Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions

R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang and S. Zhou
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794.

ABSTRACT

Unlike signature or misuse based intrusion detection techniques, anomaly detection is capable of detecting novel attacks. However, the use of anomaly detection in practice is hampered by a high rate of false alarms. Specification-based techniques have been shown to produce a low rate of false alarms, but are not as effective as anomaly detection in detecting novel attacks, especially when it comes to network probing and denial-of-service attacks. This paper presents a new approach that combines specification-based and anomaly-based intrusion detection, mitigating the weaknesses of the two approaches while magnifying their strengths. Our approach begins with state-machine specifications of network protocols, and augments these state machines with information about statistics that need to be maintained to detect anomalies. We present a specification language in which all of this information can be captured in a succinct manner. We demonstrate the effectiveness of the approach on the 1999 Lincoln Labs intrusion detection evaluation data, where we are able to detect all of the probing and denial-of-service attacks with a low rate of false alarms (less than 10 per day). Whereas feature selection was a crucial step that required a great deal of expertise and insight in the case of previous anomaly detection approaches, we show that the use of protocol specifications in our approach simplifies this problem. Moreover, the machine learning component of our approach is robust enough to operate without human supervision, and fast enough that no sampling techniques need to be employed. As further evidence of effectiveness, we present results of applying our approach to detect stealthy email viruses in an intranet environment.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access; C.2.3 [**Network Operations**]: Network monitoring

General Terms

Security, Management

Keywords

Intrusion detection, anomaly detection, network monitoring

1. INTRODUCTION

Intrusion detection approaches can be divided into *misuse detection*, *anomaly detection* and *specification-based detection*. Misuse detection techniques detect attacks as instances of *attack signatures*. This approach can detect known attacks accurately, but is ineffective against previously unseen attacks, as no signatures are available for such attacks.

Anomaly detection overcomes the limitation of misuse detection by focusing on *normal system behaviors*, rather than attack behaviors. This approach is characterized by two phases: in the *training phase*, the behavior of the system is observed in the absence of attacks, and machine learning techniques used to create a *profile* of such normal behavior. In the *detection phase*, this profile is compared against the current behavior of the system, and any deviations are flagged as potential attacks. Unfortunately, systems often exhibit legitimate but previously unseen behavior, which leads anomaly detection techniques to produce a high degree of false alarms. Moreover, the effectiveness of anomaly detection is affected greatly by what aspects (also called "features") of the system behavior are learnt. The problem of selecting an appropriate set of features has proved to be a hard problem.

Specification-based techniques are similar to anomaly detection in that they also detect attacks as deviations from a norm. However, instead of relying on machine learning techniques, specification-based approaches are based on manually developed specifications that capture legitimate (rather than previously seen) system behaviors. They avoid the high rate of false alarms caused by legitimate-but-unseen-behavior in the anomaly detection approach. Their downside, however, is that development of detailed specifications can be time-consuming. Thus, one has to trade off specification development effort for increased false negatives (i.e., likelihood that some attacks may be missed).

Given the complementary nature of the strengths and weaknesses of anomaly and specification-based approaches, a natural question is whether the two approaches can be combined in such a way that we can realize the combination of their strengths, while avoiding the weaknesses of either one. We answer this question affirmatively in this paper by developing a new and effective network intrusion detection technique that combines these two approaches. We demonstrate the effectiveness of our approach using experiments involving the 1999 Lincoln Labs Intrusion detection data [7]. As further evidence, we also summarize the results of using our approach in a very different context, namely, detection of email flooding attacks due to viruses.

1.1 Overview of Approach

The first step in our approach is to develop specifications of hosts and routers in terms of network packets received or transmitted by

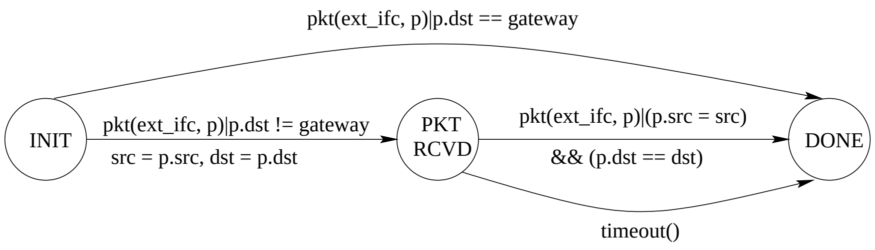
pkt(ext_ifc, p)|p.dst == gateway
INIT
pkt(ext_ifc, p)|p.dst != gateway
src = p.src, dst = p.dst
PKT RCVD
pkt(ext_ifc, p)|(p.src = src)
&& (p.dst == dst)
DONE
timeout()

Figure 1: Simplified IP Protocol State Machine

them. These specifications are derived from RFCs or other descriptions of protocols such as the IP, ARP, TCP and UDP. Consider a gateway node that connects an organization's local network to the Internet. Figure 1 is a pictorial representation of a specification characterizing the gateway's behavior, as observed at the IP protocol layer. The figure incorporates the following simplifications: no IP fragmentation is modeled, and only packets *from* the Internet (but not those sent to the Internet) are captured. These packets may be destined for the gateway itself, in which case the state machine makes a transition from the `INIT` to `DONE` state. Otherwise, a packet may be destined for an internal machine, in which case the gateway will first receive it on its external network interface, and make a transition from the `INIT` to `PKT_RCVD` state. Next, it will relay the packet on its internal network interface, making a transition to the `DONE` state. Occasionally, the relay may not take place. This may be due to a variety of reasons, including (a) the gateway could not resolve the MAC address corresponding to the IP address of the target machine, (b) the gateway machine is malfunctioning, etc. We model such situations with a timeout transition from the `PKT_RCVD` state to the `DONE` state.

As shown in Figure 1, specifications are based on *extended finite state automata (EFSA).* An EFSA is similar to a finite-state automaton, with the following differences: (a) an EFSA makes transitions on events that may have arguments, and (b) it can use a finite set of *state variables* in which values can be stored. In the figure, we see two events, namely, `pkt` and `timeout`. The former event denotes the reception or transmission of a packet. Its first argument identifies the network interface on which the packet was received or transmitted. Its second argument captures the packet contents. The `timeout` denotes a time out transition, which will be taken if no other transitions are taken out of a state for a predefined period of time. The IP state machine uses two state variables `src` and `dst`. These variables are used to store the source and destination IP addresses seen in a packet arriving on the gateway's external interface. By using these state variables, the state machine is able to match a packet received on the external interface with the corresponding packet (when it is relayed) on the internal interface.

To understand how such EFSA specifications can be used for monitoring protocol behavior, consider the IP state machine again. For each IP packet received on the external network interface, we create an instance of the IP state machine that is in the `INIT` state, and add this instance to the list of active IP machine instances. Next, the packet is given to every IP state machine instance in this list, and each of them that can make a transition on this packet is permitted to do so. Any state machine that reaches the `DONE` state is deleted from the list. Thus, when monitoring protocol behavior, we create many instances of the state machine, each of which traces a path in the EFSA from the `INIT` to the `DONE` state. (A trace is characterized by a sequence of states, where "state" includes not only the control state of the automata (e.g., `DONE`, `PKT_RCVD` and `DONE`) but also the values of state variables.)

Now, we superimpose statistical machine learning over this specification as follows. Note that the EFSA specifications map the statistical properties of the IP packet stream to properties of traces accepted by the EFSA specifications. Thus, we can characterize the statistical properties of the IP packet stream in terms of:

- the frequency (across traces) with which a particular transition in the EFSA is taken, e.g., the frequency with which the timeout transition is taken
- the most commonly encountered value of a state variable at a particular control state of the EFSA, e.g., the most common value for the `dst` state variable at the `PKT_RCVD` state. (This value would correspond to the IP address to which maximum number of IP packets are received from the Internet.)
- the distribution of values of a state variable, e.g., how frequently does the protocol field (in the IP header) have the value `TCP`, `UDP`, etc.

In addition, we may be interested in statistical properties across a *subset* of traces, rather than all traces. The traces of interest can be specified on the basis of state variable values. For instance, we may be interested in the number of IP-packets being relayed by the gateway to a particular local machine M. We will do this by selecting traces that have `dst` equal to M in their `PKT_RCVD` state, and identifying the number of times the transition from `PKT_RCVD` to `DONE` was taken in these traces[1]. A second, orthogonal way to select a subset of traces is based on time: we may be interested in traces that were observed within the last T seconds.

Based on learning statistical properties associated with the IP-state machine, we could detect several kinds of attacks. We describe the detection of IP sweep attack in particular, as the detection mechanism is quite interesting. Typically, detection of IPsweep attack requires an IDS to incorporate knowledge about IPsweeps at some level. Often, a particular statistic is designed that specifically targets IPsweep, e.g., the number of different IP addresses for which packets were received in the last t seconds for some suitably small value of t. Once this is done, there is no surprise that the attacks can be detected fairly accurately, based on anomalies in this statistic. In contrast, we do not encode any knowledge about IPsweeps in our approach. Nevertheless, we are able to detect them as follows. Since an IPsweep attack is designed to identify the IP addresses in use on a target network, the attacker does not know legitimate IP addresses in the target domain at the time of attack. This implies that several packets will be sent by the attacker to nonexistent hosts. This would result in a sudden spurt of `timeout` transitions being taken in the IP state machine. Thus, the statistics on the frequency of timeout transitions from the `PKT_RCVD` state can serve as a reliable indicator of the IPsweep attack.

[1]More powerful primitives for trace selection are possible, but not necessary — a complex selection condition can be directly incorporated into the EFSA as follows: introduce a new state variable that records the outcome of testing this condition. Now, the complex selection criteria reduces to that of selecting those traces where this new state variable has the value $true$.

1.2 Benefits of Approach

Our approach:

- *provides accurate attack detection.* Our experimental results illustrate that our approach provides:
  - *excellent detection of known and unknown attacks.* Although there have been questions about the realism with the 1999 Lincoln Labs evaluation data, it is nevertheless remarkable that our approach can identify *all* of the attacks that were within the scope of our system. The detected attacks include very stealthy attacks, e.g., port sweeps that involve 3 packets from two different hosts. As further evidence of effectiveness, we provide preliminary results in a very different context, namely, for detecting anomalies caused by email viruses.
  - *low false alarm rates.* Our system generated, on the average, 5.5 false alarms per day. This is at the low end of the false alarm rates reported in the 1999 evaluation, even when misuse detection based approaches (which traditionally have had much lower false alarm rates compared to anomaly detection) are taken into account.
- *simplifies feature selection.* One of the difficulties in anomaly detection is the choice of parameters that should be learnt. With network packet data, there is a large number of parameters, with many parameters assuming values from very large sets. Moreover, attack detection often requires one to consider sequences of packets. Note that the number of possible parameter combinations across packet sequences increases rapidly with sequence length — for instance, if a single packet has 10 parameters of interest, a sequence of 3 packets has a total of 10^3 possible parameters. Selecting a small set of parameters from such an extremely large space of parameters is a challenging problem. In our approach, properties of sequences are mapped into properties associated with individual transitions in the state machines. This enables us to detect most attacks by simply monitoring the distribution of frequencies with which each transition is taken.
- *employs redundancy to improve attack detection.* Our approach tends to learn very detailed information about many different characteristics of network protocols. Although most attacks can be detected by looking at a fraction of these characteristics, the redundant characteristics benefit in two ways:
  - An attack would likely change at least a subset of the large set of characteristics being monitored. Thus, the redundancy provides a "safety cushion" against making a poor choice of characteristics to monitor, or inadequacies in the learning algorithms.
  - It becomes much harder to craft evasive attacks, wherein an attacker attempts to carry out an attack without perturbing the parameters and features being monitored. Clearly, it is much harder to craft attacks that preserve many different characteristics and features of the system, as opposed to just a few.
- *supports unsupervised learning.* Our approach is robust enough to accommodate unsupervised learning, i.e., the information learnt at the end of the training phase does not need to be inspected or modified by a human before it is used for detection[2].

In summary, our approach enables seamless combination of anomaly detection and specification-based detection. The combination provides significantly more value than the "sum of its parts," as many attacks undetectable by either of those approaches become detectable using our approach. At the same time, the false alarm rate is contained at a low level. Preliminary performance measurements (which are preliminary in the sense that no systematic attempt to improve performance has been undertaken so far) indicates that our implementation provides adequate performance, processing an entire day's data (about 0.7GB) in under ten minutes.

1.3 Organization of the Paper

In Section 2, we present a summary of our specification language. The language is designed to enable concise specifications of protocols. We illustrate the language with a complete specification of the IP state machine shown in Figure 1. Further discussion of specification development process appears in Section 3, together with a discussion of our TCP state machine specification. Section 4 describes how anomaly detection is mapped onto these specifications. Section 5 describes our experimental results with the 1999 Lincoln Labs data. We used state machine models of IP and TCP protocols in this experiment. Section 5.2 provides a short description of a second experiment where our approach was used to detect email viruses. Comparison with related work appears in Section 6. Finally, concluding remarks appear in Section 7.

2. STATE-MACHINE LANGUAGE

As mentioned earlier, network protocols are modeled using *extended finite state automata (EFSA)*, which augment traditional FSA with a set of state variables. Formally, an EFSA M is a septuple $(\Sigma, Q, s, f, V, \mathcal{D}, \delta)$, where:

- Σ is the *alphabet* of the EFSA. It is an *event alphabet,* i.e., elements of Σ are characterized by an event name as well as event arguments.
- Q is a finite set of states (also called as *control states*) of the EFSA
- $s \in Q$ is the *start state* of the EFSA
- $f \in Q$ is the *final state*. In our models, f is a *sink state,* i.e., a state that has no outward transitions.
- V is a finite tuple $(v_1, ..., v_n)$ of *state variables*
- $\mathcal{D}$ is a finite tuple $(D_1, ..., D_n)$, where D_i denotes the domain of values for the variable v_i.
- $\delta : Q \times \mathcal{D} \times \Sigma \to (Q, \mathcal{D})$ is the transition relation.

Below, we describe our language for specifying EFSA that model network protocols.

2.1 State Machine Specification

State machines specifications follow the EFSA definition given above. The set Σ (events) are specified as part of an *interface declaration,* which lists the events and their argument types. (Interface declarations are omitted in this paper to conserve space.) Instead of $\mathcal{D}$, the specifications define the set V, as well as the types of each member of V. Specifically, the following declarations specify Q, s, f and V.

- The (control) states of a state machine may be declared using `states` $\{s_1, ..., s_n\}$.
- The start state of the state machine can be specified using the declaration `startstate` s.
- The final state of the state machine can be specified using the declaration `finalstate` f.

[2]Given the volume of information learnt by our approach, manual inspection would not even be practical.

```
event tx(int interfaceId, ether_hdr data);
event rx(int interfaceId, ether_hdr data);

StateMachine ip_in(int in, int ext,
                   IPaddr in_ip, IPaddr ext_ip) {
   /* in and ext refer to internal and external
      interfaces. The corresponding IP addresses
      are in_ip and ext_ip   */

   states {INIT, PKT_RCVD, DONE};
   startstate INIT;
   finalstate DONE;

   IPaddr src, dst; /* state variables */

   timeout 60 seconds {PKT_RCVD};

   map rx(ifc, pkt) when (ifc == ext);
   map tx(ifc, pkt) when (ifc == in) &&
     (pkt.ipsrc == src) && (pkt.ipdst == dst);

   rx(ifc, pkt)|(ifc == ext) && (state == INIT)
      && (pkt.ipdst != in_ip) && (pkt.ipdst != ext_ip)
   --> state=PKT_RCVD; src=pkt.ipsrc; dst=pkt.ipdst;
   rx(ifc, pkt)|(ifc == ext) && (state == INIT)
       && ((pkt.ipdst==in_ip) || (pkt.ipdst==ext_ip))
   --> state = DONE;
   tx(ifc, pkt)|(ifc == in) && (state == PKT_RCVD)
   --> state = DONE;
   timeout|(state == PKT_RCVD) --> state = DONE;
}
```

Figure 2: IP machine specification.

- The variables in V are declared using syntax similar to variable declarations in typical programming languages.

The transition relation δ is specified using rules of the form:

$$e(x_1, ..., x_n)|cond \rightarrow action$$

Here e is an event name, and the variables $x_1, ..., x_n$ denote the arguments of this event. The expression $cond$ should evaluate to a boolean value, and can make use of common arithmetic and relational operators. It involves the variables in V, the event arguments, and the distinguished variable $state$ that refers to the current control state of the EFSA. The $action$ component consists of actions that will be taken when the event e occurs, and $cond$ evaluates to true. Allowable actions include assignments to state variables (i.e., variables in V) and invocations of external functions. The action must also include an assignment to $state$.

In general, protocol state machines are non-deterministic. We simulate non-determinism by cloning k copies of the state machine whenever it can make one of k different transitions. (The cloning operation duplicates not only the control state, but also all of the variables in V.) Clearly, we cannot have a situation where the number of state machine instances increases forever. To deal with this problem, we automatically delete state machine instances that reach their final state. Note that final states are some what different from "accepting states" of an FSA – they are similar to "sink" states from which no progress can be made.

In general, there can be many instances of a state machine at runtime. Thus, for each incoming event, we may have to search through all of these state machine instances to discover those that can make a transition. This operation can be very expensive, so we use a mechanism that speeds up this operation in a situation that arises frequently: often, we use one state machine instance to track a "session," and the session to which an event applies can be computed efficiently from the event parameters. The following language construct is used to specify such mapping:

$$\texttt{map}\ event(eventArgs)\ \texttt{when}\ condition$$

Here $event$ can be a primitive or an abstract event that is defined without the use of any conditions. The $condition$ component must be of a special form: it should be a conjunction of equality tests, where the left-hand side of the test is an expression on $eventArgs$ and the right-hand side is a state variable. This restriction is imposed so that the identification of the right state machine instance can be implemented using a hash-table lookup.

Our language also permits timeout transitions to be described. Timeouts values can be declared using one or more declarations of the form:

$$\texttt{timeout}\ t\ \texttt{in}\ \{s_1, ..., s_m\}$$

This declaration states that a state machine will stay in one of the states $s_1, ..., s_m$ for at most t seconds. At the end of this period, a transition associated with the special event `timeout` will be taken.

Figure 2 shows the complete specification of the IP state machine shown in Figure 1.

3. SPECIFICATION DEVELOPMENT

Unlike software in general, network protocols are designed through a careful and deliberate process. The design is captured in a precise fashion in standards documents. Such documents provide an obvious starting point for our state machine specifications.

In our work, we have tended to abstract from this specification to capture only the essential details of most protocols. Such information may be readily obtained from standard texts on network protocols rather than (the much longer) Internet RFCs. While strict adherence to protocol standards documents is possible, this may not be desirable for two reasons. First, developing precise specifications would entail more effort than that required for more abstract specifications. Second, with strict specifications, there is always the possibility that due to minor difference in interpretation, some traffic may be classified as invalid by the state machine, and hence not processed properly. Furthermore, such incorrect processing may happen with some TCP implementations and not others. Using a more abstract specification, where the state machines accept a superset of what is permitted by the standards, provides a satisfactory solution to these problem.

We conclude this section with a specification of the TCP state machine, as observed on a gateway connecting an organization's internal network to the Internet. Our specification is depicted pictorially in Figure 3. A new session starts in the *LISTEN* state. Data transfer takes place in the connection *ESTABLISHED* state. If the TCP connection is initiated from an external site, then the state machine goes through *SYN_RECD* and *ACK_WAIT* states to reach the *ESTABLISHED* state. If the connection is initiated from an internal machine, then the *ESTABLISHED* state is reached through the *SYN_SENT* state.

In order to tear down the connection, either side can send a TCP segment with the FIN bit set. If the FIN packet is sent by an internal host, the state machine waits for an ACK of FIN to come in from the outside. Data may continue to be received till this ACK to the FIN is received. It is also possible that the external site may initiate a closing of the TCP connection. In this case we may receive a FIN, or a FIN + ACK from the external site. This scenario is represented by the states *FIN_WAIT_1*, *FIN_WAIT_2*, *CLOSING*, *CLOSING_1* and *CLOSING_2* states. Our state machine characterizes receive and transmit events separately, and this necessitates additional intermediate states that are not identified in the TCP RFCs.

If the connection termination is initiated by an external host, note that the TCP RFCs do not have the states CLOSE_WAIT_1,

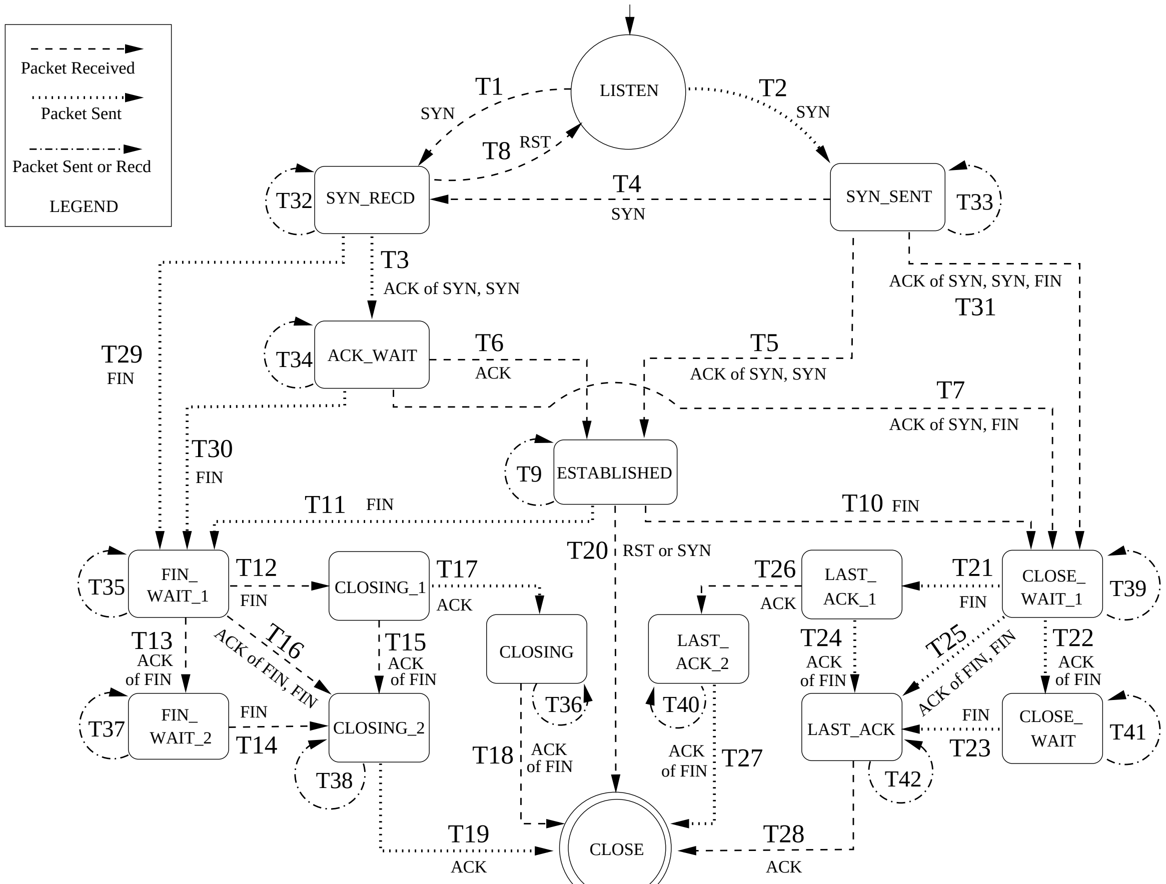

Figure 3: TCP Protocol State Machine

CLOSE_WAIT_2, LAST_ACK_1, and LAST_ACK_2 since they deal with packets observed at one of the ends of the connection. In that case, it is reasonable to assume that no packets will be sent by a TCP stack implementation after it receives a FIN from the other end. In our case, we are observing traffic at an intermediate node (gateway), so the tear down process is similar regardless of which end initiated the tear down.

To reduce clutter, the following classes of abnormal transitions are not shown: (a) abnormal conditions under which a TCP connection may be terminated, including when an RST packet (with correct sequence number) is sent by either end of the connection, as well as timeouts, (b) conditions where an abnormal packet is discarded without a state transition, e.g., packets received without correct sequence numbers (after connection establishment) and packets with incorrect flag settings.

4. ANOMALY DETECTION

Information sources such as network packets pose a significant challenge for anomaly detection techniques for two reasons. First, the volume of data, and consequently, the space of possible statistical properties of interest, is extremely large. Second, raw network packet data tends to be unstructured, making it difficult to distinguish meaningful information from "background noise." To deal with this problem, the raw packet data is usually processed to extract important "features" that are deemed to be of interest. This process greatly reduces the amount of data to be processed by an anomaly detection system. Moreover, it identifies important information from the packet streams, while discarding less useful information.

The importance of good feature selection is acknowledged by most researchers in anomaly detection. Currently, feature selection is driven by human expert's knowledge and judgment regarding what constitutes "useful information" for detecting attacks. While human experts are often in a position to identify some useful features, it is far from clear that they can do a comprehensive job. Often, their notion of a useful feature is influenced by their knowledge of known attacks. Consequently, they may not necessarily select features that are useful in detecting unknown attacks.

In our approach, a higher degree of automation is brought to the process of feature selection. Specifically, (statistical) properties of packet sequences are mapped into (statistical) properties associated with the transitions of the state machine. Since the number of transitions is relatively small as compared to the number of possible combinations of network packets, this mapping reduces the space of possible features. At the same time, our experiments provide evidence that this reduction does not decrease detection efficacy.

4.1 Mapping packet sequence properties to properties of state-machine transitions

As mentioned earlier, specifications divide up packet sequences into *traces*, where each trace corresponds to a path in the state machine. For instance, the IP state machine described above partitions the sequence of packets received at the external interface of the gateway or transmitted at the internal interface into one of the following kinds of traces:

- `rx(ext, pkt)` where `pkt` is destined for the gateway
- `rx(ext, pkt1) tx(int, pkt2)` where `pkt2` is a packet that is relayed by the gateway in response to receiving `pkt1`

- `rx(ext, pkt1) timeout`, where a packet is received by the gateway with destination address other than that of the gateway, but the packet is not relayed by the gateway (usually due to a packet error, such as invalid IP address, or a gateway error/failure)

This partitioning brings a lot of structure into what would otherwise be a long, unstructured sequence of packets. It also reduces the space of possible properties of interest, since a trace, being fairly short, has much fewer properties than that of extremely long packet sequences. In addition to reducing the space of possible properties, the transitions in the state machine specifications provide concrete clues on what properties may be of interest. For instance, some transitions represent unexpected packets, which usually occur due to network failures or an attack. Similarly, absence of expected packets, and the consequent transition on a timeout event, suggests a failure or an attack. For this reason, our approach is focused on properties related to individual transitions. We identify two categories of properties:

- **Type 1:** whether a particular transition on the state machine is taken by a trace. (Example: is the timeout transition taken by a trace?)
- **Type 2:** the value of a particular state variable or a packet field when a transition is traversed by a trace. (Example: what is the size of IP packet when the transition from `INIT` to `PKT_RCVD` state is taken?)

More complex properties that involve multiple transitions, e.g., whether a trace traverses a particular combination of transitions, can also be captured in our approach. This would be accomplished by augmenting the original specification with an auxiliary state variable that would be set if a trace traversed the desired combination of transitions. Now the original property reduces to a simple state transition property – specifically, that of taking the last of the transitions under the condition that the auxiliary state variable is set. In our experience to date, however, we have not found it necessary to introduce such auxiliary variables.

4.2 Learning statistical properties

Anomaly detection is concerned with detecting "unusual behaviors." With our state machine models, we are ultimately mapping behaviors to transitions of the state machine. Thus, unusual behaviors can be detected if our approach learns how frequently a transition is taken (for type 1 properties), or the commonly encountered values of state variables on a transition (for type 2 properties). One obvious way to represent this information is as an average, e.g., the average frequency with which a transition is taken. However, it is well-known that network phenomena tend to be highly bursty, and hence averages do not provide an adequate way to characterize such phenomena. Therefore, in our approach, we focus on capturing *distributions* rather than averages. For type 1 properties, we maintain frequency distributions, whereas for type 2 property, we maintain the distribution of values for the state variable of interest.

The representation of distributions differs, depending on the nature of the values in the distribution. If the values are *categorical* (e.g., an IP address), then a distribution simply counts the number of times each distinct value occurs in the distribution. Often, the number of possible categories may be too large, so the distribution may represent only those categories that occur most frequently. If the values represent a scalar quantity such as a packet size, then the distribution can be represented compactly using a histogram. Since frequencies represent a scalar quantity, frequency distributions can also be represented using histograms.

Often, we are interested in properties that hold across a subset of traces. One way to select traces of interest is based on recency, e.g., traces witnessed during the last t seconds. This would enable us to focus on recent behavior, as opposed to behaviors observed a long time in the past. A second way to select traces is based on values of state variables or packet fields. For instance, we may be interested in:

- traces corresponding to fragmented packets
- traces involving packets from a particular host and/or to a particular host

Statistical properties to be learnt can be specified conveniently in our specification language as follows. To illustrate such specifications, consider the statement:

```
on all frequency
    timescale (0.001, 0.02, 0.5, 10, 100, 1000)
```

This statement indicates that `frequency` distribution information should be learnt on `all` transitions, and that six different distributions should be maintained, corresponding to six different `timescales`. A timescale specifies the period over which we count the number of times a transition is taken. Use of short time scales enables faster attack detection. However, since network phenomena tend to be more bursty at shorter time scales, slow attacks tend to be missed at shorter time scales. They can be detected by observing statistics over larger time scales, but those time scales imply longer latencies before attack detection. By using six time scales that range from a millisecond to a thousand seconds, we combine the benefits of fast detection of rapidly progressing attacks, with delayed (but more certain) detection of slower attacks.

As a second example, consider the following statement:

```
on all frequency wrt (src) size 100
        timescale (0.001, 0.02, 0.5, 10, 100, 1000)
```

This statement indicates that we wish to maintain frequency distribution on a per-source-host basis. Since the number of possible source hosts can be large, the language allows the use of `size` declarations to bound the storage requirements. For instance, a bound of 100 is declared in the above statement. If more than this many source hosts are active at any time, then only the most active 100 of these hosts will be retained in the table, and the others would be purged. Our notion of "most active" incorporates aging, so that hosts that were active in the past but have become inactive for a long period since, will be discarded from the table. (Recall that src is the name of a state variable in the IP state machine specification.)

The keyword `value` is used in place of `frequency` to indicate value distributions, as opposed to frequency distributions. This allows us to monitor specific ranges of values a state variable can take. In a similar way, we can restrict our monitoring to a specific subset of transitions by listing their labels, instead of the key word `all`.

4.3 Detecting Anomalies

During the detection phase, the statistics specified for learning are computed again, and compared with the values learnt during the training phase. If the statistics vary substantially from what was learnt, then an anomaly is raised.

We are currently investigating ways to precisely control what is considered "substantial difference." Meanwhile, our implementation uses a simple thresholding scheme that applies to distribution data maintained as histograms (i.e., frequency distributions and distribution of values of scalar parameters). For a parameter p, let t_p denote the highest histogram bin with nonzero count during training, and d_p denotes the corresponding number during detection phase, then an anomaly will be flagged if $t_d - t_p$ exceeds a threshold. While the threshold could be explicitly specified, to simplify things further as follows. First, we use geometric ranges for histogram bins, e.g., successive bins may correspond to values $[0-1]$, $[1-2]$, $[2-4]$, $[4-8]$..... Then we set the threshold to a fixed value such as 1 or 2. Our experiments use a threshold of 1.

5. EXPERIMENTAL RESULTS

5.1 Experiments with 1999 Lincoln Labs Evaluation Data

We studied the effectiveness of our approach by testing our approach using 1999 DARPA/Lincoln Labs evaluation [7]. The evaluation organizers set up a dedicated network to conduct a variety of attacks. Care was taken to ensure that the distribution of traffic in terms of different protocols and services was similar to that seen at a large organization. All of the network traffic was recorded in tcpdump format and provided to the participants of the evaluation. The data provided consisted of training data, plus two weeks of test data. The uncompressed size of the tcpdump files was approximately 700MB per day.

Our experiments have focused on attacks on lower layers of protocols such as IP and TCP, due to the fact that we have so far developed state machine models of only these two protocols. Such attacks correspond to the probing and denial-of-service (DoS) attacks in the Lincoln Labs data, with one exception: Since our approach recognizes anomalies based on repetition, at least two packets must be involved in an attack before the attack can be expected to be detected by our approach[3]. This eliminates the following attacks from consideration: arppoison (poisoning of an ARP cache by providing wrong address resolution information), crashiis (a malformed packet that causes Microsoft IIS server to crash), dosnuke (another malformed packet that crashes Microsoft Windows), syslogd (single packet to syslogd that causes it to crash), land (single TCP syn packet with source and destination being equal), and teardrop (overlapping IP fragment with bad offset value — requires two packets, but the attack itself is present only in the second one). Note, however, that other short-sequence attacks such as ping-of-death and 3-packet portsweeps are still within our scope. Also eliminated from consideration are certain attack instances (but not attack types) that are present exclusively on the "inside tcpdump" data, since our TCP state machine model was developed for the "outside tcpdump" data that records the traffic observed on the external network interface of the gateway host. Finally, we have not shown a couple of other attack instances where the tcpdump data provided by Lincoln Labs was corrupted around the time of attack. Figure 4 summarizes our result on the rest of the attacks. The highlights of our experimental results are as follows:

- *Excellent attack detection.* All of the attacks within the scope of our prototype were detected. Particularly note worthy was the detection of some stealthy portsweep attacks, some of which involved just 3 probe packets, each from a different source host!

  Another interesting aspect is that we were able to detect sweeps at all. Many anomaly detection systems incorporate knowledge into their system about such sweeps, and are explicitly programmed to look for anomalies such as "accessing so many ports within a certain period of time." In contrast, our approach has no knowledge about sweeps encoded into it. Nevertheless, it is able to detect sweeps, typically because of increased frequency of occurrence of certain abnormal transitions in the protocol state machines (e.g., the timeout transition in the IP state machine.)
- *Low false positives.* Our system generated, on the average, 5.5 false alarms per day. This is at the low end of the false alarm rates reported in the 1999 evaluation, even when misuse based approaches are taken into consideration.
- *Adequate processing capacity.* No systematic performance tuning has been attempted in our prototype implementation, and hence our performance results are to be treated as preliminary. Currently, our system can process an entire day's data within ten minutes (excluding I/O time) while running on a 700MHz Pentium III processor with 1GB memory.

A more detailed discussion of the attacks and the manner in which they are detected in our system is provided below.

5.1.1 Attacks detected by IP machine

A simplified version of our IP state machine was presented earlier in this paper. The version used in the experiment differs from this version in two ways. First, it handles packets originating from internal hosts in addition to packets originating from external hosts. Second, it handles IP fragmentation. Specifically, it treats a sequence of IP fragments that are part of the same IP packet as a single trace. (In contrast, the simplified version treats each fragment as if it is independent of other fragments.)

The statistics learnt by the IP state machine is captured by the following specification:

```
ts = (0.001, 0.01, 0.1, 1, 10, 100 and 1000)
(1) on all frequency timescale ts
(2) on all frequency wrt (src) size 100 timescale ts
(3) on all frequency wrt (dst) size 100 timescale ts
(4) on all frequency wrt (src, dst) size 100 timescale ts
```

Not all of these statistics were necessary for detecting the attacks in the Lincoln Labs data. However, one cannot easily predict in advance which of these parameters were necessary. Since we wanted to study the effectiveness of our approach in the absence of careful feature selection, we simply selected the most obvious parameters using which the traffic can be subdivided.

Based on these statistics, the following attacks are detected by the IP state machine.

IP Sweep: As mentioned earlier, IP sweeps manifest as a spurt in the frequency with which timeout transitions are taken in the IP state machine. Since the sweep is usually conducted by a single source machine, it is most obvious with statistic (2) above. Sweeps that involve a reasonable number of destination hosts also raise an anomaly in statistic (1).

Ping of Death: A ping of death attack typically involves a large number of fragmented IP packets. Thus, it manifests a spurt in the frequency of transitions that are taken when fragmented packets are received. The spurt is most noticeable when we consider a single destination, i.e., statistic (3). It is also noticeable when source destination pairs are consider, as with statistic (4), and also with statistic (2).

Smurf: This is a flooding attack involving the reception of a very large number of packets. As expected with such attacks, we witness an anomaly with almost every statistic mentioned above, but the most anomalous statistics correspond to (1) and (3).

We note that our approach, at this point, is not identifying attacks. It is only capable of producing alarm reports on each packet that results in an anomaly. A higher level system merges alarms that are temporally close together into a single alarm. While a sophisticated approach for such alarm aggregation is possible, that is not

[3]Indeed, it would be very difficult, if not impossible, to detect such single-packet attacks using anomaly detection, unless a high degree of false alarms can be tolerated.

Attack Name	Attacks Present	Attacks Detected	Description
Apache2	2	2	Dos attack on Apache web server
Back	3	3	Dos attack on Apache web server
IP Sweep	6	6	Probe to identify potential victims
Mailbomb	3	3	Large volume of mail to a server
Mscan	1	1	Attack tool
Neptune	3	3	SYN-flood attack
Ping-of-Death	4	4	Over-sized ping packets
Smurf	3	3	ICMP echo-reply flood
Queso	3	3	Stealthy probe to identify victim OS
Satan	2	2	Attack tool
Portsweep	13	13	Probing to identify exploitable servers
Total	43	43	

Figure 4: **Attacks detected in 1999 Lincoln Labs IDS Evaluation Data**

the focus of this paper. Thus, we use a simple strategy that is adequate for this data: combine alarm reports that are spaced less than a few minutes apart.

5.1.2 *Attacks Detected by the TCP Machine*

The statistics monitored by the TCP state machine is given by the following statements. Note again that we have avoided putting any great effort into feature selection. First, we have indiscriminately selected every transition in the state machine for statistics computation. Second, we have chosen to specialize this statistics collection with respect to the most obvious parameters that identify tcp sessions, namely, the source and destination addresses (or components thereof).

```
(5) on all frequency timescale ts
(6) on all frequency wrt (ext_ip) size 1000 timescale ts
(7) on all frequency wrt (int_ip) size 1000 timescale ts
(8) on all frequency wrt (ext_ip, int_ip)
        size 1000 timescale ts
(9) on all frequency wrt (int_ip, int_port)
        size 1000 timescale ts
(10) on all frequency wrt (ext_ip, int_ip, int_port)
        size 1000 timescale ts
(11) on all frequency
        wrt (ext_ip, ext_port, int_ip, int_port)
        size 1000 timescale ts
```

Here `ext_ip` and `ext_port` refer to IP address and port information on the external network (Internet), while `int_ip` and `int_port` refer to address and port information on the internal network. Some combinations such as `(ext_ip, ext_port, int_ip)` are left out since we were most interested in traffic destined for local servers, in which case the remote port information is not useful.

Portsweep: In this attack, an attacker attempts to probe for services running on a victim host by systematically attempting to access all ports. This leads to a large number of connection attempts seen at a victim host. Thus, anomalies are detected on statistics given by (7) and (8) above, if the connection attempt is a normal attempt. If the scan involves reset packets or other unusual packets, then anomalies occur in the transition from the LISTEN state to itself, which is T49 (not shown in the diagram). It is interesting to note that our approach is able to detect portsweeps that consist of 3 packets originating from 2 or 3 different hosts.

Queso: Queso is a utility program which is used to determine which operating system that is running at a certain IP address. Queso sends a series of 7 TCP packets to any one port of a machine and uses the return packets it receives to lookup the machine in a database of responses. These packets usually have unusual combinations of the TCP flags, and arrive when unexpected. Thus, we see a spurt in packets in the transition from LISTEN state to itself.

Neptune (SYN Flood): In this attack, an external host, usually using a spoofed address, sends a SYN packet to a server, thereby initiating a connection. But the attacker never responds to the SYN-ACK packet from the server. This leads to a situation known as "half-open" TCP connections on the server. Since such connections use up resources, TCP implementations limit the number of half-open connections. If this limit is exceeded, the server refuses subsequent connection requests. In our approach, we see a spurt in the frequency of timeout transitions from the half-open state. This happens on statistics (6), (7), (8), (9), (10) and (11). (If the attacker changes the (spoofed) source address quickly, then the attack can be obscured on any statistics that includes the external IP address, but it will still be detected by (7) and (9).)

Satan/Saint: SAINT is the Security Administrator's Integrated Network Tool, which probes for common vulnerabilities in services that are used most frequently. The probes generate anomalies similar to those seen with port sweeps.

Mscan: Mscan is a tool used to enumerate the systems on a network via DNS zone transfer requests, IP address scanning, etc. This attack too generates anomalies similar to port sweeps.

Mailbomb: A Mailbomb is an attack where the victim's resources are overloaded by sending exorbitant number of emails to a server, overflowing that server's mail queue and possibly causing system failure. Excessive traffic to the mail server leads to anomalies in (7), (8), (9), (10) and (11).

Apache2: This is a DOS attack that can cause an Apache web server to use disproportionate amounts of memory and CPU time by sending a large number of MIME headers with the same name. The large size of the http headers causes an increase in the frequency with which packets are received in the ESTABLISHED state of the TCP machine. The anomaly is most pronounced in the case of (10) and (9).

Back: In this denial of service attack against the Apache web server, an attacker submits requests with URL's containing many slashes. As the server tries to process these requests it will slow down. Due to its similarity with Apache2, it is detected in the same manner.

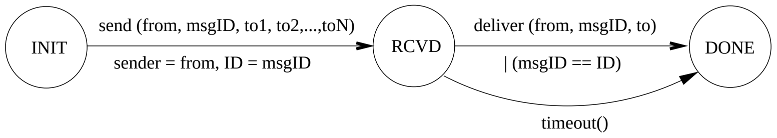
INIT
send (from, msgID, to1, to2,...,toN)
sender = from, ID = msgID
RCVD
deliver (from, msgID, to)
| (msgID == ID)
DONE
timeout()

Figure 5: Email Virus Detection State Machine

5.2 Experiments with Email Virus Detection

In order to further test our approach, we applied it to a very different problem: that of detecting email virus propagation in an intranet. The state machine model, as observed at the mail server for the intranet, is shown in Figure 5. The state machine has three states. It moves from the INIT state to RCVD state on the event $send$. This event models the action of an email client, which connects to the mail server and deposits mail. Arguments to this event are the sender, a message identifier, and the recipients. The mail is subsequently delivered to all the recipients, and this action is modeled by the event $deliver$. Note that since there is one copy of mail delivered to each recipient, and hence the $deliver$ event has only one recipient argument, unlike the $send$ event.

The statistical properties can again be specified in our language as follows.

```
ts = (10, 30, 120, 500, 2000, 8000, 25000)
(1) on all frequency timescale ts
(2) on all frequency wrt (sender) timescale ts
```

Our study was part of DARPA's SARA experiment, which was concerned with determining the effectiveness of automated responses in containing intrusions. In this experiment, a test network was set up with 400 email clients and one sendmail server. Normal email traffic was simulated using "bots" that capture typical user behavior that relates to email reading/replying/deleting etc. A variety of simple to highly sophisticated viruses were introduced, and various defense mechanisms were tested in terms of their ability to stop virus propagation. The experiment used a variety of detectors, many of which are misuse detectors that capture such aspects as excessive rate of email generation etc. Since "sneaky" viruses can evade such detectors, we also deployed an anomaly detector that was based on the approach described in this paper.

Due to the way the experiment was structured, no good response actions could be launched in response to attack reports from the anomaly detectors. The only possible option was to shut down a large number of clients and/or the email server. Since this is a drastic response, a large delay was introduced into the detection by anomaly detector, so that other detectors were able to try to control the virus before drastic actions were attempted. Specifically, the anomaly detector was tuned to detect attacks only at a point where other defensive mechanisms were unable to stop the virus, and thus, the virus was out of control.

The experiments involved hundreds of runs involving about ten different virus types, ranging from very simple viruses to very sophisticated ones. Of these, there were only seven runs where the virus was not checked by other defense mechanisms. Since our anomaly detector was tuned to detect only such cases, its performance in those seven runs is shown in Figure 6. Note again that our approach was able to detect the virus in each one of these seven runs. In addition, there were no false alarms. (Very low false alarm rate is to be expected, given that the anomaly detector was tuned to delay detection.)

6. RELATED WORK

Intrusion detection techniques can be broadly classified into *misuse* detection, *anomaly* detection and *specification based* detection. *Misuse* detection [20, 12], which detects known misuses accurately, is not very effective against unknown attacks. *Anomaly* detection [1, 5, 6] copes better with unknown attacks, but can generate a lot of false positives, and hence not deployed widely. Specification-based approach [11, 23] is a recently developed technique that can detect novel attacks, while maintaining a low degree of false alarms.

Most network intrusion detection systems [8, 19, 9, 13, 17, 25] reconstruct higher level interactions between end hosts and remote users, and identify anomalous or attack behaviors. Other approaches operate mainly on the basis of packet header contents [21, 24, 22]. These techniques provide a way to define signatures not only on the basis of textual data in the reconstructed TCP sessions, but also on packet fields. These approaches can provide better detection of certain classes of attacks (especially, probing attacks) that do not result in valid TCP sessions. Our approach also relies primarily on inspecting network packer fields, but can use data in the reconstructed sessions if necessary.

Data mining is concerned with the extraction of useful information from large volumes of data, thus it is natural to ask if this technique can be used to extract attack detection rules from large volumes of network traffic data. [13] was one of the first works to propose the use of data mining techniques for intrusion detection. Since then, this topic has received substantial research interest, with a lot of ongoing activity. As compared to our approach, the main difference is that these works still rely much more on expert identification of useful features for network intrusion detection. For instance, [13] selects a long list of features that include, among many others, the following: successful TCP connection, connection rejection, failure to receive SYN-ACK, spurious SYN-ACKs, duplicate ACK rate, wrong size rate, bytes sent in each direction, normal connection termination, half-closed connections, and failure to send all data packets. In our approach, we do not rely on such expert judgment to identify features, but on the protocol state machine specifications.

The NATE (Network Analysis of Anomalous Traffic Events) system [24] uses statistical clustering techniques to learn normal behavior patterns in network data. Training data is used in the formation of clusters, or groups, of similar data. During detection, data points that do not fall into some cluster are seen as anomalous. Clustering requires the use of some similarity measure and, for network data, sampling techniques are also necessary. NATE was able to detect most network probes and DOS attacks in the MIT Lincoln Labs data. No comprehensive information on false positives/negatives is provided in [24]. The technique used by NATE is sensitive to the the sampling methodology and distance measure used, so continuing research is involved in trying to develop more accurate methods. Unlike our approach, NATE requires the use of sampling to select a small subset of packet data for training. Moreover, the information learnt by NATE requires checking by a human before it is used for detection. Perhaps most important, attack detection in [24] is based on identifying anomalous data values in individual packets, whereas our approach is focused mainly on properties of packet sequences.

The EMERALD system [19] contains a statistical component called eStat described partly in [18]. This statistical component maintains short and long-term distribution information for several types of "measures", using a decay mechanism to age out less recent events. While the techniques do not require prior knowledge

Virus type	Time of detection	Traffic due to virus
Simple virus	3.7 min	< 5 %
Polymorphic virus	36.4 min	< 5 %
Persistent Polymorphic	3.0 min	< 5 %
Persistent Polymorphic variant I (fast propagation)	2.2 min	< 5 %
Persistent Polymorphic variant II (slow propagation)	22.7 min	< 5 %
Persistent Polymorphic variant III (medium propagation)	3.3 min	< 5 %
Persistent Polymorphic variant IV (multiple attachment types)	3.1 min	< 5 %

Figure 6: Email Virus Detection Performance

of attack activity, such knowledge is used in the choice of attributes that constitute measures and time ranges used for intensity measures.

EMERALD also has a component that combines signature and anomaly-based approaches called eBayes. EBayes uses a belief network to determine from a number of features whether the values of those features fits with some normal behavior (http, ftp, etc.), some predefined bad behavior (mailbomb, ipsweep, etc.), or neither of these (other).

7. CONCLUSIONS

In this paper, we presented a new approach for network intrusion detection called specification-based anomaly detection. Through our experiments, we showed that the new approach combines the primary benefits of anomaly detection and specification-based detection, namely, good detection of unknown attacks and low false alarm rates. At the same, the new approach alleviates the principal problems associated with either approach — specification development is guided by protocol specifications, and is hence simplified. Moreover, only a handful of protocols need to be specified in order to detect most attacks.

We showed that protocol specifications simplify manual feature selection process that often plays a major role in other anomaly detection approaches. In particular, most attacks discussed in the experimentation section could be detected by simply monitoring frequency distribution information associated with state machine transitions. Detection of other attacks required further partitioning of frequency information based on sources and destinations of network packets. Thus, in these experiments, our approach enables features to be selected without any significant degree of analysis or insight.

Another contribution of this paper is the specification language for modeling state machines and for succinctly stating the anomaly detection information to be learnt. This language makes it easy to apply our approach to deal with other higher layer (such as HTTP) or lower layer (e.g., ARP) protocols.

8. REFERENCES

[1] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[2] T. Bowen, D. Chee, M. Segal, R. Sekar, P. Uppuluri, and T. Shanbhag, Building Survivable Systems: An Integrated Approach Based on Intrusion Detection and Confinement, DISCEX 2000.

[3] P.K. Chan and S. Stolfo, Toward parallel and distributed learning by metalearning, AAAI workshop in Knowledge Discovery in Databases, 1993.

[4] D. Denning, An Intrusion Detection Model, IEEE Trans. on Software Engineering, Feb 1987.

[5] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.

[6] A. Ghosh, A. Schwartzbard and M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999.

[7] J. Haines, R. Lippmann, D. Fried, E. Tran, S. Boswell and M. Zissman, 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures, MIT Lincoln Laboratory Technical Report TR-1062, 2001.

[8] L. Heberlein et al, A Network Security Monitor, Symposium on Research Security and Privacy, 1990.

[9] J. Hochberg et al, NADIR: An Automated System for Detecting Network Intrusion and Misuse, Computers and Security 12(3), May 1993.

[10] G. Jakobson and M. Weissman, Alarm Correlation, IEEE Network, Vol. 7, No. 6., 1993.

[11] C.Ko, M. Ruschitzka and K. Levitt, Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach, IEEE Symposium on Security and Privacy, 1997.

[12] S. Kumar and E. Spafford, A Pattern-Matching Model for Intrusion Detection, Nat'l Computer Security Conference, 1994.

[13] W. Lee and S. Stolfo, Data Mining Approaches for Intrusion Detection, USENIX Security Symposium, 1998.

[14] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham and M. Zissman, Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation, Proceedings of the DARPA Information Survivability Conference and Exposition, 2000.

[15] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, Lawrence Berkeley Laboratory, Berkeley, CA, 1992.

[16] B. Mukherjee, L. Heberlein and K. Levitt, Network Intrusion Detection, IEEE Network, May/June 1994.

[17] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, USENIX Security Symposium, 1998.

[18] P. Porras and A. Valdes, Live Traffic Analysis of TCP/IP Gateways, Networks and Distributed Systems Security Symposium, 1998.

[19] P. Porras and P. Neumann, EMERALD: Event Monitoring Enabled Responses to Anomalous Live Disturbances, National Information Systems Security Conference, 1997.

[20] P. Porras and R. Kemmerer, Penetration State Transition Analysis:A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.

[21] M. Roesch, Snort: Lightweight intrusion detection for networks, USENIX LISA Conference, 1999.

[22] R. Sekar, Y. Guang, T. Shanbhag and S. Verma, A High-Performance Network Intrusion Detection System, ACM Computer and Communication Security Conference, 1999.

[23] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, USENIX Security Symposium, 1999.

[24] C. Taylor and J. Alves-Foss. NATE — Network Analysis of Anomalous Traffic Events, A Low-Cost Approach, New Security Paradigms Workshop, 2001.

[25] G. Vigna and R. Kemmerer, NetSTAT: A Network-based Intrusion Detection Approach, Computer Security Applications Conference, 1998.