Selection, Combination, and Evaluation of Effective Software Sensors for Detecting Abnormal Computer Usage

Jude Shavlik
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
shavlik@cs.wisc.edu

Mark Shavlik
Shavlik Technologies
2665 Long Lake Road, Suite 4000
Roseville, MN 55113
mark.shavlik@shavlik.com

ABSTRACT

We present and empirically analyze a machine-learning approach for detecting intrusions on individual computers. Our Winnow-based algorithm continually monitors user and system behavior, recording such properties as the number of bytes transferred over the last 10 seconds, the programs that currently are running, and the load on the CPU. In all, hundreds of measurements are made and analyzed each second. Using this data, our algorithm creates a model that represents each particular computer's range of normal behavior. Parameters that determine when an alarm should be raised, due to abnormal activity, are set on a per-computer basis, based on an analysis of training data. A major issue in intrusion-detection systems is the need for very low false-alarm rates. Our empirical results suggest that it is possible to obtain high intrusion-detection rates (95%) and low false-alarm rates (less than one per day per computer), without "stealing" too many CPU cycles (less than 1%). We also report which system measurements are the most valuable in terms of detecting intrusions. A surprisingly large number of different measurements prove significantly useful.

Categories and Subject Descriptors

D.4.6 **[Security and Protection]**,
I.2.6 **[Artificial Intelligence]**: Learning

General Terms

Algorithms, Experimentation, Security

Keywords

Intrusion detection, anomaly detection, machine learning, user modeling, Windows 2000, feature selection, Winnow algorithm

1. INTRODUCTION

In an increasingly computerized and networked world, it is crucial to develop defenses against malicious activity in information systems. One promising approach is to develop computer algorithms that detect when someone is inappropriately intruding on the computer of another person. However, *intrusion detection* is a difficult problem to solve [3]. System performance cannot be adversely affected, false positives must be minimized, and intrusions must be caught (i.e., false negatives must be very low). The current state of the art in intrusion-detection systems is not good; false positives are much too high and successful detection is unfortunately too rare. We report on an approach where we have made significant advances toward creating an intrusion-detection system that requires few CPU cycles (less than 1%), produces few false alarms (less than one per day), and detects most intrusions quickly (about 95% within five minutes).

Intrusion-detection systems (IDS's) can either (a) look for known attack patterns or (b) be "adaptive software" that is smart enough to monitor and learn how the system is supposed to work under normal operation versus how it works when misuse is occurring [9]. We address approach (b) in this article. Specifically, we are empirically determining which sets of fine-grained system measurements are the most effective at distinguishing usage by the assigned user of a given computer from misusage by others, who may well be "insiders" [3; 11] within an organization.

We have created a prototype anomaly-detection system that creates statistical profiles of the normal usage for a given computer running Windows 2000. Significant deviations from normal behavior indicate that an intrusion is likely occurring. For example, if the probability that a specific computer receives 10 Mbytes/sec during evenings is measured to be very low, then when our monitoring program detects such a high transfer rate during evening hours, it can suggest that an intrusion may be occurring.

The algorithm we have developed measures over two-hundred Windows 2000 properties every second, and creates about 1500 "features" out of them. During a machine-learning "training" phase, it learns how to weight these 1500 features in order to accurately characterize the particular behavior of each user – each user gets his or her own set of feature weights. Following training, every second all of the features "vote" as to whether or not it seems like an intrusion is occurring. The weighted votes "for" and "against" an intrusion are compared, and if there is enough evidence, an alarm is raised. (Section 2 presents additional details about our IDS algorithm that are being glossed over at this point.)

This ability to create statistical models of individual computer's normal usage means that each computer's unique characteristics serve a protective role. Similar to how each person's antibodies can distinguish one's own cells from invading organisms, these statistical-profile programs can, as they gather data during the normal operation of a computer, learn to distinguish "self" behavior from "foreign" behavior. For instance, some people use Notepad to view small text files, while others prefer WordPad. Should someone leave their computer unattended and someone else try to inappropriately access their files, the individual differences between people's computer usage will mean that our statistical-modeling program will quickly recognize this illegal access.

We evaluate the ability to detect computer misuse by collecting data from multiple employees of Shavlik Technologies, creating user profiles by analyzing "training" subsets of this data, and then experimentally judging the accuracy of our approach by predicting whether or not data in "testing sets" is from the normal user of a given computer or from an intruder. The key hypothesis investigated is whether or not creating statistical models of user behavior can be used to accurately detect computer misuse. We focus our algorithmic development on methods that produce very low false-alarm rates, since a major reason system administrators ignore IDS systems is that they produce too many false alarms.

Our empirical results suggest that it is possible to detect about 95% of the intrusions with less than one false alarm per (8 hr) day per user. It should be noted, though, that these results are based on our model of an "insider intruder," which assumes that when Insider *Y* uses User *X*'s computer, that *Y* is not able to alter his or her behavior to explicitly mimic *X*'s normal behavior. The training phase our approach can be computationally intensive due to some parameter tuning, but this parameter tuning could be done on a central server or during the evenings when users are not at work. The CPU load of our IDS is negligible during ordinary operation; it requires less than 1% of the CPU cycles of a standard personal computer. Our approach is also robust to the fact that users' normal behavior constantly changes over time.

Our approach can also be used to detect abnormal behavior in computers operating as specialized HTTP, FTP, or email servers. Similarly, these techniques could be used to monitor, say, the behavior of autonomous intelligent software agents in order to detect rogue agents whose behavior is not consistent with the normal range of agent behavior for a given family of tasks. However, the experiments reported herein only involve computers used by humans doing the normal everyday business tasks. While we employ the word "user" throughout this article, the reader should keep in mind that our approach applies equally well to the monitoring of servers and autonomous intelligent agents. All that would be needed to apply our approach to a different scenario would be to define a set of potentially distinctive properties to measure and to write code that measured these properties periodically.

Previous empirical studies have investigated the value of creating intrusion-detection systems by monitoring properties of computer systems, an idea that goes back at least 20 years [2]. However, prior work has focused on Unix systems, whereas over 90% of the world's computers run some variant of Microsoft Windows. In addition, prior studies have not looked at as large a collection of system measurements as we use. For example, Warrender et al. [13], Ghosh et al. [4], and Lane and Brodley [5] only look at Unix system calls, whereas Lee et al. [7] only look at audit data, mainly from the TCP program. Lazarevic et al. [6] provide a summary of some of the recent research on the application of data mining to network-based anomaly detection.

In Section 2 we describe the algorithm we developed that analyzes the Windows 2000 properties that we measure each second, creating a profile of normal usage for each user. Section 3 presents and discusses empirical studies that evaluate the strengths and weaknesses of this algorithm, stressing it along various dimensions such as the amount of data used for training. This section also lists which Windows 2000 properties end up with the highest weights in our weighted-voting scheme. Section 4 describes possible future follow-up research tasks, and Section 5 concludes this article.

2. ALGORITHM DEVELOPED

In this section we describe the algorithm that we developed. Our key finding is that a machine-learning algorithm called Winnow [8], a *weighted-majority* type of algorithm, works very well as the core component of an IDS.

This algorithm operates by taking weighted votes from a pool of individual prediction methods, continually adjusting these weights in order to improve accuracy. In our case, the individual predictors are the Windows 2000 properties that we measure, where we look at the probability of obtaining the current value and comparing it to a threshold. That is, each individual measurement suggests that an intrusion may be occurring if:

$$\text{Prob (measured property has value } v) \; < \; p \qquad \text{[Eq. 1]}$$

Each property we measure votes as to whether or not an intrusion is currently occurring. When the weighted sum of votes leads to the wrong prediction (intrusion vs. no intrusion), then the weights of all those properties that voted incorrectly are halved. Exponentially quickly, those properties that are not informative end up with very small weights. Besides leading to a highly accurate IDS, the Winnow algorithm allows us to see which Windows 2000 properties are the most useful for intrusion detection, namely those properties with the highest weights following training (as we shall see, when viewed across several users, a surprisingly high number of properties end up with high weights).

Actually, rather than using Equation 1, we found [12] it slightly better to compare probabilities relative to those in general (i.e., computed over all our experimental subjects) and use:

$$\text{Prob(value | user X)} \; / \; \text{P(value | general public)} \; < \; r \qquad \text{[Eq. 2]}$$

An alarm is sounded if this ratio is less than some constant, *r*. This way we look for relatively rare events for a specific user rather than rare events in general (it may well make sense to use *both* Equations. 1 and 2, and in the one experiment [12] where we did so - using *W* = 1200sec in Table 1's algorithm - we increased the detection rate from 94.7% to 97.3% while still meeting our target of less than one false-alarm per day).

The idea behind using the above ratio is that it focuses on feature values that are rare for this user relative to their probability of occurrence in the general population. For example, feature values that are rare for User *X* but also occur rarely across the general population may not produce low ratios, while feature values that are rare for User *X* but are not rare in general will. That is, this ratio distinguishes between "rare for User *X* and for other computer users as well" and "rare for User *X* but not rare in general."

We estimate *prob(**feature=value** for the general population)* by simply pooling all the training data from our experimental subjects, and then creating a discrete probability distribution using ten bins, using the technique explained below. Doing this in a fielded system would be reasonable, since in our IDS design one requires a pool of users for the training and tuning phases.

Because our experimental setup only involves measurements from normal computer users, the use of our ratio of probabilities makes sense in our experiments, since it defines "rare for User *X*" relative to the baseline of other computer users operating normally. However, it is likely that the behavior of intruders, even insiders working at the same facility, may be quite different from normal computer usage (unfortunately we do not yet have such data to analyze). For example, an intruder might do something that is rare in general, and hence Equation 2 above might not produce a value less than the setting for the threshold *r*.

Before presenting our algorithm that calls as a subroutine the Winnow algorithm, we discuss how we make "features" out of the over two-hundred Windows properties that we measure. Technically, it is these 1500 or so features that do the weighted voting.

2.1 Features Used

Space limitations preclude describing here all of the 200+ properties measured. Appendix A of our full project report [12] lists and briefly describes all of the Windows 2000 properties that we measure; some relate to network activity, some to file accesses, and others to the CPU load of the programs currently running. Most come from Windows' *perfmon* ("performance monitor") program. Several features we measure appear in Tables 3 and 4 of this article. For each of these measurements, we also derive additional measurements:

Actual Value Measured
Average of the Previous 10 Values
Average of the Previous 100 Values
Difference between Current Value and Previous Value
Difference between Current Value and Average of Last 10
Difference between Current Value and Ave of Last 100
Difference between Averages of Previous 10 and Previous 100

As we discovered in our experiments, these additional "derived" features play an important role; without them intrusion-detection rates are significantly lower. For the remainder of this article, we will use the term "feature" to refer the combination of a measured Windows 2000 property and one of the seven above transformations. In other words, each Windows 2000 property that we measure produces seven features. (The first item in the above list is not actually a *derived* feature; it is the "raw" measurement, but we include it in the above list for completeness.)

2.2 Our IDS Algorithm

Table 1 contains our main algorithm. We take a machine-learning [10] approach to creating an IDS, and as is typical we divide the learning process into three phases. First, we use some *training data* to create a model; here is where we make use of the Winnow algorithm (see Table 2), which we further explain below. Next, we use some more data, the *tuning set*, to "tune" some additional parameters in our IDS. Finally, we evaluate how well our learned IDS works be measuring its performance on some *testing data.* We repeat this process for multiple users and report the average *test-set* performance in our experiments.

The Windows 2000 properties that we measure are continuous-valued, and in Step 1b of Table 1 we first decide how to discretize each measurement into 10 bins; we then use these bins to create a discrete probability distribution for the values for this feature. Importantly, we do this discretization separately for each user, since this way we can accurately approximate each user's probability distribution with our 10 bins. (We did not experiment with values other than 10 for the number of bins. We chose 10 arbitrarily, though it does make sense that this number be small to reduce storage demands and to "smooth" our measurements.)

We always place the value 0.0 in its own bin, since it occurs so frequently. We then choose the "cut points" that define the remaining bins by fitting a sample of the measurements produced by each user to each of several standard probability distributions: uniform, Gaussian, and Erlang (for *k* ranging from 1 to 100). When *k* = 1 the Erlang is equivalent to the better known ("decaying") Exponential distribution, and as *k* increases the distribution looks more and more like a Gaussian. We then select the probability distribution that best fits the sample data (i.e., has the lowest root-mean-squared error), and create our 10 bins as follows:

- For the *uniform* probability distribution, we uniformly divide the interval *[minimum value, maximum value]* into seven bins, and use the two remaining bins for values less than the minimum and greater than the maximum (since values encountered in the future might exceed those we have seen so far).
- For the *Gaussian* probability distribution, we place the lowest 0.005% of the probability mass in the first bin, the next 0.1% in the second bin, 5% in the next bin, and 15% in the bin after that. We do the same working down from the highest value, which leaves about 60% for the middle bin (60% is roughly one standard deviation around the mean of a Gaussian).
- For the *Exponential* probability distribution, we put half the probability mass in the first bin, and then half of the remaining probability mass in each successive bin (except for the last bin).

- For the *Erlang* probability distribution, we execute a combination of what we do for the Gaussian and the Exponential, depending on the value of *k*.

We did not investigate alternate design choices in our discretization process; we developed the above approach and then used it unchanged during our subsequent learning-algorithm development and evaluation.

Table 1. Creating and Maintaining an IDS for User *X*

Step 1: Initial Training

Step 1a: Collect measurements from User *X* and place in TRAINSET.

Step 1b: Using TRAINSET, choose good "cut points" (for User X) to discretize continuous values. See text.

Step 1c: Select weights for User *X*'s measured features by applying the Winnow algorithm (see Table 2 and accompanying text) using TRAINSET and an equal number of "archived" sample measurements from other users. However, be sure to discretize the measurements from the other users by applying User *X*'s cut points, since we will be pretending that the other users are inappropriately using *X*'s computer.

Step 2: Parameter Tuning

Step 2a: Using new measurements collected from User *X* and other users (the TUNESET), perform Steps 2b and 2c, calculating *false-alarm* and *intrusion-detection* rates in conceptually independent runs for as many as possible combinations of the parameters being tuned: W, $thresh_{mini}$ and $thresh_{full}$.

Step 2b: Use the weighted features to "vote" on "mini-alarms" each second;
if $(wgtedVotes_{FOR} / wgtedVotes_{AGAINST}) > thresh_{mini}$ then raise a *mini-alarm*. See Steps 2a and 2b of Table 2.

Step 2c: If the fraction of *mini-alarms* in the last W seconds $\geq thresh_{full}$ then raise an alarm signaling that an intrusion might be occurring. After each "call," wait another W seconds (i.e, the windows do no overlap).

Step 2d: Given the specified maximum false-alarm rate per (8-hour) day, choose the parameter settings that produce the highest intrusion-detection rate on the set of sample "other" users, while not producing more than the desired number of false alarms for User *X*.

Step 3: Continual Operation

Using Step 2d's chosen settings for W, $thresh_{mini}$ and $thresh_{full}$, repeat Steps 2b and 2c on the TESTSET. (It might make sense to periodically retrain and retune in order to adjust to changing user behavior – see text.)

Most of our features turned out to be best modeled by Gaussians, with the Exponential distribution being the second most common selection. One final point about converting to a discrete probability distribution needs to be mentioned: for those Windows 2000 measurements that vary over orders of magnitude (e. g., bytes sent per second); we use the *log* of their values.

After we have discretized our features, we simply count how often in the training data did a feature value fall into a given bin, thereby producing a probability distribution (after normalizing by the total number of counts). Following standard practice, we initialize all bins with a count of 1; this ensures that we will never estimate from our finite samples a probability of zero for any bin. We are now able to estimate the *Prob(feature = measured value)* that was mentioned earlier in Equations 1 and 2.

Table 2. Variant of Winnow that is Used

Step 1: Initialize User *X*'s weights on each measured feature (wgt_f) to 1.

Step 2: For each training example do:

Step 2a: Zero $wgtedVotes_{FOR}$ and $wgtedVotes_{AGAINST}$.

Step 2b: If then relative probability (Eq. 2) of the current measured value for feature $f < r$, then add wgt_f to $wgtedVotes_{FOR}$ otherwise add wgt_f to $wgtedVotes_{AGAINST}$.

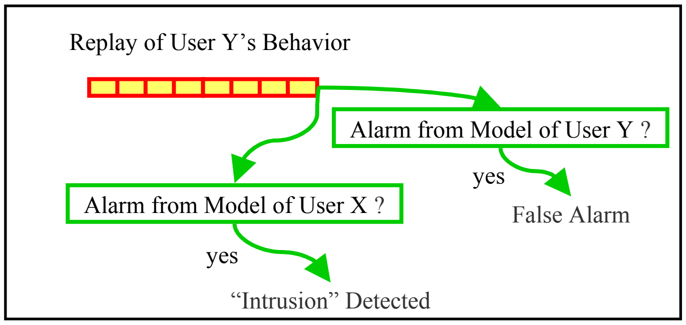
I.e., if the relative probability of the current value of feature *f* is "low," then this is evidence that something anomalous is occurring. In our experiments, we found that $r = 0.33$ worked well; however, overall performance was robust in regards to the value of *r* (and Eq 1's *p*). Various values for *r* that we tried in the range [0.25, 0.75] all worked well.

Step 2c: If $wgtedVotes_{FOR} > wgtedVotes_{AGAINST}$ then call the current measurements *anomalous.*

Step 2d: If User *X* produced the current measurements and they are considered anomalous, then a *false-alarm error* has been made. Multiply by ½ all those features that incorrectly voted *for* raising an alarm.

Otherwise if some other user produced the current measurements and they were *not* considered anomalous, then an *intrusion* has been missed. Multiply by ½ all those features that incorrectly voted *against* raising an alarm. When neither a *false-alarm* nor a *missed-intrusion* error occurred, leave the current weights unchanged.

We next turn to discuss using these probabilities to learn models for distinguishing the normal user of a given computer from an intruder. Ideally we would use training data where some User *X* provided the examples of normal (i. e., non-intrusion) data and we had another sample of data measured during a wide range of intrusions on this user's computer. However, we do not have such data (this is a problem that plagues IDS research in general), and so we use what is a standard approach, namely we collect data from several users (in our case, 10), and we then simulate intrusions by replaying User *Y's* measurements on User *X*'s computer. We say that a *false alarm* occurs when User *Y*'s recent measurements are viewed as anomalous - that is, suggestive of an intrusion - when replayed on his or her own computer. A *detected intrusion* occurs when we view User *Y*'s measurements as being anomalous when evaluated using *X*'s feature discretization and feature weighting. (Notice that we need to use *X*'s discretization, rather than *Y*'s, since we are assuming that *Y* is operating on *X*'s computer.) Figure 1 abstractly illustrates how we define false alarms and detected intrusions in our experimental setting.



Replay of User Y's Behavior
Alarm from Model of User Y ?
yes
False Alarm
Alarm from Model of User X ?
yes
"Intrusion" Detected

Figure 1. False Alarms and Detected Intrusions

As mentioned, we use Table 2's version of Littlestone's Winnow algorithm [8] to choose weights on the features we measure. This algorithm is quite simple, yet has impressive theoretical properties and practical success on real-world tasks, especially those that have a very large number of features, which is the case for our task. As already discussed, this algorithm sums weighted votes "for" and "against" the possibility that an intrusion is currently occurring. When the winning choice (i. e., "for" or "against") is wrong, then all those features that voted for the wrong choice have their weights halved. We perform the Winnow algorithm for each user, in each case using a 50-50 mixture of examples, with half drawn from this user's measured behavior (the "*against* an intrusion" examples) and half randomly drawn from some other user in the experiment (the "*for* an intrusion" examples).

In order to raise an alarm after the training phase (Step 1 in Table 1) has set the feature weights, our algorithm does not simply use the current weighted vote. Instead, the current weighted vote can raise what we call a *mini alarm*, and we require that there be at least *N* mini alarms in the last *W* seconds in order to raise an actual alarm. In other words, our intrusion detector works as follows (Steps 2b and 2c in Table 1):

If *weighted_vote(current measurements)* $>$ $thresh_{mini}$
then raise "mini" alarm

If fraction of "mini" alarms in last *W* sec $\geq$ $thresh_{full}$
then predict intrusion

As will be seen in Section 3, *W* needs to be on the order of 100 to get good detection rates with few false alarms.

We choose the settings for our parameters *on a per-user basis* by evaluating performance on a set of *tuning* data – see Step 2 of Table 1. One significant advantage of a data-driven approach like ours is that we do not have to pre-select parameter values. Instead, the learning algorithm selects for each user his or her personal set of parameter values, based on the performance of these parameters on a substantial sample of "tuning set" data.

The only computationally demanding portion of our algorithm is the parameter-tuning phase, which depends on how many parameter combinations are considered and on how much tuning data each combination is evaluated. In a fielded system, it might make sense to do this step on a central server or during the evenings. The other tasks of measuring features, computing weighted sums, and using Winnow to adjust weights can all be done very rapidly. Outside of the parameter tuning, Table 1's algorithm requires less than 1% of a desktop computer's CPU cycles.

Notice that even during the testing phase (e. g., Step 3 in Table 1), we find it necessary to still execute the Winnow algorithm, to adjust the weights on the features after our algorithm decides whether or not an intrusion occurred. If we do not do this, we get too many false alarms when the user's behavior switches, and the intrusion-detection rates drastically drops to 20% from about 95%. On the other hand continually adjusting weights means that if we *miss* an intrusion we will start learning the behavior of the intruder, which is a weakness of our approach (and a weakness of statistics-based approaches for intrusion detection in general). *This also means that the empirical results reported in the next section should properly be interpreted as estimating the probability that we will detect an intruder after his or her* <u>*first*</u> *W seconds of activity.* A subject for future work is to empirically evaluate how likely our approach will detect an intruder in the *second* (and successive) W seconds of activity, given we did not detect the intruder in the first W seconds. On the other hand, the fact that we continually are adjusting the weights means that after the legitimate user reauthenticates himself or herself after a false alarm, our algorithm will adapt to the change in the user's behavior.

Obviously there is a delicate balance between adapting quickly to changes in the legitimate user's behavior, and thus reducing false alarms, and adapting too quickly to the activity of an intruder and thus thinking the intruder's behavior is simply a change in the behavior of the normal user of the given computer and thereby missing actual intrusions. It is a simple fact of life that most users' behavior is wide ranging and changing over time. The more consistent a user's behavior is, and the more accurately we can capture his or her idiosyncrasies, the better our approach will work.

3. EXPERIMENTAL EVALUATION

This section reports some experimental evaluation of our IDS algorithm. Additional experiments are reported in detail in Shavlik and Shavlik [12], with some of their results mentioned in this article.

3.1 Methodology

We collected about 8 GB of data from 16 employees of Shavlik Technologies who volunteered to be experimental subjects. We only collected data between 9am and 5pm on weekdays.

Of these 16 experimental subjects, we use 10 during training (Steps 1 and 2 of Table 1); for each one, we train our IDS to recognize the differences in behavior of that user from the other 9 users. We call these 10 users "insiders" and view them as members of a small group of co-workers. The remaining 6 subjects, for whom we have a total of about 50 work days of measurements, serve as simulated "external" intruders, i.e., users whose computer-usage behavior is not seen during training (including computing the denominator in Eq. 2) – these 6 experimental subjects are only used during the *testing* phase (Step 3 of Table 1) and are never used during the training and tuning phases. Hence, one expects that these 6 "outsiders" would be harder to recognize as intruders on User *X*'s computer since their behavior is not observed while the IDS's are still learning.

3.2 Primary Results and Discussion

Figure 2 shows, as a function of *W* (see Table 1) the detection and false-alarm rates for the scenario where the training lasts 15 work days (432,000 seconds), and the tuning, and testing periods each last 10 work days (288,000 seconds). The train, tune, and test sets are temporally disjoint from one another. This scenario involves a five-week-long training process, but as presented in Shavlik and Shavlik [12] shorter training periods produce results nearly as good.

The results are averages over the 10 "insiders;" that is, each of these 10 experimental subjects is evaluated using the other 9 subjects as "insider intruders" and the above-described 6 "outsider intruders," and the 10 resulting sets of false-alarm and detection rates are averaged to produce Figure 2. During the tuning phase of Table 2, the specified false-alarm rate of Step 2e was set to 0; such a extreme false-alarm rate could always be produced on the tuning set, though due to the fact we are able to explicitly fit our parameters only to the tuning data, a false-alarm rate of zero did not result during the testing rate (as one expects). *Over fitting* (getting much higher accuracies on the tuning data than on the testing data due to having too many "degrees of freedom" during the tuning phase) is arguably the key issue in machine learning and is central to adaptive IDS's.

As can be seen in Figure 2, for a wide range of window widths (from 1 to 20 minutes), the false-alarm rates are very low – always less than one per eight-hour work day per user - and the intrusion-detection rates are impressively high, nearly 95%. Interestingly, the detection rate for "outsiders," whose behavior is never seen during training, is approximately the same as for "insiders." This suggests that our learning algorithm is doing a good job of learning what is characteristic about User *X*, rather than just exploiting idiosyncratic differences between User *X* and the other nine "insiders."

Based on Figure 2, 300 seconds is a reasonable setting for W in a fielded system, and in most of the subsequent experiments in this section use that value.

(It should be noted that going down to W = 60 sec in Figure 2 is not completely appropriate. Some of the features we use are averages of a given measurement over the last 100 seconds, as explained earlier in this article. In all of our experiments, we do not use any examples where the user's computer has not been turned on for at least 100 seconds. Hence, when we replay a 60-second window of activity from User *Y* on User *X*'s computer, there is some "leakage" of User *Y*'s data going back 100 seconds. In a fielded system, 40 seconds worth of the data would actually be from User *X* and 60 seconds from User *Y*. However, our experimental setup does not currently support such "mixing" of user behavior. Should a fielded system wish to use W=60 sec, a simple solution would be to average over the last 60 seconds, rather than the last 100 seconds as done in our experiments. We do not expect the impact of such a change to be significant. The data point for W = 10 sec in Figure 2 only uses features that involve no more than the last 10 seconds of measurements, as a reference point – the issue of using less or more than the last 100 seconds of measurements is visited in more depth in the next section.)

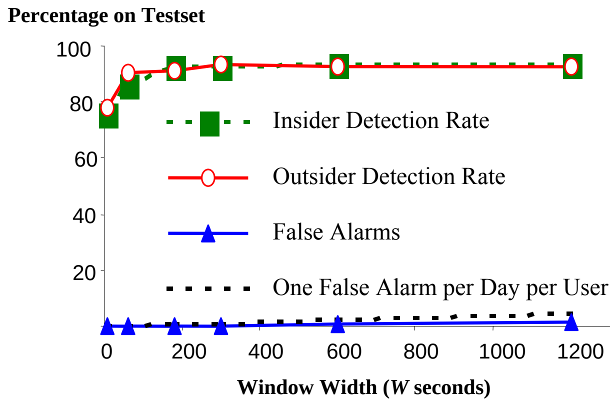


Figure 2. False Alarm and Detection Rates

One potentially confusing technical point needs to be clarified at this point. In an eight-hour work day, there are 480 sixty-second-wide, non-overlapping windows (i. e., *W* = 60) but only 48 six-hundred-second-wide (*W* = 600) ones. So one false alarm per day for *W* = 60 sec corresponds to a false-alarm rate of 0.2%, whereas for *W* = 600 sec a false-alarm rate of 2.1% produces one false-alarm per day on average. The (lower) dotted line in Figure 2 shows the false-alarm rate that produces one false alarm per day per user. Although it cannot be seen in Figure 2, as *W* increases the actual number of false alarms per day decreases. Making a call every second leads to too many false alarms [12], so we use non-overlapping windows. Conversely, as *W* increases an intruder is able to use someone else's computer longer before being detected.

To produce Figure 2's results, Table 2's tuning step considered 11 possible settings for $\text{threshold}_{\text{mini}}$ (0.8, 0.85, 0.90, 0.95, 0.97, 1.0, 1.03, 1.05, 1.1, 1.15, and 1.2) and 26 for $\text{threshold}_{\text{full}}$ (0.01, 0.25, 0.5, 0.75, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.975, 0.99, and 1.0), that is 11x26=286 different combinations of these two parameters. We did not experiment with different choices for the particular values and number of the candidate parameter settings, except we found it necessary to restrict $\text{threshold}_{\text{mini}}$ = 1.0 in the cases in Figure 2 where W = 10 sec and W = 60 sec.

Table 3 shows the highest-weighted features at the end of Figure 2's experiment, where the weights are averaged over all ten of our experimental subjects and over those values for W > 10 used to create Figure 2; for each experimental subject and setting for W, we normalize the weights so that they sum to 1, thus insuring that each configuration contributes equally. Remember that the weights are always changing, so this table should be viewed as a representation "snapshot." (Appendix A of Shavlik and Shavlik [12] contains additional explanations of several of these features).

Table 3. Features with the 25 Highest Weights Averaged Across the Experiments that Produced Figure 2.

Print Jobs, *Average of Previous 100 Values (ranked #1)*

Print Jobs, *Average of Previous 10 Values*

System Driver Total Bytes, *Actual Value Measured*

Logon Total, *Actual Value Measured*

Print Jobs, *Actual Value Measured*

LSASS: Working Set, *Average of Previous 100 Values*

Number of Semaphores, *Average of Previous 100 Values*

Calc: Elapsed Time,
Difference between Averages of Prev 10 and Prev 100

Number of Semaphores, *Actual Value Measured*

LSASS: Working Set, *Average of Previous 10 Values*

CMD: Handle Count,
Difference between Current and Average of Last 10

CMD: Handle Count, *Average of Previous 10 Values*

Write Bytes Cache/sec,
Difference between Current and Average of Last 10

Excel: Working Set,
Difference between Current and Average of Last 10

Number of Semaphores, *Average of Previous 10 Values*

CMD: % Processor Time,
Difference between Averages of Prev 10 and Prev 100

LSASS: Working Set, *Actual Value Measured*

System Driver Total Bytes, *Average of Previous 100 Values*

CMD: % Processor Time,
Difference between Current and Average of Last 100

CMD: % Processor Time,
Difference between Current and Average of Last 10

System Driver Resident Bytes, *Actual Value Measured*

Excel: Handle Count, *Average of Previous 10 Values*

Errors Access Permissions,
Difference between Current and Average of Last 10

File Write Operations/sec, *Average of Previous 100 Values*

System Driver Resident Bytes, *Average of Previous 10 Values*

Table 4. The 25 Measurements with the Highest Number of Occurrences in the Top 10 Weights, Including Ties, in the Experiments that Produced Figure 2 (the numbers in parentheses are the percentages of Top 10 appearances)

Number of Semaphores (43%)

Logon Total (43%)

Print Jobs (41%)

System Driver Total Bytes (39%)

CMD: Handle Count (35%)

System Driver Resident Bytes (34%)

Excel: Handle Count (26%)

Number of Mutexes (25%)

Errors Access Permissions (24%)

Files Opened Total (23%)

TCP Connections Passive (23%)

LSASS: Working Set (22%)

LSASS: % Processor Time (22%)

SYSTEM: Working Set (22%)

Notepad: % Processor Time (21%)

CMD: Working Set (22%)

Packets/sec (21%)

Datagrams Received Address Errors (21%)

Excel: Working Set (21%)

MSdev: Working Set (21%)

UDP Datagrams no port / sec (17%)

WinWord: Working Set (17%)

File Write Operations / sec (16%)

Bytes Received / sec (16%)

Bytes Transmitted / sec (16%)

Observe that a wide range of features appear in Table 3: some relate to network traffic, some measure file accesses, others refer to which programs are being used, while others relate to the overall load on the computer. It is also interesting to notice that for some features their average values over 100 seconds are important, whereas for others their instantaneous values matter, and for still others what is important is the *change* in the feature's value.

A weakness of Table 3 is that a measured Windows 2000 property that is important for only one or two subjects might not have a very high average weight. Table 4 provides a different way to see which features play important roles. To produce this table we count how often each measured property appears in the Top 10 weights (including ties, which are common) following training. Surprisingly, over half of the Windows 2000 properties we measure appear at least once in some Top 10 list! This supports our thesis that one should monitor a large number of system properties in order to best create a behavioral model that is well tailored to each individual computer user. Our project's final report [12] displays longer and additional lists of the highest-weighted features, including those for one specific user.

Most of the "derived" calculations (see Section 2.1) are used regularly in the highly weighted features, with the exception of "*Difference from Previous Value*," which appears in the Top 50 weighted features only about 1/20th as often as the others., presumably because it is too noisy of an estimate and needs to be smoothed. "*Difference between Current and Average of Last 10*" is the most used, but the difference between the most used and the 6th-most used is only a factor of two.

3.3 Additional Results

Tables 3 and 4 show that the features that use the last *N* measurements of a Windows 2000 property play an important role. Figure 3 illustrates the performance of Table 1's algorithm when we restrict features to use at most the last 1, 10, 100, or 1000 measurements, respectively, of the Window 2000 properties that we monitor. The *Y*-axis is the test-set detection rate and in all cases the false-alarm rate meets our goal of no more than one per user per workday. Figure 3's data is from the case where W = 300 seconds; 15 days of training data, 3 of tuning, and 3 of testing are used for each experimental subject.

Figure 3 shows that there is an advantage in considering features that have longer "histories." However, the cost of a longer history is that more data needs to be collected to define a feature value. That is, if histories can go back as far as 1000 seconds (a little over 15 minutes), then it will take 1000 seconds after an intrusion until all of the feature values are due solely to the intruder's behavior. It appears that limiting features to at most the last 100 seconds of measurements is a good choice.

So far we have reported results average over our pool of 10 insiders and 6 outsiders. It is interesting to look at results from individual experimental subjects. Table 5 reports how often User *Y* was *not* detected when "intruding" on User *X*'s computer. For example, the cell *<row=User4, column=User1>* says that the probability of detection is 0.86 when User 1 operates on User 4's computer for 1200 seconds. (The rightmost column is the detection rate when outsiders operate on each insider's computer.)

Given that the overall detection rate is about 95% (i.e., only 5% of 1200-sec intrusions do not sound alarms), one might expect that most of the individual penetration rates would range from, say, 2% to 10%. However, the results are much more skewed. In most cases, *all* (or nearly all) the attempted intrusions are detected – the majority of cells in Table 5 contain 0's (in fact we report "penetration" rates rather than detection rates in this table because otherwise all of the 100%'s would be visually overwhelming). But in several cases a user is frequently not detected when operating on another user's computer.

One implication of the results in Table 5 is that for a fielded system one could run experiments like these on some group of users, and then identify for which ones their computer behavior is sufficiently distinctive that Table 1's algorithm provides them effective protection.

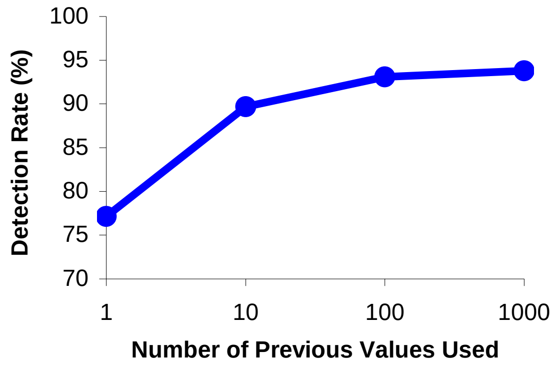


Figure 3. Detection Rate as Function of Number of Previous Values Used (*W* = 300 sec)

Table 5. Percentage (%) of Times that User *Y* Successfully Intruded on User *X*'s Machine (using *W* = 1200 sec). The columns range over *Y* and the rows over *X*. The rightmost column (O) reports the rate of successful intrusions by the set of six outsiders. Cells with values less than 0.5% are left blank.

Y / X	1	2	3	4	5	6	7	8	9	10	O
1			7		23				41		15
2	5		1		35					2	
3					26					5	
4	14										
5	2		4							4	
6									1		20
7									45	75	2
8											4
9								3			2
10	9	60	16	1		43	6		25		3

3.4 Comparison to Naïve Bayes

A successful algorithm on many tasks is the Naïve Bayes algorithm [10], in which one assumes all the features (i.e., measurements in our case) are conditionally independent of each other given the category, and estimates the probability of obtaining the current set of measurements given each of the possible categories (intrusion versus normal behavior in our case).

We applied the Naïve Bayes algorithm in the same experimental setup as used to evaluate Table 1's algorithm. However, the best results we have been able to obtain (for *W* = 1200 seconds) are a 59.2% detection rate with an average of 2.0 false alarms per day per user, which compares poorly to Table 1's algorithm's results, in the identical scenario, of a 93.6% detection with an average of

0.3 false alarms per day per user. (In fact, we started out this project using the Naïve Bayes algorithm, and then switched to our Winnow-based approach when we realized that Naïve Bayes' independence assumption was too severely violated for us to create an effective anomaly detector.)

4. FUTURE WORK

We discuss a few possible extensions to the work reported above that have not yet been fully discussed. An obvious extension is to obtain and analyze data from a larger number of users, as well as data from a collection of server machines. And of course it would be greatly beneficial to have data gathered during actual intrusions, rather than simulating them by replaying one user's measurements on another user's computer. Among other advantages, having data from a larger pool of experimental subjects would allow "scaling up" issues to be addressed, statistically justified confidence intervals on results to be produced, and parameters to be better tuned (including many for which we have "hard-wired in" values in our current experiments).

When we apply the Winnow algorithm during the training phase (Step 1 in Table 1), we get remarkable accuracies. For example, out of 3,000,000 seconds of examples (half that should be called an intrusion and half that should not), we consistently obtain numbers on the order of only 150 missed intrusions and 25 false alarms, and that is from starting with all features weighted equally. Clearly the Winnow algorithm can quickly pick out what is characteristic about each user and can quickly adjust to changes in the user's behavior. In fact, this rapid adaptation is also somewhat of a curse (as previously discussed in Section 2), since an intruder who is not immediately detected may soon be seen as the normal user of a given computer. This is why we look for *N* mini-alarms in the last *W* seconds before either sounding an alarm or calling the recent measurements normal and then applying Winnow to these measurements; our assumption is that when the normal user changes behavior, only a few mini-alarms will occur, whereas for intruders the number of mini-alarms produced will exceed *N*. Nevertheless, we still feel that we are not close to fully exploiting the power of the Winnow algorithm on the intrusion-detection task. With more tinkering and algorithmic variations, it seems possible to get closer to 99% detection rates with very few false alarms.

In Section 2's Winnow-based algorithm we estimate the probability of the current value for a feature and then make a simple "yes-no" call (see Eq. 1 and 2), regardless of how close the estimated probability is to the threshold. However, it seems that an extremely low probability should have more impact than a value just below the threshold. In the often-successful Naïve Bayes algorithm, for example, actual probabilities appear in the calculations, and it seems worthwhile to consider ways of somehow combining the weights of Winnow and the actual (rather than thresholded) probabilities.

In our main algorithm (Table 1) we did not "condition" the probabilities of any of the features we measured. Doing so might lead to more informative probabilities and, hence, better performance. For example, instead of simply considering *Prob(File Write Operations/sec)*, it might be more valuable to use *Prob(File Write Operations/sec | MS Word is using most of the recent cycles)*, where '|' is read "given." Similarly, one could use the Winnow algorithm to select good *pairs* of features. However these alternatives might be too computationally expensive unless domain expertise was somehow used to choose only a small subset of all the possible combinations.

In none of the experiments of this article did we *mix* the behavior of the normal user of a computer and an intruder, though that is likely to be the case in practice. It is not trivial to combine two sets of Windows 2000 measurements in a semantically meaningful way (e. g., one cannot simply add the two values for each feature or, for example, CPU utilizations of 150% might result). However, with some thought it seems possible to devise a plausible way to mix normal and intruder behavior. An alternate approach would be to run our data-gathering software while someone is trying to intrude on a computer that is simultaneously being used by another person.

In the results reported in Section 3, we tune parameters to get zero false alarms on the tuning data, and we found that on the testing data we were able to meet our goal of less than one false alarm per user per day (often we obtained test-set results more like one per week). If one wanted to obtain even fewer false alarms, then some new techniques would be needed, since our approach already is getting no false alarms on the tuning set. One solution we have explored is to tune the parameters to zero false alarms, and then to increase the stringency of our parameters - e. g., require 120% of the number of mini-alarms as needed to get zero tuning-set false alarms. More evaluation of this and similar approaches is needed.

We have also collected Windows 2000 event-log data from our set of 16 Shavlik Technologies employees. However we decided not to use that data in our experiments since it seems one would need to be using data from people actually trying to intrude on someone else's computer for interesting event-log data to be generated. Our approach for simulating "intruders" does not result in then generation of meaningful event-log entries like failed logins.

Another type of measurement that seems promising to monitor are the *specific* IP addresses involved in traffic to and from a given computer. Possibly interesting variables to compute include the number of different IP addresses visited in the last *N* seconds, the number of "first time visited" IP addresses in the last *N* seconds, and differences between incoming and outgoing IP addresses.

A final possible future research topic is to extend the approaches in this article to local *networks* of computers, where the statistics of behavior across the set of computers is monitored. Some intrusion attempts that might not seem anomalous on any one computer may appear highly anomalous when looking at the behavior of a set of machines.

5. CONCLUSION

Our approach to creating an effective intrusion-detection system (IDS) is to continually gather and analyze hundreds of fine-grained measurements about Windows 2000. The hypothesis that we successfully tested is that a properly (and automatically) chosen set of measurements can provide a "fingerprint" that is unique to each user, serving to accurately recognize abnormal usage of a given computer. We also provide some insights into which system measurements play the most valuable roles in creating statistical profiles of users (Tables 3 and 4). Our experiments indicate that we may get high intrusion-detection rates and low false-alarm rates, without "stealing" too many CPU cycles. We believe it is of particular importance to have very low false-alarm rates; otherwise the warnings from IDS will soon be disregarded.

Specific key lessons learned are that it is valuable to:

- consider a large number of different properties to measure, since many different features play an important role in capturing the idiosyncratic behavior of at least some user (see Table 3 and 4)
- continually reweight the importance of each feature measured (since users' behavior changes), which can be efficiently accomplished by the Winnow algorithm [8]
- look at features that involve more than just the instantaneous measurements (e. g., difference between the current measurement and the average over the last 10 seconds)
- tune parameters on a per-user basis (e. g., the number of "mini alarms" in the last *N* seconds that are needed to trigger an actual alarm)
- tune parameters on "tuning" datasets and then estimate "future" performance by measuring detection and false-alarm rates on a separate "testing" set (if one only looks at performance on the data used to train and tune the learner, one will get unrealistically high estimates of future performance; for example, we are always able to tune to *zero* false alarms)
- look at the variance in the detection rates across users; for some, there are no or very few missed intrusions, while for others many more intrusions are missed – this suggests that for at least some users (or servers) our approach can be particularly highly effective

An anomaly-based IDS, such as the one we present, should not be expected to play the sole intrusion-detection role, but such systems nicely complement IDS that look for known patterns of abuse. New misuse strategies will always be arising, and anomaly-based approaches provide an excellent opportunity to detect them even before the internal details of the latest intrusion strategy are fully understood.

6. ACKNOWLEDGMENTS

We wish to thank the employees of Shavlik Technologies who volunteered to have data gathered on their personal computers. We also wish to thank Michael Skroch for encouraging us to undertake this project and Michael Fahland for programming support for the data-collection process. Finally we also wish to thank the anonymous reviewers for their insightful comments. This research was supported by DARPA's *Insider Threat Active Profiling* (ITAP) program within the ATIAS program.

7. REFERENCES

[1] R. Agarwal & M. Joshi, PNrule: A New Framework for Learning Classifier Models in Data Mining (A Case-Study in Network Intrusion Detection*) Proc. First SIAM Intl. Conf. on Data Mining,* 2001**.**

[2] J. Anderson, *Computer Security Threat Monitoring and Surveillance*, J. P. Anderson Company Technical Report, Fort Washington, PA, 1980.

[3] DARPA, *Research and Development Initiatives Focused on Preventing, Detecting, and Responding to Insider Misuse of Critical Defense Information Systems*, DARPA Workshop Report, 1999.

[4] A. Ghosh, A. Schwartzbard, & M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, *USENIX Workshop on Intrusion Detection & Network Monitoring*, April 1999.

[5] T. Lane & C. Brodley, Approaches to Online Learning and Concept Drift for User Identification in Computer Security, *Proc. KDD,* pp 259-263, 1998.

[6] A. Lazarevic, L. Ertoz, A. Ozgur, J. Srivastava & V. Kumar, A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection, *Proc. SIAM Conf. Data Mining*, 2003.

[7] W. Lee, S.J. Stolfo, and K. Mok, A Data Mining Framework for Building Intrusion Detection Models, *Proc. IEEE Symp. on Security and Privacy,* 1999*.*

[8] N. Littlestone, Learning Quickly When Irrelevant Attributes Abound. *Machine Learning* 2, pp. 285—318.

[9] T. Lunt, A Survey of Intrusion Detection Techniques, *Computers and Security* 12:4, pp. 405-418, 1993.

[10] T. Mitchell, *Machine Learning*, McGraw-Hill.

[11] P. Neumann, *The Challenges of Insider Misuse*, SRI Computer Science Lab Technical Report, 1999

[12] J. Shavlik & M. Shavlik, Final Project Report for DARPA's *Insider Threat Active Profiling* (ITAP) program, April 2002.

[13] C. Warrender, S. Forrest, & B. Pearlmutter. Detecting Intrusions using System Calls. *IEEE Symposium on Security and Privacy*, pp. 133-145, 1999.