations of the algorithms, we felt it important to implement a real-time intrusion detection system that can be deployed in a real installation. In order to implement a real-time prototype, we performed a feasibility study, determined how to collect audit data in a real-time, modified our algorithms to work in a real-time environment, then designed and implemented a working prototype. These are described briefly in this section.

The first task in creating a real-time intrusion detection tool was to make sure that our approach was actually feasible in a real-time environment. In order to work in real-time, the intrusion detection prototype should be able to process audit data that is generated by a computer under normal use, as fast, or faster than the data is being generated. We measured this by collecting a set of audit data, and then measuring how long it took us to process that data off-line.

Our first approach was to use `praudit`, the built-in Solaris utility for translating binary BSM files to a text format, to translate the collected BSM files, and perform simple processing on the result. We did this because our off-line evaluation techniques processed `praudit` format data, and not BSM files directly. Next, an example of the results from real-time processing of BSM files is presented.

Amount of data processed:

- amount of BSM data: 8,195,371 bytes
- number of events: 48,871
- time frame that BSM data was collected over: 5 minutes 3 seconds

Amount of CPU time required:

- clock time: 14 minutes 57.79 seconds
- user cpu time: 5 minutes 20.98 seconds
- system cpu time: 6 minutes 38.27 seconds

As can be seen, processing this data took longer than it took the system to create the data. The solution to this problem was to not use `praudit`, but rather to process the binary BSM data directly. When we did this, processing the above data set gave us better timing results as shown below.

Amount of data processed:

- amount of BSM data: 8,195,371 bytes
- number of events: 48,871
- time frame that BSM data was collected over: 5 minutes 3 seconds

Amount of CPU time required:

- clock time: 1 minutes 48.12 seconds
- user cpu time: 0 minutes 10.99 seconds
- system cpu time: 1 minute 36.96 seconds

These results show that our approach is feasible to be implemented in real-time.

We had to make sure our intrusion detection algorithms were amenable to a real-time domain. This meant three things. First, the algorithms had to run fast enough. Second, they had to be able to process data as it is generated, and not require all of the audit data at the same time. Third, the algorithm had to be reentrant, meaning that it had to process multiple data streams simultaneously.

We chose the Elman networks as the first intrusion detection algorithm to implement in a real-time prototype. Neural networks perform recall quickly, so the first real-time requirement was already satisfied. The way that we use the Elman nets in the off-line evaluations was to process the data in order, so it already met the second real-time requirement as well. The third requirement was not met, because our implementations of the Elman Nets were in C, which meant that only one instance would exist at a time. This was not satisfactory because in a real-time environment it is possible to have multiple copies of the same program being run at the same time, and it is important that each execution is evaluated by its own neural net. To solve this problem, we modified the Elman networks so that they were implemented as C++ objects, allowing multiple instantiations to exist simultaneously. This satisfied our third real-time requirement.

Our last step was to actually design and implement a real-time prototype. This was a straight forward software engineering task. We designed the prototype such that it is modular enough to incorporate the other intrusion detection algorithms in a plug-and-play manner. We reviewed it to make sure that it would achieve our immediate goals of having something to use for internal testing and for ease of modification.

The initial prototype has now been implemented and is in testing. A demonstration of the real-time prototype has been created as well. In the process of creating the real-time prototype, we created a library called BSMart ("be smart") to parse BSM data directly from the operating system in binary form in any number of configurable ways. Because we deem this library to be a valuable contribution to the ID community wishing to perform host-based intrusion detection on the Solaris platform, we are releasing the library in source code form to the research community. The goal is to foster research in host-based intrusion detection by eliminating obstacles (such as engineering a prototype to read BSM data directly from the platform) for other researchers. Please contact the authors for more information on how to download the library.

6 Conclusions

Most of today's commercial intrusion detection systems are designed only to detect known attacks. Because new attacks are discovered on a weekly and sometimes daily basis, we feel it is imperative that approaches to detecting novel attacks be developed. To this end, we have developed an anomaly detection approach that learns normal program behavior.

We implemented three different machine learning algorithms for the purpose of program-based intrusion detection: Elman artificial neural networks, a string transducer, and a state tester. The results from evaluating these algorithms in the 1999 Lincoln Laboratory/DARPA Intrusion Detection evaluation are summarized here. The results demonstrate that these techniques are very good at detecting user-to-root types of attacks, and program misuse attacks in general, with low false alarm rates.

We have implemented a real-time prototype that implements the Elman network. The robust real-time prototype allows for swapping in and out different ID algorithms, including the three described in this paper. We have currently released the real-time BSM parser, BSMart, for other researchers in the community. In the future, we intend to release our robust real-time prototype as well.

Acknowledgment

This work was sponsored under the Defense Advanced Research Projects Agency (DARPA) Contract DAAH01-98-C-R145. 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. We also acknowledge Aaron Schwartzbard for his contributions to this paper.

References

1. Michle Basseville and Igor V. Nikiforov. *Detection of Abrupt Changes - Theory and Application*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

2. B. Pearlmutter C. Warrender, S. Forrest. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy*, pages 133–145, 1999.
3. B. Pearlmutter C. Warrender, S. Forrest. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy*, pages 133–145, 1999.
4. P. D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: Algorithms, analysis and implications. In *IEEE Symposium on Security and Privacy*, 1996.
5. J.L. Elman Finding structure in time. *Cognitive Science*, 14:179–211, 1990.
6. S. Forrest, S.A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.
7. S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE, May 1996.
8. Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society, IEEE Computer Society Press, May 1996.
9. Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. *Information and Computation*, 138(1):23–48, 10 October 1997.
10. A.K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*. USENIX Association, April 11-12 1999. To appear.
11. A.K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98)*, December 1998.
12. M. Kearns and L.G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 433–444, New York, NY, 1989. ACM.
13. Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):24–42, September/October 1997.
14. A.P. Kosoresow and S.A. Hofmeyr. Intrusion detection via system call traces. *Software*, 14(5):35–42, September-October 1997. IEEE Computer Society.
15. T. L. Lai. Information bounds and quick detection of parameter changes in stochastic systems. *IEEE Transactions on Information Theory*, 44(7):2917–2929, 1998.
16. W. Lee, S. Stolfo, and P.K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of AAAI97 Workshop on AI Methods in Fraud and Risk Management*, 1997.
17. L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall (Signal Processing Series), Englewood Cliffs, NJ, 1993.
18. R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proceedings of the 1998 National Information Systems Security Conference (NISSC'98)*, pages 338–347, October 1998.
19. V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.