Anomalous System Call Detection

Darren Mutz[†], Fredrik Valeur[†], Christopher Kruegel[‡], and Giovanni Vigna[†]
[†]Reliable Software Group, University of California, Santa Barbara
[‡]Secure Systems Lab, Technical University of Vienna

Intrusion detection systems (IDSs) are used to detect traces of malicious activities targeted against the network and its resources. Anomaly-based IDSs build models of the expected behavior of applications by analyzing events that are generated during the applications' normal operation. Once these models have been established, subsequent events are analyzed to identify deviations, in the assumption that anomalies represent evidence of an attack. Host-based anomaly detection systems often rely on system call sequences to characterize the normal behavior of applications. Recently, it has been shown how these systems can be evaded by launching attacks that execute legitimate system call sequences. The evasion is possible because existing techniques do not take into account all available features of system calls. In particular, system call arguments are not considered.

We propose two primary improvements upon existing host-based anomaly detectors. First, we apply multiple detection models to system call arguments. Multiple models allow the arguments of each system call invocation to be evaluated from several different perspectives. Second, we introduce a sophisticated method of combining the anomaly scores from each model into an overall aggregate score. The combined anomaly score determines whether an event is part of an attack.

Individual anomaly scores are often contradicting, and therefore, a simple weighted sum cannot deliver reliable results. To address this problem, we propose a technique that uses Bayesian networks to perform system call classification. We show that the analysis of system call arguments and the use of Bayesian classification improves detection accuracy and resilience against evasion attempts. In addition, the paper describes a tool based on our approach and provides a quantitative evaluation of its performance in terms of both detection effectiveness and overhead. A comparison with four related approaches is also presented.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Intrusion Detection, Anomaly Detection, Bayesian Network, Computer Security

1. INTRODUCTION

Intrusion detection techniques have traditionally been classified as either *misuse-based* or *anomaly-based*. Systems that use misuse-based techniques [Paxson 1998; Lindqvist and Porras 1999; Vigna et al. 2003] contain a number of attack descriptions, or signatures, that are matched against a stream of audit data looking for evidence that the modeled attacks are occurring. These systems are usually efficient and generate few erroneous detections, called false positives. The main disadvantage of misuse-based techniques is the fact that they can only detect those attacks that have been modeled. That is, they cannot detect intrusions for which they do not have a signature.

Anomaly-based techniques [Denning 1987; Ko et al. 1997; Ghosh et al. 1998] follow an approach that is complementary to misuse detection. The detection is based on models of normal user or application behavior, called profiles. Any deviation from an established profile is interpreted as being associated with an attack. The main advantage of anomaly-based techniques is the ability to identify previously unknown attacks. By defining an expected, normal state, any abnormal behavior can be detected, whether it is part of the threat model or not. The advantage of being able to detect previously unknown attacks is, however, usually paid for with a large number of false positives.

Anomaly-based systems create models of normal behavior by analyzing different input events from either network-based or host-based auditing facilities. Network-based systems [Porras and Neumann 1997; Neumann and Porras 1999; Staniford et al. 2000; Bykova et al. 2001] monitor network traffic and analyze packet headers and payloads. Host-based systems, on the other hand, focus their analysis on user activity or program behavior, as observed at the operating system or application level.

In [Denning 1987], a host-based approach is described that builds profiles based on user login times and resources accessed by users (e.g. files, programs). Simple statistical methods are then used to determine whether the observed behavior conforms to the stored profile. Unfortunately, user behavior often changes suddenly and is not easy to characterize. As a consequence, the general focus of anomaly detection research shifted from user to program behavior.

A possible approach to create models of program behavior are system call sequences [Forrest 1996]. The key observation is the fact that a program has to interact with the underlying operating system through system calls to cause permanent damage to the system. When an observed system call sequence deviates from the expected behavior, an attack is assumed. An apparent weakness of this approach is that it only takes into account the sequence of system call invocations and discards potential valuable information, such as system call arguments and return values. In addition, only a single application programming interface is examined (i.e., the interface that the operating system kernel exposes to user programs). Despite these shortcomings, researchers extended Forrest's initial work [Warrender et al. 1999; Wagner and Dean 2001; Feng et al. 2003] and system call sequences remain the most popular choice for analyzing program behavior.

Instead of analyzing system call sequences, this paper presents a novel anomaly detection approach that takes into account the information contained in *system call arguments*. We introduce several models that learn the characteristics of legitimate argument values and are capable of finding malicious instances. Based on the proposed models, we developed a host-based intrusion detection system that monitors running applications to identify malicious behavior. The system includes a novel technique for performing Bayesian classification of the outputs of individual detection models. This technique provides an improvement over the naïve threshold-based schemes traditionally used to combine model outputs.

Because we focus on the analysis of individual system calls, our technique is more resistant to *mimicry attacks* [Tan and Maxion 2002; Tan et al. 2002; Wagner and Soto 2002] than sequence-based approaches. A mimicry attack is an attack where the attacker can inject exploit code that imitates the system call sequence of a legitimate program run, but performs malicious actions.

The paper is structured as follows. Section 2 discusses related work. Section 3 presents our anomaly detection technique in detail and Section 4 describes the models we employ to perform the analysis of system call arguments. Section 5 shows our mechanism to aggregate the outputs of individual models for the purpose of classifying a system call as malicious or legitimate. Section 6 discusses implementation issues. Section 7 presents the experimental evaluation of the approach and Section 8 briefly concludes.

2. RELATED WORK

Many different anomaly detection techniques have been proposed to analyze different event streams. Examples include data mining on network traffic [Lee et al. 1999] and statistical analysis of audit records [Javitz and Valdes 1991].

The sequence of system calls produced by applications has also been the object of anomaly detection analysis. The techniques proposed so far fall into the areas of specification-based and learning-based approaches.

Specification-based techniques rely on application-specific models that are either written manually [Ko et al. 1997; Bernaschi et al. 2002; Chari and Cheng 2002] or derived using program analysis techniques [Wagner and Dean 2001]. [Goldberg et al. 1996] and [Provos 2003] describe systems that interactively create application-specific profiles with the help of the user. The profiles are then used as the input to a real-time intrusion detection system that monitors the corresponding application. When a non-conforming system call invocation is detected, an alarm is raised.

A major problem of specification-based systems is the fact that they exhibit only a very limited capability for generalizing from written or derived specifications. An additional disadvantage of hand-written specification-based models is the need for human interaction during the training phase. Although it is possible to include predefined models for popular applications, these might not be suitable for every user, especially when different application configurations are used. Systems that use automatically generated specifications, on the other hand, often suffer from significant processing overhead, caused by the complexity of the underlying models. For example, [Wagner and Dean 2001] reports a processing overhead of more than one hour for a single `sendmail` transaction.

Also, both classes of the specification-based approach often require access to the source code of an application. Recent work in [Giffin et al. 2004], however, addresses these drawbacks. The proposed model achieves levels of precision comparable to those in [Wagner and Dean 2001], with acceptable levels of overhead in most cases. Giffin's system is furthermore able to construct application models using binary static analysis, without access to the original source code. In general, however, models written or computed for specification-based systems are sensitive to changes in application source code. Such changes usually require that the specification be re-written or re-computed.

Learning-based techniques do not rely on any *a priori* assumptions about the applications. Instead, profiles are built by analyzing system call invocations during normal execution. An example of this approach is presented by Forrest [Forrest 1996]. During the training phase, the system collects all distinct system call sequences of a certain specified length. During detection, all actual system call sequences are compared to the set of legitimate ones, raising an alarm if no match is found.

This approach has been further refined in [Lee et al. 1997] and [Warrender et al. 1999], where the authors study similar models and compare their effectiveness to the original technique. However, these models do not take into account system call arguments. This particular shortcoming exposes these systems to mimicry attacks [Wagner and Dean 2001; Tan and Maxion 2002; Tan et al. 2002; Wagner and Soto 2002].

This paper introduces a learning-based anomaly detection system that analyzes the arguments of system calls. Thus, it is possible to considerably reduce the ability of an attacker to evade detection by imitating legitimate system calls sequences. Applying learning-based methods to system call arguments is a novel approach. Some existing anomaly detection systems do utilize system call arguments, however all of these systems are specification-based.

An approach to analyze program behavior by monitoring command-line arguments and program variables using neural networks is presented in [Ghosh et al. 1998]. This work is similar to ours in the sense that program behavior is modeled by analyzing argument and variable *values*, without taking system call sequences into account. The work is different in the way these values are modeled. In addition, in [Ghosh et al. 1998], only one command-line argument and one program variable are monitored, and both variables take on anomalous values during the execution of the only attack that the authors used to evaluate the effectiveness of their system. Our system, on the other hand, creates models for *all* arguments of security-relevant system calls.

3. SYSTEM OVERVIEW

The anomaly detection approach presented in this paper is based on the application-specific analysis of individual system calls. The input to the detection process consists of an ordered stream $S = \{s_1, s_2, \dots\}$ of system call invocations recorded by the operating system. Every system call invocation $s \in S$ has a return value r^s and a list of argument values $< a_1^s, \dots, a_n^s >$. Note that relationships between system calls or sequences of invocations are not taken into account.

For each system call used by an application, a distinct profile is created. Consider, for example, the `sendmail` application. The intrusion detection approach builds a profile for each of the system calls invoked by `sendmail`, such as `read`, `write`, `exec`, etc. Each of these profiles captures the notion of a "normal" system call invocation by characterizing "normal" values for one or more of its arguments.

The expected "normal" values for individual arguments are determined by models. A model is a set of procedures used to evaluate a certain feature of an argument, such as the length of a string. The argument type dictates which features can be evaluated by models. For example, while it useful to have a model that describes the distribution of characters for strings, this approach is not applicable to integers.

A model can operate in one of two modes, learning or detection. In learning mode, the model is trained and the notion of "normality" is developed by inspecting samples. Samples are values which are considered part of a regular execution of a program and are either derived directly from a subset of the input set S (learning on-the-fly) or provided by previous program executions (learning from a training set). It is important that the input to the training phase is as exhaustive and free from anomalous events as possible, although some models exhibit a certain degree of robustness against polluted or incomplete training data. The gathering of quality training data is a difficult problem by itself and is not discussed in this paper. We assume that a set of system call invocations that was created during normal operation is available. Section 7 describes how we obtained the training data for our experiments.

In detection mode, the task of a model is to return the probability of occurrence of an argument value based on the model's prior training phase. This value reflects the likelihood that a certain feature value is observed, given the established profile. The assumption is that feature values with a sufficiently low probability (i.e., abnormal values) indicate a potential attack. To classify the entire system call as normal or anomalous, the probability values of all models are aggregated. Section 5 discusses various ways to aggregate the probabilities and thus perform a classification of system calls as malicious or legitimate.

There are two main assumptions underlying our approach. The first is that attacks will appear in the arguments of system calls. If an attack can be carried out without performing system call invocations or without affecting the value of the arguments of such invocations, then our technique will not detect it. The second assumption is that the system call arguments used in the execution of an attack differ substantially from the values used during the normal execution of an application. If an attack can be carried out using system call argument values that are indistinguishable from the values used during normal execution then the attack will not be detected. The ability to identify abnormal values depends on the effectiveness and sophistication of the models used to build profiles for the system call features. Good models should make it extremely difficult to perform an attack without being detected.

Given the two assumptions above, we developed a number of models to characterize the features of system calls. We used these models to analyze attack data that escaped detection in previous approaches, data that was used in one of the most well-known intrusion detection evaluations [Lippmann et al. 2000], as well as data collected on a real Internet server. In all cases, our assumptions proved to be reasonable and the approach delivered promising results.

4. MODELS

This section introduces the models that are used to characterize system call arguments and to identify anomalous occurrences. For each model, we describe the learning phase and the detection phase. The former is the process that determines the model parameters associated with normal behavior, while the latter is the process of computing the probability of observing a system call argument appearing in the input, given the previously built model. This probability is then used to calculate an anomaly score for the argument in question.

When selecting the detection features for our models, we were naturally guided by current attacks. We analyzed how a large set of known attacks manifest themselves in system call arguments and selected a set of features that effectively detect these attack manifestations. Our evaluation shows that our anomaly system, utilizing these models, performs better than existing approaches. Of course, it is possible that future classes of attacks appear normal when evaluated according to these features. This problem can only be addressed by using a range of different detection features that focus on the structure of normal arguments as well as those that are geared towards the characteristic properties of known attack classes. The hope is then that future attacks lead to an observable deviation in at least one of the modeled system call features.

4.1 String Length

Usually, system call string arguments represent canonical file names that point to an entry in the file system. These arguments are commonly used when files are accessed (`open`, `stat`) or executed (`execve`). Their length rarely exceeds a hundred characters and they mostly consist of human-readable characters.

When malicious input is passed to programs, it is often the case that this input also appears in arguments of system calls. Consider, for example, an attack that exploits a format string vulnerability by inserting a string with a large number of substrings of the form "`%x`" in order to cause the formatted printing procedure in the victim application to overwrite a particular memory address. Suppose that a format string vulnerability is present in the log function of an application. Assume further that a failed open call is logged together with the file name. To exploit this kind of flaw, an attacker has to carefully craft a file name that triggers the format string vulnerability when the application attempts (and subsequently fails) to open the corresponding file. In this case, the exploit code manifests itself as an argument to the `open` call that contains a string with a length of several hundred bytes.

4.1.1 *Learning.* The goal of this model is to approximate the actual but unknown distribution of the lengths of a string argument and detect instances that significantly deviate from the observed normal behavior. Clearly, one cannot expect that the probability density function of the underlying real distribution would follow a smooth curve. One also has to assume that it has a large variance. Nevertheless, the model should be able to identify obvious deviations.

The mean $\dot{\mu}$ and the variance $\dot{\sigma}^2$ of the real string length distribution are approximated by calculating the sample mean μ and the sample variance σ^2 for the lengths $l_1, l_2, \ldots, l_n$ of the argument strings processed during the learning phase.

4.1.2 *Detection.* Given the estimated string length distribution with parameters μ and σ^2, it is the task of the detection phase to assess the regularity of an argument string with length l. The probability of l is calculated using the Chebyshev inequality.

$$p(|x - \mu| > t) < \frac{\sigma^2}{t^2} \tag{1}$$

The Chebyshev inequality puts an upper bound on the probability that the difference between the value of a random variable x and the mean μ of its corresponding distribution exceeds a certain threshold t (for an arbitrary distribution of the variable x with variance σ^2 and mean μ). Note that although this upper bound is symmetric around the mean, the underlying distribution is not restricted (indeed, our experimental data showed that the distribution of string lengths was not symmetric). When a string with length l is evaluated, we calculate the probability that any string is at least as long as the current one of length l. This is equivalent to the probability of a string deviating more from the mean than the current instance. To this end, the threshold t in Equation 1 is substituted with the difference between the string length l of the current instance and the mean μ of the string length distribution.

The probability value $p(l)$ for a string with length l (given that $l > \mu$) is then calculated as shown below. For strings shorter than or equal to μ, $p(l) = 1$.

$$p(l : l > \mu) = p(|x - \mu| > |l - \mu|) = \frac{\sigma^2}{(l - \mu)^2} \tag{2}$$

Only strings with lengths that exceed μ are assumed to be malicious. This is reflected in our probability calculation as only the upper bound for strings that are longer than the mean is relevant. Note that an attacker cannot disguise malicious input by padding the string and thus increasing its length, because an increase in length can only reduce the probability value.

We chose the Chebyshev inequality as a reasonable and efficient metric to model decreasing probabilities for strings with lengths that are increasingly greater than the mean. In contrast to schemes that define a valid interval (e.g., by recording all strings encountered during the training phase), the Chebyshev inequality takes the variance of the data into account and provides the increased resolution of gradually decreasing probability values (instead of a simple "yes/no" decision).

4.2 String Character Distribution

The string character distribution model captures the concept of a "normal" or "regular" string argument by looking at its character distribution. The approach is based on the observation that strings have a regular structure, are mostly human-readable, and almost always contain only printable characters.

A large percentage of characters in such strings are drawn from a small subset of the 256 possible 8-bit values (mainly from letters, numbers, and a few special characters). As in English text, the characters are not uniformly distributed, but occur with different frequencies. Obviously, it cannot be expected that the frequency distribution would be identical to standard English text. Even the frequency of a certain character (e.g., the frequency of the letter '`e`') varies considerably between different arguments. However, there are similarities between the character frequencies of arguments of legitimate system calls. This becomes apparent when the *relative frequencies* of all characters are sorted in descending order.

Our algorithm is based only on the frequency values themselves and does not rely on the distributions of particular characters. That is, it does not matter whether the character with the most occurrences is an '`a`' or a '`/`'. In the following, the *sorted*, relative character frequencies of a string are called its *character distribution*. For example, consider the text string "`passwd`" with the corresponding ASCII values of "112 97 115 115 119 100". The absolute frequency distribution is 2 for 115 and 1 for the four others. When these absolute counts are transformed into sorted, relative frequencies (i.e., the character distribution), the resulting values are 0.33, 0.17, 0.17, 0.17, 0.17 followed by 0 occurring 251 times.

For a string argument of a legitimate system call, one can expect that the relative frequencies slowly decrease in value (the path separator `'/'` often being the character with the most occurrences). In case of malicious input, however, the frequencies can drop extremely fast (because of a peak caused by a very high frequency of a single character) or nearly not at all (in case of a nearly uniform character distribution).

The character distribution of an argument that is perfectly normal (i.e., non-anomalous) is called the argument's *idealized character distribution ($\mathcal{ICD}$).* The idealized character distribution is a discrete distribution with:

$$\mathcal{ICD} : \mathfrak{D} \mapsto \mathfrak{P} \text{ with } \quad \mathfrak{D} = \{n \in \mathcal{N} | 1 \leq n \leq 256\}, \mathfrak{P} = \{p \in \mathfrak{R} | 0 \leq p \leq 1\}, \sum_{i=1}^{256} \mathcal{ICD}(i) = 1.0$$

The relative frequency of the character that occurs n-most often (1-most denoting the maximum) is given as $\mathcal{ICD}(n)$. When the character distribution of the sample string "`passwd`" is interpreted as the idealized character distribution, then $\mathcal{ICD}(1) = 0.33$ and $\mathcal{ICD}(2)$ through $\mathcal{ICD}(5)$ have a value of 0.17.

In contrast to signature-based approaches, the character distribution model has the advantage that it cannot be evaded by certain well-known techniques to hide malicious code inside a string. In fact, signature-based systems often contain rules that raise an alarm when long sequences of 0x90 bytes (the `nop` operation in Intel x86-based architectures) are detected in a packet. An intruder may substitute these sequences with instructions that have a similar behavior (e.g., `add rA,rA,0`, which adds 0 to the value in register A and stores the result back to A). By doing this, it is possible to prevent signature-based systems from detecting the attack. Such sequences, nonetheless, cause a distortion of the string's character distribution and, therefore, the character distribution analysis still yields a high anomaly score. In addition, characters in malicious input are sometimes disguised by `xor`'ing them with constants or shifting them by a fixed value (e.g., using the ROT-13 code). These evasion attempts do not change the resulting character distribution and the anomaly score of the analyzed system call argument is unaffected.

4.2.1 *Learning.* The idealized character distribution is determined during the training phase. First, the character distribution is stored for each observed argument string. The idealized character distribution is then approximated by calculating the average of all stored character distributions. This is done by setting $\mathcal{ICD}(n)$ to the mean of the n^{th} entry of the stored character distributions $\forall n : 1 \leq n \leq 256$. Because all individual character distributions sum up to unity, their average will do so as well. This ensures that the idealized character distribution is well-defined.

4.2.2 *Detection.* Given an idealized character distribution $\mathcal{ICD}$, the task of the detection phase is to determine the probability that the character distribution of an argument is an actual sample drawn from its $\mathcal{ICD}$. This probability, or more precisely, the confidence in the hypothesis that the character distribution is a sample from the idealized character distribution, is calculated using a statistical test.

This test should yield a high confidence in the correctness of the hypothesis for normal (i.e., non-anomalous) arguments while it should reject anomalous ones. A number of statistical tests can be used to determine the agreement between the idealized character distribution and the actual sample. We use a variant of the Pearson χ^2-test as a "goodness-of-fit" test [Billingsley 1995]. This test was chosen because it is a simple and efficient way to assess the "normality" of the character distribution.

The χ^2-test requires that the function domain is divided into a small number of intervals, or bins, and it is preferable that all bins contain at least "some" elements (the literature considers five elements to be sufficient for most cases). As the exact division of the domain does not significantly influence the outcome of the test, we have chosen the six segments for the domain of $\mathcal{ICD}$ as follows: {[1], [2,4], [5,7], [8,12], [13,16], [17,256]}. Although the choice of these six bins is somewhat arbitrary, it reflects the fact that the relative frequencies are sorted in descending order. Therefore, the values of $\mathcal{ICD}(x)$ are higher when x is small, and thus all bins contain some elements with a high probability.

When a new system call argument is analyzed, the number of occurrences of each character in the string is determined. Afterward, the values are sorted in descending order and combined by aggregating values that belong to the same bin. The χ^2-test is then applied to calculate the probability that the given sample was drawn from the idealized character distribution. The derived probability value p is used as the return value for this model. When the probability that the sample is drawn from the idealized character distribution increases, p increases as well.
The standard test requires the following steps to be performed.

(1) *Calculate the observed and expected frequencies* - The observed values O_i (one for each bin) are already given. The expected number of occurrences E_i are calculated by multiplying the relative frequencies of each of the six bins as determined by the $\mathcal{ICD}$ times the length of the argument (i.e., the length of the string).

(2) *Compute the χ^2-value* as $\chi^2 = \sum_{i=0}^{i<6} \frac{(O_i - E_i)^2}{E_i}$ - note that i ranges over all six bins.

(3) *Determine the degrees of freedom and obtain the significance* - The degrees of freedom for the χ^2-test are identical to the number of addends in the formula above minus one, which yields five for the six bins used. The actual probability p that the sample is derived from the idealized character distribution (that is, its significance) is read from a predefined table using the χ^2-value as index.

The result of this test are used directly to assign an anomaly score to the model's input.

4.3 Structural Inference

Often, the manifestation of an exploit is immediately visible in system call arguments as unusually long strings or strings that contain repetitions of non-printable characters. There are situations, however, when an attacker is able to craft her attack in a manner that makes its manifestation appear more regular. For example, non-printable characters can be replaced by groups of printable characters. In such situations, we need a more detailed model of the system call argument. This model can be acquired by analyzing the argument's structure. For our purposes, the structure of an argument is the regular grammar that describes all of its normal, legitimate values. Structural inference is the process by which this grammar is inferred by analyzing a number of legitimate strings during a training phase.

For example, consider the first argument of the `open` system call. It is a null-terminated character string that specifies the canonical name of the file that should be opened. Assume that during normal operation, an application only opens files that are located in the application's home directory and its subdirectories. For this application, the structure of the first argument of the `open` system call should reflect the fact that file names always start with the absolute path name to the program's home directory followed by a (possibly empty) relative path and the file name. In addition, it can be inferred that the relative path is an alternation of slashes and strings. If the directory names consist of lowercase characters only, this additional constraint can be determined as well. When an attacker exploits a vulnerability in this application and attempts to open an "anomalous" file such as "`/etc/passwd`", an alert should be raised, as this file access does not adhere to the inferred pattern.

4.3.1 *Learning.* When structural inference is applied to a system call argument, the resulting grammar must be able to produce at least all training examples. Unfortunately, there is no unique grammar that can be derived from a set of input elements. When no negative examples are given (i.e., elements that should not be derivable from the grammar), it is always possible to create either a grammar that contains exactly the training data or a grammar that allows production of arbitrary strings. The first case is a form of over-simplification, as the resulting grammar is only able to derive the learned input without providing any level of abstraction. This means that no new information is deduced. The second case is a form of over-generalization because the grammar is capable of producing all possible strings, but there is no structural information left.

The basic approach used for our structural inference is to generalize the grammar as long as it seems to be "reasonable" and stop before too much structural information is lost. The notion of "reasonable generalization" is specified with the help of Markov models and Bayesian probability.

In a first step, we consider the set of training items as the output of a *probabilistic* grammar. A probabilistic grammar is a grammar that assigns probabilities to each of its productions (i.e., some words are more likely to be produced than others). This fits well with the evidence gathered from system calls, as some system call argument values appear more often, representing important information that should not be lost in the modeling step.

A probabilistic regular grammar can be transformed into a non-deterministic finite automaton (NFA). Each state S of the automaton has a set of n_S possible output symbols o which are emitted with a probability of $p_S(o)$. Each transition t is marked with a probability $p(t)$ that characterizes the likelihood that the transition is taken. An automaton that has probabilities associated with its symbol emissions and its transitions can also be considered a Markov model.

The output of the Markov model consists of all paths from its start state to its terminal state. A probability value can be assigned to each output word w (that is, a sequence of output symbols $o_1, o_2, \ldots, o_k$). This probability value (as shown in Equation 3) is calculated as the sum of the probabilities of all distinct paths through the automaton that produce w. The probability of a single path is the product of the probabilities of the emitted symbols $p_{S_i}(o_i)$ and the taken transitions $p(t_i)$. Note that the probabilities of all possible output words w sum up to 1.

$$p(w) = p(o_1, o_2, \ldots, o_k) = \sum_{(paths\ p\ for\ w)} \prod_{(states\ \in\ p)} p_{S_i}(o_i) * p(t_i) \tag{3}$$

For example, consider the NFA in Figure 1. The probabilities associated with each transition ($p(t_i)$) are labelled on the edges in the graph. Similarly, the probabilities associated with emitting a particular symbol ($p_{S_i}(o_i)$) are given in each node in the graph. To calculate the probability of the word "ab", one has to sum the probabilities of all possible paths that produce this string (in this case there are two, one that follows the left arrow and one that follows the right one). The start state emits no symbol and has a probability of 1. Following Equation 3, the result is

$$\begin{aligned} p(w) &= (1.0 * 0.3 * 0.5 * 0.2 * 0.5 * 0.4) + \\ &\quad (1.0 * 0.7 * 1.0 * 1.0 * 1.0 * 1.0) \\ &= 0.706 \end{aligned} \tag{4}$$

The target of the structural inference process is to find a NFA that has the highest likelihood for the given training elements. An excellent technique to derive a Markov model from empirical data is explained in [Stolcke and Omohundro 1993]. It uses Bayes' theorem to state this goal as

$$p(Model|TrainingData) = \frac{p(TrainingData|Model) * p(Model)}{p(TrainingData)} \tag{5}$$

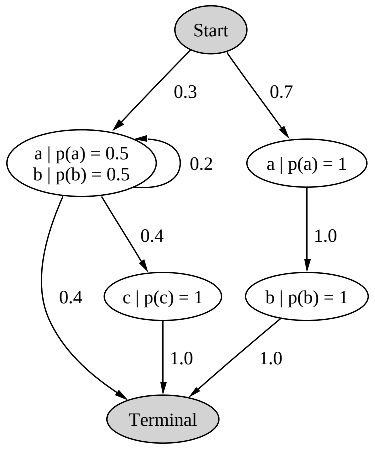

Fig. 1. Markov model example.

The probability of the training data is considered a scaling factor in Equation 5 and it is therefore ignored. As we are interested in maximizing the *a posteriori* probability (i.e., the left-hand side of the equation), we have to maximize the product shown in the enumerator on the right-hand side of the equation. The first term, which is the probability of the training data given the model, can be calculated for a certain automaton by adding the probabilities calculated for each input training element as discussed above. The second term, which is the prior probability of the model, is not as straightforward. It has to reflect the fact that, in general, smaller models are preferred. The model probability is calculated heuristically and takes into account the total number N of states as well as the number of transitions t_S and emissions o_S at each state S. This is justified by the fact that smaller models can be described with fewer states as well as fewer emissions and transitions. The actual value is derived from Equation 6 as follows:

$$p(Model) \;\; \alpha \prod_{S \in States} N^{-(\sum_{S \in States} t_S)} * N^{-(\sum_{S \in States} o_S)} \tag{6}$$

The product of the probability of the model given the data times the prior probability of the model itself (i.e., the term that is maximized in Equation 5) reflects the intuitive idea that there is a conflict between simple models that tend to over-generalize and models that perfectly fit the data but are too complex. Models that are too simple have a high model probability, but the likelihood for producing the training data is extremely low. This results in a small product when both terms are multiplied. Models that are too complex have a high likelihood of producing the training data (up to 1 when the model only contains the training input without any abstractions), but the probability of the model itself is very low. By maximizing the product, the Bayesian model induction approach creates automatons that generalize enough to reflect the general structure of the input without discarding too much information.

The model building process starts with an automaton that exactly reflects the input data and then gradually merges states. This state merging is continued until the *a posteriori* probability no longer increases. The interested reader is referred to [Stolcke and Omohundro 1993] and [Stolcke and Omohundro 1994] for details.

4.3.2 *Detection.* Once the Markov model has been built, it can be used by the detection phase to evaluate string arguments. When the word is a valid output from the Markov model, the model returns 1. When the value cannot be derived from the given grammar, the model returns 0.

4.4 Token Finder

The purpose of the token finder model is to determine whether the values of a certain system call argument are drawn from a limited set of possible alternatives (i.e., argument values are tokens or elements of an enumeration). An application often passes identical values such as flags or handles to certain system call arguments. When an attack changes the normal flow of execution and branches into maliciously injected code, such constraints are often violated. When no enumeration can be identified, it is assumed that the values are randomly drawn from the argument type's value domain (i.e., random identifiers for every system call).

4.4.1 *Learning.* The classification of an argument as an enumeration or as a random identifier is based on the observation that the number of different occurrences of argument values is bound by some unknown threshold t in the case of an enumeration, while it is unrestricted in the case of random identifiers. Obviously, t is considered to be significantly smaller than the number of distinct values of a certain domain (such as the number of different integer values that can be represented on the underlying machine architecture). Otherwise, every argument type could be considered a huge enumeration itself.

When the number of different argument instances grows proportional to the total number of arguments, the use of random identifiers is indicated. If such an increase cannot be observed and the number of different identifiers follows a standard diminishing growth curve [Lee et al. 2002], we assume an enumeration. In this case, the complete set of identifiers is stored for the subsequent detection phase.

The decision between an enumeration and random identifiers can be made utilizing a simple statistical test, such as the non-parametric Kolmogorov-Smirnov variant as suggested in [Lee et al. 2002]. That paper discusses a problem similar to our token finder for arguments of SQL queries and the solution proposed by the authors can be applied to our model.

4.4.2 *Detection.* When it has been determined that the values of a system call argument are tokens drawn from an enumeration, any new value is expected to appear in the set of known identifiers. When it does, 1 is returned, otherwise the model returns 0. When it is assumed that the argument values are random identifiers, the model always returns 1.

5. SYSTEM CALL CLASSIFICATION

The task of a model m_i, associated with a certain system call, is to assign an anomaly score as_i to a single argument of an invocation of the system call. This anomaly score is calculated (and reflects) the probability of the occurrence of the given argument value with regards to an established profile. Based on the anomaly score outputs $\{as_i | i = 1 \dots k\}$ of k models $M = \{m_1, \dots, m_k\}$ and possibly additional information I, the decision must be made whether a certain system call invocation is malicious (anomalous) or legitimate (normal). This decision process is called system call classification.

Given the definitions introduced above, system call classification can be defined more formally as a function C that, for a certain system call s with a set of arguments, accepts as input the corresponding model outputs (i.e. anomaly scores) $\{as_i | i = 1 \dots k\}$ and additional information I. The result of this classification function is a binary value that identifies the system call s as normal or anomalous. That is, for a certain system call s, the function call classification function C is defined as follows.

$$C_s(as_1, as_2, \dots, as_k, I) = \{\text{normal}, \text{anomalous}\} \tag{7}$$

In most current anomaly-based intrusion detection systems, C is a simple function that calculates the sum of the anomaly scores as_i and compares the result to a threshold represented by I. That is, C is defined as follows.

$$C(as_1, as_2, \ldots, as_k, I) \;=\; \begin{cases} sc \text{ is normal : } & \sum_{i=1}^{k} as_i \leq I \\ sc \text{ is anomalous : } & \sum_{i=1}^{k} as_i > I \end{cases} \tag{8}$$

In our anomaly detection system, this simple summation scheme is replaced by a Bayesian network (for a good introduction to Bayesian networks, refer to [Jensen 2001]). This network consists of a root node (i.e., the hypothesis node) that represents a variable with two states, namely normal and anomalous. In addition, one child node is introduced for each model (called a model node) to capture the model's respective outputs, that is $\{as_i | i = 1 \ldots k\}$. The root node is connected to each child node, reflecting the fact that the aggregate score is dependent upon the individual model scores.

Depending on the domain, there might be causal dependencies between models that require appropriate links to be introduced into the network. One example is a positive or a negative correlation between models (i.e., one anomalous argument makes it more or less likely that another one is also anomalous). Another example is the situation where the output of one model indicates that the quality of a test performed by another model is reduced.

Additional information sources might indicate that anomalous behavior is in fact legitimate or might support the decision that observed behavior is malicious. This could be information from other intrusion detection systems or system health monitors (e.g., CPU utilization, memory usage, or process status).

An important piece of additional information is the confidence value associated with each model. Depending on the system calls, a certain argument might not be very suitable to distinguish between attacks and legitimate behavior. It might be the case that the same values of this argument appear in both legitimate and malicious behavior or that the variance is very high. In these situations, it is useful to reduce the influence of the model output on the final decision.

The confidence in the output of a model is an indication of the expected accuracy of this model. In traditional systems, the confidence is often neglected or approximated with static weights. When a model is expected to produce more accurate results, it receives a higher *a priori* weight. However, this is not sufficient, as the confidence in a model can vary depending on the training data used to create the corresponding profile. Consider, for example, the token finder model. When this model detects an enumeration during the learning phase, its anomaly scores are considered highly accurate. When random identifiers are assumed, the anomaly score is not meaningful.

In the Bayesian network used in our system, each model confidence is represented by a node that is connected to its corresponding model node. When models create their profiles of normal behavior, the variance of the input training data is evaluated. When the variance of the analyzed feature is high, a low confidence value is assumed. When a small, coherent set of feature values is observed during the training, the confidence in the correctness of the model output is high. Note that these additional nodes require a non-naïve Bayesian network (i.e., a network with nodes that have more than one parent node). This is because model nodes have the root node and the corresponding confidence node as parents.

The next section shows an example of a Bayesian network and the model dependencies that have been identified for the `open` system call. In Section 7, experimental results are presented that show the advantage of combining model outputs using an approach based on Bayesian networks over a simple threshold-based scheme.

6. IMPLEMENTATION

Using the models presented in the previous section, we have implemented an intrusion detection system (IDS) that detects anomalies in system call arguments. The system retrieves input events (i.e., system call invocations) from an operating system auditing facility (in the case of Linux) or from audit logs (as in Solaris' Basic Security Module, or BSM). It then utilizes the models to compute anomaly scores for the monitored system call arguments and finally classifies each call as malicious or legitimate using a Bayesian network.

The intrusion detection system monitors a selected number of security-critical applications. These are usually programs that require `root` privileges during execution such as server applications and `setuid` programs. For each program, the IDS maintains data structures that characterize the normal profile of every monitored system call. A system call profile consists of a set of models for each system call argument and a function that calculates the anomaly score for input events from the corresponding model outputs.

The architecture of our system and an overview of the relationship between applications, profiles, models and system calls are depicted in Figure 2. System calls are made available to the system through the audit facility, and are issued to the appropriate application-specific modules. Application modules then forward the system call event to the profile specific to the system call in question, which in turn maps the system call argument values to trained model instances. The figure shows the dispatch of an `open` system call issued by the `ftpd` daemon to the corresponding profile and the delivery of its arguments to the appropriate models.

An open-source auditing facility called Snare [Snare 2003] is used to obtain system call records under Linux. It is implemented as a dynamically loadable kernel module and can be installed without changing or recompiling the operating system (given that support for loadable kernel modules is available). Auditing is introduced by exchanging the original function pointer entries in the kernel system call table with pointers to special wrapper functions provided by Snare. Whenever an audited system call is invoked, the corresponding wrapper function gets called and the system call is logged together with its arguments as a Snare record object. Then, the control is transferred to the original system call handler. Snare records are passed to the intrusion detection system through an entry in the `/proc` file system.

In addition to the Snare auditing facility, a BSM audit module has been implemented to permit the analysis of system calls in the Solaris operating system. This module converts Solaris' BSM data into a format comparable to Snare's and allows us to process BSM audit files. This translation module allowed us to evaluate our detection techniques with respect to the well-known MIT Lincoln Labs data set [Lippmann et al. 2000], whose operating system audit data is only available as BSM traces.

For performance reasons, an important decision is selecting a subset of system

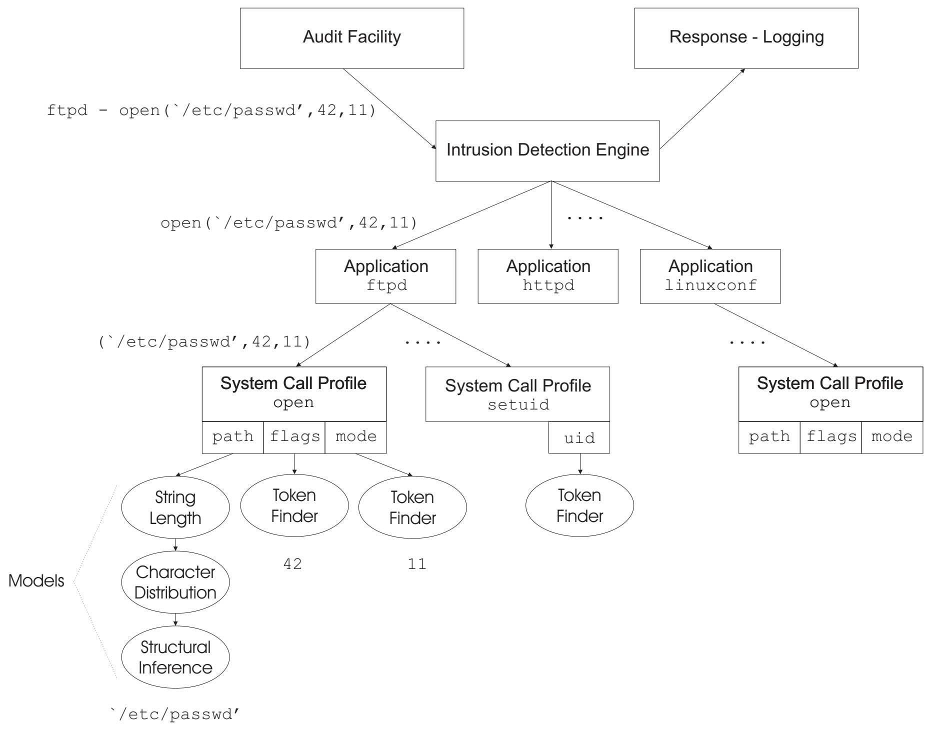

Fig. 2. Intrusion detection system architecture.

calls to be monitored by our intrusion detection system. If a larger number of system calls are monitored, the system can base its decisions on more complete input data. However, this might have a negative impact on performance as more events have to be processed. In [Axelsson et al. 1998], it is argued that it is possible to detect a large set of intrusions by observing only the `execve` system call and its arguments. Starting from there, the analysis of several exploits has shown that attacks also frequently manifest themselves in the arguments of `open`, `setuid`, `chmod`, `chown`, and `exit` calls. Snare's audit facility records a number of system calls in addition to these, however, it was noted that these additional system calls appear very infrequently in system call traces (e.g., `mount` and `mkdir`). We therefore decided to include all system calls that are audited by Snare.

After the set of audited system calls had been determined, suitable models had to be selected for their arguments. For our purposes, arguments can be divided into four different categories: *file name*, *user id*, *flags*, and *execution parameters*. *File names* are of type `string` and represent the canonical names of files including their paths. *User ids* are of type `integer` and describe arguments that refer to the various system identifiers used for users and groups. *Flags* are of type `integer` and refer to mode identifiers or sets of flags that influence the behavior of a system call. *Execution parameters* are of type `string` and describe the parameters of the `execve` system call.

We use the string length model, the string character distribution model, and the structural inference model to characterize the features of `string` arguments. The string length model and the character distribution model are straightforward to apply, while the structural inference model requires some preprocessing of the input.

String arguments are not directly inserted into the structural inference model as sequences of characters. Instead, every string passes through two processing steps. First, each character is replaced by a token that corresponds to a character class. We define the main character classes as `digit`, `uppercase` letter and `lowercase` letter. Characters that do not belong to one of the aforementioned categories are considered to be a class of their own. Then, repetitions of directly adjacent items that belong to the same character class are fused into single occurrences of the appropriate token. For example, consider the string "`/etc/passwd`". After the first step, both slashes would remain, but the characters of the two words are replaced by three and five `lowercase` tokens, respectively. Then, adjacent, identical tokens are merged, and the string is transformed into the sequence "`slash - lowercase - slash - lowercase`". This sequence is the input passed to the structural inference model.

The reason for this preprocessing step is the fact that it is more efficient for the inference process to determine general structures when the input size is small and domain-specific knowledge has been used to perform an initial classification. For human-readable strings, it is sensible to emphasize the appearance of special characters and combine regular letters or digits into single structural elements.

The token finder model can be applied to `string` and `integer` arguments. However, it is mostly used for *flags* and *user ids* because values for *flags* and *user ids* are often drawn from a limited set of tokens and deviations indicate anomalous behavior. For example, consider a web server that, during normal operation, only calls `setuid` with the identifier of a user with limited privileges. Then, the invocation of `setuid` with a different argument, such as the `root` user, should be reported as suspicious. A similar consideration also applies to the argument of the `exit` call. Usually, applications either report their successful termination or return one of a few possible errors. Unexpected deviations are usually the effect of anomalous activity which is often caused by malicious intent.

Given the models for the different arguments of the monitored system calls, a suitable Bayesian network can be constructed. As an example, Figure 3 shows the structure of the Bayesian networks for the `open` and `execve` system calls. Both system calls have two arguments. Three models (string length, string character distribution and structural inference) are attached to the first `string` argument (a *file name* argument in the case of the `open` call, and *execution parameters* in the case of the `execve` call). The token finder model is attached to the `integer` argument in the case of the `open` call (*flags*) and to another `string` argument in case of the `execve` call (a *file name* argument referring to the program image executed). The causal relationships between individual model scores that are encoded in the network in Figure 3 are explored in detail in Section 7.2. Similar but simpler networks are used for other monitored system calls that have only a single argument. A different Bayesian network instance is utilized for each system call; however, most of these networks have an identical structure.

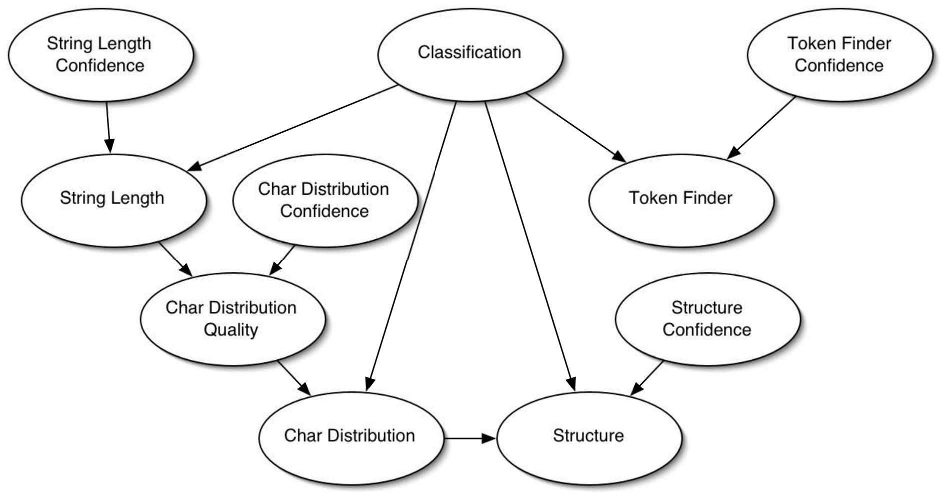

Fig. 3. Bayesian network for `open` and `execve` system calls.

In addition to the structure, conditional probability tables (CPTs) were specified for each node of a Bayesian network. We used domain-specific knowledge to estimate appropriate probability values for the various tables. For each node, one has to provide the probabilities for all states of the corresponding variable, conditionally dependent on the states of all parent nodes. When a suitable structure of the network has been chosen, these probabilities are mostly intuitive and can be determined in a sufficiently accurate way by a domain expert. Note that we have not tuned the CPTs in any way for our experiments. The probabilities were selected before the evaluation began and were not modified thereafter.

The output of a model is a probability in the interval [0,1] that describes the deviation of the system call argument from the expected normal value described by the learned model. This probability value is mapped onto one of five possible anomaly score states that are associated with each model node in the network. The mapping of a continuous function output onto a number of different states is called *discretization*. This process is required to keep the CPTs of the Bayesian network manageable and to allow efficient calculations of the probabilities at the root node. As shown in Table I, model outputs close to zero indicate normal arguments while outputs close to one indicate anomalous ones.

Table I. Anomaly score intervals.

Probability Range	Anomaly Score
$[0.00, 0.50)$	Normal
$[0.50, 0.75)$	Uncommon
$[0.75, 0.90)$	Irregular
$[0.90, 0.95)$	Suspicious
$[0.95, 1.00]$	Very Suspicious

The Bayesian network in Figure 3 shows the two model dependencies that we have introduced into our intrusion detection system. One dependency connects the node corresponding to the output of the string length model to the quality of the character distribution (which is also influenced by the confidence in the output of the character distribution). The mediating node `Char Distribution Quality` in our network expresses the idea that the quality of the anomaly score calculated by the character distribution is not only dependent on the *a priori* confidence of the model in the quality of its learned model, but also on the length of the string that is currently analyzed. When this string is very short, the quality of the statistical test that assesses the character distribution is significantly reduced. This is reflected by the conditional probability tables of the `Char Distribution Quality` node.

The other dependency is introduced between the nodes representing the character distribution and the structure model. The reason is that an "abnormal" character distribution is likely to be reflected in a structure that does not conform to the learned grammar. This is an example of a simple positive correlation of output values between models. In Section 7.2, a quantitative evaluation is presented that supports our belief that these two model dependencies are in fact reasonable.

During the analysis phase, the output (i.e., anomaly scores) of the four models and their confidences are entered as evidence into the Bayesian network. The output of the network is computed at the `Classification` node. The probabilities of the two states (normal, anomalous) associated with the output node are calculated. When the probability of an event being anomalous is high enough, an alarm is raised. Note that this requirement (i.e., a probability value needs to be "high enough" to raise an alarm) could be interpreted as a threshold as well. However, unlike simple threshold-based approaches, this probability value directly expresses the probability that a certain event is an attack, given the specific structure of the Bayesian network. The sum of model outputs in a threshold-based system, on the other hand, is not necessarily proportional to the probability of an event being an attack. It is possible, due to the assumption of independence of model outputs and the potential lack of confidence information in these systems, that the sum of the outputs is increasing while the true probability of an attack is, in fact, decreasing.

Both the threshold in a traditional system and the notion of a sufficiently high probability for raising an alarm in the Bayesian approach can be utilized to tune the sensitivity of the intrusion detection system. However, the result of the Bayesian network directly reports the probability that an event is anomalous, given the model outputs and the structure of the network, while a simple summation of model outputs is only an approximation of this probability. The difference between the exact value and the approximation is important, and accounts for a significant number of false alarms, as shown in Section 7.

All detection models used by our system are implemented as part of a general library. This library, called *libAnomaly*, provides a number of useful abstract entities for the creation of anomaly-based intrusion detection systems and makes frequently-used detection techniques available. The library was created in response to the observation that almost all anomaly-based IDSs have been developed in an ad-hoc way, with much basic functionality implemented from scratch for each new prototype.

7. EVALUATION

This section details the experiments undertaken to evaluate the classification effectiveness and performance characteristics of our intrusion detection system. The goal of our system is to provide reliable classification of system call events in a performance-critical server environment. Additionally, the validity of the Bayesian network structure proposed for combining individual model scores is explored.

7.1 Classification Effectiveness

In this section, the ability of our intrusion detection system to correctly distinguish attacks from events associated with normal system usage is investigated. Accuracy of detection is especially important for anomaly-based systems as they are prone to the generation of false alarms. Often, excessive false positives have the effect of making the system unusable by desensitizing the system administrator. To validate the claim that our detection technique is accurate, a number of experiments were conducted.

For the first experiment, we ran our system on the well-known 1999 MIT Lincoln Lab Intrusion Detection Evaluation Data [Lippmann et al. 2000]. We used data recorded during two attack free weeks (Week 1 and Week 3) to train our models and then performed detection on the test data that was recorded during two subsequent weeks (Week 4 and Week 5). Week 2 was not considered for model training since it contained attacks.

The truth file provided with the evaluation data lists all attacks carried out against the network installation during the test period. When analyzing the attacks, it turned out that many of them were reconnaissance attempts such as network scans or port sweeps, which are only visible in the network dumps and do not leave any traces in the system call logs. These network-based events cannot be detected by our system as it focuses only on host-based events.

Another class of attacks are policy violations. These attacks do not allow an intruder to elevate privileges directly. Instead, they help to obtain information that is classified as secret by exploiting some system misconfiguration. This class of attacks contains intrusions that do not exploit a weakness of the system itself, but rather take advantage of a mistake that an administrator made in setting up the system's access control mechanisms. Such incidents are not visible for our system either, as information is leaked by "normal" but unintended use of the system.

Table II. 1999 MIT Lincoln Lab evaluation results.

Application	Total System Calls	Attacks	Identified Attacks	False Alarms
`eject`	138	3	3 (14)	0
`fdformat`	139	6	6 (14)	0
`ffbconfig`	21	2	2 (2)	0
`ps`	4,949	14	14 (55)	0
`ftpd`	3,229	0	0	14
`sendmail`	71,743	0	0	8
`telnetd`	47,416	0	0	17
`Total`	127,635	25	0	39

The most interesting class of attacks are those that exploit a vulnerability in a remote or local service, allowing an intruder to elevate her privileges. The MIT Lincoln Lab data contains 25 instances of attacks that exploit buffer overflow vulnerabilities in four different programs. Table II summarizes the results produced by the system for the attacks against these four programs, namely `eject`, `ps`, `fdformat`, and `ffbconfig`. In addition, we present results for interesting daemon and `setuid` programs to assess the number of false alarms. The *Total* column shows the sum of all system calls that are executed by the corresponding program and analyzed by our system. The *Attacks* column shows the number of attacks against the vulnerable programs in the data set. *Identified Attacks* states the number of attacks that were successfully detected by our system and, in parentheses, the number of corresponding system calls that were labeled as anomalous. It is very common that a single attack results in a series of anomalous system calls. The *False Alarms* column shows the number of program traces that were flagged as anomalous although these invocations are not related to any attack.

In addition to analyzing the identified attacks and false positives of our own system, we have compared the detection accuracy to four approaches that were previously suggested in the literature. All four techniques are based on the analysis of system call sequences and work with unlabeled data. We selected unsupervised intrusion detection systems that operate on unlabeled training data to allow a fair comparison to our technique, which also does not require labeled input. Also, the need for labeled input significantly limits the usefulness of a system because such data is almost never available in practice.

The first system used for our experiments is the approach proposed by Forrest [Forrest 1996], which uses a sliding window of fixed length n over the system call traces. All sequences of length n that occur during the training period are added to a database of normal behavior. During detection, each observed sequence of length n is checked against this database. When the lookup fails, an alert is raised. The second system [Kang et al. 2005] extends the system call sequences to bags of system calls. In this representation, the last n system calls are not treated as an ordered sequence but as a set. Because the order information between system calls is lost, the technique produces less false positives. This is paid for by a higher number of missed attacks. The third and fourth system [Portnoy et al. 2001] use machine learning techniques to identify outliers in a high-dimensional vector space. One approach is based on the k-nearest neighbor classification scheme, the other approach uses cluster-based estimation.

Table III shows the results for the four intrusion detection approaches and our proposed system when run on the MIT Lincoln Lab data. For each detection approach, the false negative (`FN`) column shows the number of real attacks missed, while the false positive column (`FP`) shows the number of traces misclassified as attack. It can be seen that our system is the only one that detects all attacks (that is, it has no false negatives), while it produces the least number of false positives for most application traces.

As mentioned in Section 6, each Bayesian network requires a probability threshold that allows it to distinguish between attacks and legitimate system calls. The results for this and all following experiments are obtained by classifying a system call as an attack when the root node of the Bayesian network shows more than 50%

Table III. 1999 MIT Lincoln Lab evaluation results.

Application	Sequences		Syscall Bags		K-Nearest		Cluster		Our System	
	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP
`eject`	1	1	1	1	2	1	0	1	0	0
`fdformat`	2	0	2	0	0	0	0	0	0	0
`ffbconfig`	0	0	0	0	0	0	0	0	0	0
`ps`	0	12	0	0	0	47	12	25	0	0
`ftpd`	0	21	0	15	0	21	0	20	0	14
`sendmail`	0	75	0	1	0	89	0	106	0	8
`telnetd`	0	99	0	99	0	21	0	6	0	17
`Total`	3	208	3	116	2	179	12	158	0	39

probability that the system call is anomalous. This threshold is not necessarily optimal. Figure 4 shows the Receiver Operating Characteristic (ROC) curve of our system for the MIT Lincoln Lab data. The ROC of a classifier shows its performance as a trade off between selectivity and sensitivity; a curve of the false positive rate versus the true positive rate is plotted, while a sensitivity or threshold parameter is varied. Ideally, a classifier has a true positive rate of 1 and a false positive rate of 0. The ROC curve for the Bayesian event classifier is plotted by varying the probability value at the root node of the Bayesian network that is required for an event to be reported as an attack. When a threshold of 50% is used, all attacks are detected, but 39 system calls are incorrectly reported as malicious (resulting in a false positive rate of $39/(127,635 - 85) \approx 3.06 \times 10^{-4}$). It can be seen in Figure 4 that a threshold exists where all attacks are detected and the false positive rate is only slightly greater than 2×10^{-4} (resulting from only 28 incorrectly classified system calls).

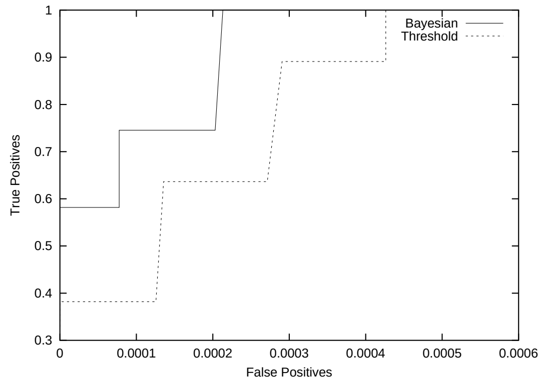

Fig. 4. ROC comparison between Bayesian network and threshold-based system.

Figure 4 also depicts the ROC curve of a threshold-based system, which classifies system calls by calculating the sum of the individual model outputs and then comparing this sum to a threshold. The ROC curve for this classifier is determined by varying the threshold that is compared to the sum of outputs. Although both classification approaches receive identical input (i.e., the outputs of the individual models), the Bayesian system consistently performs better. The graphs show that both classifiers output some false alarms when all attacks are correctly detected. However, when all attacks are correctly detected (i.e., the true positive rate is 1), the Bayesian approach only reports half as many false positives.

When analyzing the false positives raised by both classification approaches, we observed that the Bayesian scheme always reported a subset of the false alarms raised by the threshold-based mechanism. The false positives common to both approaches are caused by system call invocations that have arguments that significantly deviate from all examples encountered during the training phase. This is due to the fact that the training data was very homogeneous, leading to profiles that were very sensitive to changes. During the detection phase, legitimate system calls with significantly different arguments were observed. This resulted in their incorrect classification.

The system calls that were reported as anomalous by the threshold-based approach but correctly classified as normal by the Bayesian scheme were instances with short string arguments. As explained in Section 6, short strings can significantly influence the quality of the character distribution model, causing it to report incorrect anomalies. This problem is addressed by the Bayesian network using the mediating `Char Distribution Quality` node (refer to Figure 3), correctly evaluating these system calls as normal. Note that the shapes of the curves in Figure 4 are not a consequence of an insufficient number of data points. The horizontal and vertical segments contain intermediate points, reflecting changes in either the false positive or the true positive rate alone.

The second experiment was performed to evaluate the ability of our system to detect a number of recent attacks. Four network daemon programs (`wuftpd`, `Apache`, `OpenSSH`, and `sendmail`) and one `setuid` tool (`linuxconf`) were installed to simulate a typical Internet server. After the test environment was prepared, the intrusion detection system was installed and trained for about one hour. During the training period, we attempted to simulate normal usage of the system. Then, the intrusion detection system was switched to detection mode and more extensive tests were conducted for five more hours. No malicious activity took place. After that, we carried out three actual exploits against the system, one against `wuftpd`, one against `linuxconf` and one against `Apache`. All of them were reliably detected. As our system is currently not able to automatically determine when enough training data has been processed, the duration of the training period was chosen manually.

Table IV shows, for each application, the number of analyzed system calls, the number of detected attacks (with the number of system calls labelled anomalous by the system are given parenthetically), and the number of false alerts. An analysis of the reported false alarms confirmed that all alarms were indications of anomalous behavior that was not encountered during the training phase. Although the anomalous situations were not caused by malicious activity, they still represent deviations from the "normal" operation presented during the learning process. Many useful generalizations took place automatically and no alerts were raised when new files were accessed. However, the login of a completely new user or the unexpected termination of processes were still considered anomalous and therefore flagged as malicious.

Table IV. Detection accuracy in controlled environment.

Application	Total System Calls	Attacks	Identified Attacks	False Alarms
wuftpd	4,887	1	1 (86)	1
Apache	17,274	1	1 (2)	0
OpenSSH	9,562	0	0 (0)	6
sendmail	15,314	0	0 (0)	5
linuxconf	4,422	1	1 (16)	3
Total	51,459	3	3 (104)	15

The 7350wu attack exploits an input validation error of wuftpd [advisory-ftpd 2000]. It was chosen because it was used by Wagner and Soto [Wagner and Soto 2002] as the basis for a mimicry attack to evade detection by current techniques based on system call sequences. Our IDS labeled 86 system calls present in the trace of the 7350wu attack as anomalous, all of which were directly related to the intrusion. 84 of these anomalies were caused by arguments of the execve system call that contained binary data and were not structurally similar to argument values seen in the training data.

The large number of anomalous events is due to the fact that the 7350wu code includes a feature for discovering working parameters for the exploit via a brute force technique that repeatedly probes the FTP daemon. Not all of these 84 calls would have been necessary, were the working parameters known in advance. However, the actual exploit needs to execute at least one of them to function properly, and this invocation would be detected.

It should be noted that none of these anomalies would be missing were the exploit disguised using the mimicry technique suggested by Wagner and Soto [Wagner and Soto 2002]. Since each system call is examined independently, the insertion of intervening system calls to modify their sequence does not affect the classification of the others as anomalies. This shows that our technique is not affected by attempts to imitate normal system call sequences. This does not imply that our IDS is immune to all possible mimicry attacks (e.g., mimicry attacks that imitate legitimate system call arguments). However, by combining our system with a sequence-based approach, the potential attack space is reduced significantly because an attacker would have to subvert both systems.

The attack against linuxconf exploits a recently discovered vulnerability [linuxconf 2002] in the program's handling of environment variables. When the exploit was run, the intrusion detection system identified 16 anomalous open system calls with suspicious path arguments that caused the string length, the character distribution and the structural inference model to report anomalous occurrences. Another example is the structural inference model alerting on open being invoked with a path that is used directly by the exploit and never occurs during normal program execution.

The attack against apache exploits the KEY_ARG vulnerability in OpenSSL v0.9.6d for Apache/mod_ssl. When the attack is launched, our system detects two anomalous system calls. One of these calls, execve, is reported because Apache does not create a bash process during normal operation.

The third experiment was conducted to obtain a realistic estimate of the number of false alarms that can be expected when the system is deployed on a real-world server. To observe this behavior, we installed the system on our research group's e-mail server, trained the models for a period of two days, and then performed detection on three important daemons (`qmail`, `imapd`, `dhcpd`) for the subsequent five days. Table V shows the number of analyzed system calls as well as the number of false alarms raised during the five days, listed for each of the monitored applications.

Table V. False alarms in real-world environment.

Application	Total System Calls	False Alarms
`dhcpd`	431	0
`imapd`	418,152	4
`qmail`	77,672	11
`Total`	496,255	15

7.2 Bayesian Network Validation

As mentioned in Section 6, the Bayesian networks that were used in the detector to combine individual model scores into a single aggregate score were designed using the domain knowledge of the authors. While the prior section demonstrates the advantage of using decision networks over weighted summations of model scores, a method for validating the network's topology would empirically confirm the essentially intuitive judgments concerning the causal relationships that exist between models of system call arguments. This section proposes such a method for evaluating a chosen Bayesian network topology. A further step toward validating network design would be to evaluate the quality of the chosen CPT values. However, this step is reserved for future work.

Statistical correlation is a necessary but not sufficient condition for a causal relationship between two variables. Thus, if there exists a causal relationship between two models in the proposed Bayesian network, correlation between the variables representing the respective model scores should be observable. Conversely, if no correlation is observed, it may be concluded that there is no causal dependency between the models in question.

The sample correlation coefficient r between two random variables X and Y is defined as [Devore 1982]:

$$r(X,Y) = \frac{n\Sigma as_i^x as_i^y - (\Sigma as_i^x)(\Sigma as_i^y)}{\sqrt{n\Sigma(as_i^x)^2 - (\Sigma as_i^x)^2}\sqrt{n\Sigma(as_i^y)^2 - (\Sigma as_i^y)^2}} \tag{9}$$

where $X = as_1^x, as_2^x, \ldots, as_n^x$ is, for our purposes, the sequence of model outputs (i.e., anomaly scores) $as_1, \ldots, as_n$ produced when $model_X$ is evaluated on a sequence of n system call arguments. Each as_i^x takes on a value in the interval $(0,1)$. The sequence Y is defined for $model_Y$ similarly. The correlation coefficient r is defined to take on a value in the interval $[-1,1]$, with r's value commonly interpreted as in Table VI.

The Bayesian network shown in Figure 3 was used for both the `execve` and `open` system calls. While many causal relationships are present in the graph, we confine our interest to causality between model scores, taken pairwise. The validation

Table VI. Interpretation of correlation coefficient.

$-1.0 \leq r \leq -0.8$	strong negative correlation
$-0.8 \leq r \leq -0.5$	moderate negative correlation
$-0.5 \leq r \leq 0.5$	weak correlation
$0.5 \leq r \leq 0.8$	moderate positive correlation
$0.8 \leq r \leq 1.0$	strong positive correlation

experiment, thus, measures correlation for all six possible pairs of the four models in the network. Model confidences are static throughout the detection phase in this implementation, so computing the correlation coefficient between confidence scores and any other sequence of values is not meaningful. The remaining arcs in Figure 3 are between model scores and the overall classification, which have a well-understood causal relationship.

Our goal in validating the Bayesian network topology is to determine *overall* to what extent causal relationships captured in the Bayesian network are reflected in observed score correlation across a set of evaluation traces. While it is possible to observe score correlation between two models for one set of evaluation data and not for another, observing correlation in multiple system calls in disparate applications across the data set gives strong evidence for the presence of causality between model outputs.

To run this experiment, we again selected Week 1 and 3 from the 1999 Lincoln Lab Evaluation Data for use as a model training set. Weeks 4 and 5 were used for evaluation. Results of the experiment, given in Table VII, show each observed model-model score correlation coefficient for the `execve` and `open` system calls in the `eject`, `fdformat`, and `ps` applications. Results from the `ffbconfig` application are excluded due to the limited number of associated system calls in the data set (21 total system call invocations in the evaluation set).

Table VII. Model ⇔ Model score correlation coefficients for `execve` and `open` system calls.

	eject		**fdformat**		**ps**	
Model Mapping	**execve**	**open**	**execve**	**open**	**execve**	**open**
Token Finder ⇔ String Length	0.0	0.0	0.0	0.0	0.0	0.0
Token Finder ⇔ Char Distribution	0.0	0.0	0.0	0.0	0.0	0.0
Token Finder ⇔ Structural Inf.	0.0	0.0	0.0	0.0	0.0	1.0
String Length ⇔ Structural Inf.	0.0	0.0	0.0	0.0	0.539	0.0
String Length ⇔ Char Distribution	0.0	0.0	0.0	0.524	0.535	0.696
Char Distribution ⇔ Structural Inf.	1.0	0.0	1.0	0.500	0.906	0.0

Table VII shows the correlation coefficients for all six model pairs. The entries that correspond to the model pairs that include the token finder model (i.e., the top half of Table VII) show strong agreement with the network in Figure 3. With one exception, there is no observed correlation between model outputs of the token finder model and the remaining three models. The exception is that a strong correlation was observed between the token finder and structural inference model scores in the `ps` application for the `open` system call. A closer look at the data showed that both models accurately captured the range of normal behavior, as reflected in the training and evaluation data sets. However, when system calls resulting from attacks on `ps` appear, the arguments to the `open` system call consistently took on the values "`/tmp/foo`" and 33188, respectively. Since both values registered as anomalous by the respective models (structural inference and token finder), there was perfect correlation between the two model outputs although no causal relationship was present.

Each of the models in the pairs appearing in the bottom half of Table VII score the same (character string) argument. In both cases – `execve` and `open` – this argument corresponds to the filename that is being executed or opened, respectively. We note that the three model pairs ⟨ string length ⇔ structural inference ⟩, ⟨ string length ⇔ character distribution ⟩, and ⟨ character distribution ⇔ structural inference ⟩ each exhibit at least moderate positive correlation in one or more applications. Furthermore, it can be seen in Figure 3 that the model pair with a direct causal link, ⟨ character distribution ⇔ structural inference ⟩, shows the strongest overall correlation in Table VII. The model pair ⟨ string length ⇔ character distribution ⟩, which is separated by an intermediate variable in the Bayesian network, shows comparatively weaker correlation. Finally, the pair ⟨ string length ⇔ structural inference ⟩, separated by two intermediate variables, shows the weakest correlation. In general, it should be noted that the degree of correspondence between observed model score correlation and the causal links in the Bayesian network suggests that the proposed network topology is reasonable.

7.3 System Efficiency

To quantify the overhead of our intrusion detection system, we have measured its time and space performance characteristics.

The memory space required by each model is practically independent of the size of the training input. Although temporary memory usage during the learning phase can grow proportional to the size of the training data, eventually the models abstract this information and occupy a near constant amount of space. This is reflected in Table VIII, which shows the memory used by our system for two different runs after it had been trained with data from normal executions of `wuftpd` and `linuxconf`, respectively. The results confirm that memory usage is very similar for both test runs, although the size of the input files is different by a factor of 2.5.

Table VIII. Intrusion detection memory usage.

Application	Training Data Size	Memory Usage
`wuftpd`	37,152K	5,842K
`linuxconf`	14,663K	5,264K

To obtain measures that can quantify the impact of our intrusion detection system on a heavily utilized server, we set up a small dedicated network consisting of three PCs (1.4 GHz Pentium IV, 512 MB RAM, Linux 2.4) connected via a 100 Mbps Ethernet. One server machine hosted the intrusion detection system and `wuftpd`. The two dedicated client machines each ran $k/2$ simultaneous instances of FTP client scripts. For each k, the time to complete a series of downloads was measured. For our purposes, time to completion was measured as the point in time when the first client on either machine began its download to the point in time when the last client on either machine finished. Each instance of the client script connected to the `wuftpd` daemon on the server anonymously and downloaded five 100-kByte files, two 512-kByte files, one 1-Mbyte file, one 30-Mbyte file, and one 50-Mbyte file. The time to completion was measured for $k = 2$ simultaneous clients up to $k = 20$ clients in steps of 2.

This experiment was run three times: once without any auditing, once with system call auditing (i.e., Snare), and finally once with system call auditing (i.e., Snare) and our intrusion detection system. Figure 5 summarizes the results of this experiment, showing the average time to completion across ten trials (variances were less than 5% for all results).

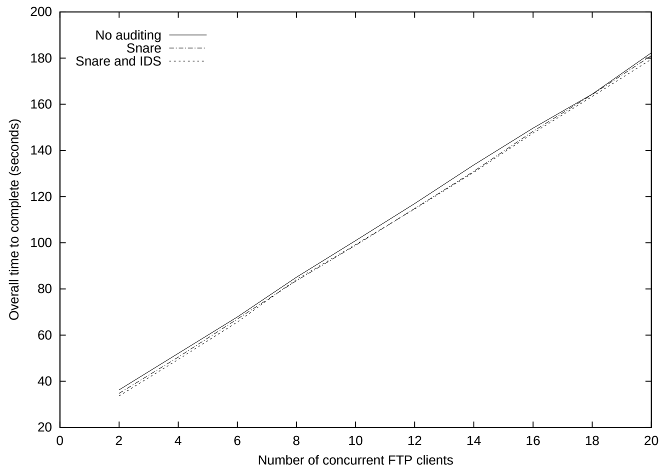

Fig. 5. FTP-client response times.

From this figure, it can be seen that the server performance experienced by each client is virtually indistinguishable for all three cases. This indicates that the number of system calls that have to be analyzed every second by the intrusion detection system is too low to be noticeable as performance degradation. Further analysis showed that the bottleneck in this experiment was the network. For all numbers of clients k, the 100 Mbps network was determined to be completely utilized. This explains the linear increase of the time to completion. The number of monitored system calls that `wuftpd` issued per second was 210 on average.

To increase the system call rate to a point that would actually stress the system, we developed a synthetic benchmark that can execute a variable number of system calls per second at a rate that far exceeds the rate of system calls normally invoked by server applications. By measuring the resulting CPU load for different rates of system calls, we obtain a quantitative picture of the impact of the IDS and its ability to operate under very high loads.

We ran the benchmark tool on an otherwise idle system for varying system call rates three times: once without any auditing, once with system call auditing (i.e., Snare), and finally once with both system call auditing (i.e., Snare) and our intrusion detection system. Figure 6 shows the resulting CPU load observed on the system as an average of 10 runs.

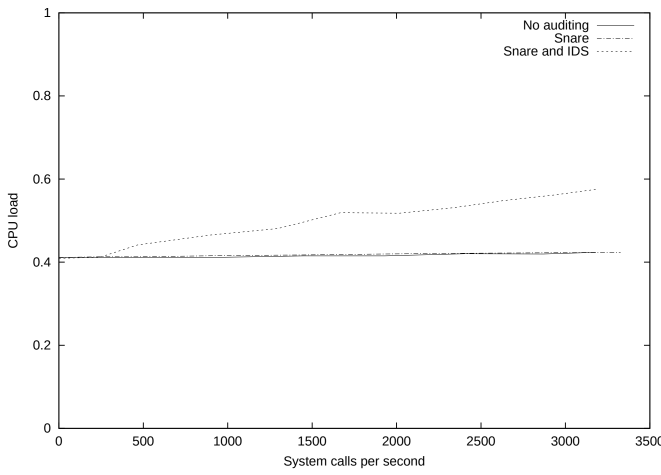

Fig. 6. CPU load for different system call rates.

The benchmark application used approximately 40% of the CPU on an idle system without auditing. As the number of system calls per second increased, a negligible impact on the CPU was observed, both with auditing turned completely off and with auditing in place. When our intrusion detection system was enabled, the CPU load increased up to 58%, when the benchmark performed about 3000 system calls per second. Note that this rise was caused by a nearly fifteen-fold increase of the number of system calls per second compared to the number that needed to be analyzed when `wuftp` was serving clients on a saturated fast Ethernet.

8. CONCLUSIONS

For a long time, system calls and their arguments have been known to provide extensive and high-quality audit data, which has been used by security applications to perform signature-based intrusion detection or policy-based access control enforcement. However, learning-based anomaly intrusion detection has traditionally focused only on the sequence of system call invocations. System call arguments have been neglected because their analysis has been considered either too difficult or too expensive computationally.

This work has demonstrated that argument models are a powerful method of detecting attacks with a low rate of false positives. Our method of combining multiple anomaly scores using a Bayesian modeling approach also showed significant improvement over traditional score aggregation approaches. Additionally, we performed a direct comparison of our approach to four other learning-based approaches on a well-known intrusion detection evaluation data set. This comparison showed that our system appreciably outperforms the detection capability of these systems.

Finally, we have shown that it is possible to analyze system call arguments with extremely low computational and memory overheads.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CCR-0238492 and CCR-0524853. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, or the U.S. Government.

REFERENCES

advisory-ftpd 2000. Advisory: Input validation problems in wuftpd. `http://www.cert.org/advisories/CA-2000-13.html`.

Axelsson, S., Lindqvist, U., Gustafson, U., and Jonsson, E. 1998. An approach to UNIX security logging. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*.

Bernaschi, M., Gabrielli, E., and Mancini, L. V. 2002. REMUS: a Security-Enhanced Operating System. *ACM Transactions on Information and System Security 5,* 36 (February).

Billingsley, P. 1995. *Probability and Measure*, 3 ed. Wiley-Interscience.

Bykova, M., Ostermann, S., and Tjaden, B. 2001. Detecting network intrusions via a statistical analysis of network packet characteristics. In *Proceedings of the 33rd Southeastern Symposium on System Theory.*

Chari, S. N. and Cheng, P.-C. 2002. Bluebox: A policy-driven, host-based intrusion detection system. In *Proceedings of the 2002 ISOC Symposium on Network and Distributed System Security (NDSS'02).* San Diego, CA.

Denning, D. 1987. An Intrusion Detection Model. *IEEE Transactions on Software Engineering 13,* 2 (Feb.), 222–232.

Devore, J. 1982. *Probability and Statistics for Engineering and the Sciences*, 1 ed. Brooks/Cole.

Feng, H., Kolesnikov, O., Fogla, P., Lee, W., and Gong, W. 2003. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy.*

Forrest, S. 1996. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy.* Oakland, CA, 120–128.

Ghosh, A., Wanken, J., and Charron, F. 1998. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the Annual Computer Security Application Conference (ACSAC'98).* Scottsdale, AZ, 259–267.

Giffin, J., Jha, S., and Miller, B. 2004. Efficient context-sensitive intrusion detection. In *proceedings of 11th Network an Distributed System Security Symposium.* San Diego, California.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. 1996. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium.* San Jose, CA, USA.

Javitz, H. S. and Valdes, A. 1991. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Symposium on Security and Privacy.*

Jensen, F. V. 2001. *Bayesian Networks and Decision Graphs.* Springer-Verlag.

Kang, D.-K., Fuller, D., and Honavar, V. 2005. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *Proceedings of 6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW).*

Ko, C., Ruschitzka, M., and Levitt, K. 1997. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy.* 175–187.

Lee, S. Y., Low, W. L., and Wong, P. Y. 2002. Learning Fingerprints for a Database Intrusion Detection System. In *7th European Symposium on Research in Computer Security (ESORICS)*.

Lee, W., Stolfo, S., and Chan, P. 1997. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *Proceedings of the AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*.

Lee, W., Stolfo, S., and Mok, K. 1999. Mining in a Data-flow Environment: Experience in Network Intrusion Detection. In *Proceedings of the* 5^{th} *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '99)*. San Diego, CA.

Lindqvist, U. and Porras, P. 1999. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*. Oakland, California, 146–161.

linuxconf 2002. Advisory: Buffer overflow in linuxconf. `http://www.idefense.com/advisory/08.28.02.txt`.

Lippmann, R., Haines, J. W., Fried, D. J., Korba, J., and Das, K. 2000. Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings of Recent Advances in Intrusion Detection*. LNCS. Springer, Toulouse, France, 162–182.

Neumann, P. and Porras, P. 1999. Experience with EMERALD to Date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring* (Santa Clara).

Paxson, V. 1998. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*. San Antonio, TX.

Porras, P. and Neumann, P. 1997. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Systems Security Conference*.

Portnoy, L., Eskin, E., and Stolfo, S. 2001. Intrusion Detection with Unlabeled Data using Clustering. In *ACM CSS Workshop on Data Mining Applied to Security (DMSA)*.

Provos, N. 2003. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*. Washington, DC.

Snare 2003. SNARE - System iNtrusion Analysis and Reporting Environment. `http://www.intersectalliance.com/projects/Snare`.

Staniford, S., Hoagland, J., and , McAlerney, J. 2000. Practical automated detection of stealthy portscans. In *Proceedings of the IDS Workshop of the 7th Computer and Communications Security Conference*. Athens.

Stolcke, A. and Omohundro, S. 1993. Hidden Markov Model Induction by Bayesian Model Merging. *Advances in Neural Information Processing Systems*.

Stolcke, A. and Omohundro, S. 1994. Inducing probabilistic grammars by bayesian model merging. In *Conference on Grammatical Inference*.

Tan, K., Killourhy, K., and Maxion, R. 2002. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *Proceedings of RAID 2002*. Zurich, CH.

Tan, K. and Maxion, R. 2002. "Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, 188–202.

Vigna, G., Valeur, F., and Kemmerer, R. 2003. Designing and Implementing a Family of Intrusion Detection Systems. In *Proceedings of the* 9^{th} *European Software Engineering Conference*. Helsinki, Finland.

Wagner, D. and Dean, D. 2001. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Press, Oakland, CA.

Wagner, D. and Soto, P. 2002. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*. Washington DC, USA, 255–264.

Warrender, C., Forrest, S., and Pearlmutter, B. 1999. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*.