A Temporal Logic Based Framework for Intrusion Detection

Prasad Naldurg, Koushik Sen, and Prasanna Thati

Department of Computer Science,
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{naldurg,ksen,thati}@cs.uiuc.edu

Abstract. We propose a framework for *intrusion detection* that is based on runtime monitoring of temporal logic specifications. We specify intrusion patterns as formulas in an expressively rich and efficiently monitorable logic called EAGLE. EAGLE supports data-values and parameterized recursive equations, and allows us to succinctly express security attacks with complex temporal event patterns, as well as attacks whose signatures are inherently statistical in nature. We use an online monitoring algorithm that matches specifications of the absence of an attack, with system execution traces, and raises an alarm whenever the specification is violated. We present our implementation of this approach in a prototype tool, called MONID and report our results obtained by applying it to detect a variety of security attacks in log-files provided by DARPA.

Key Words: Intrusion detection, security, temporal logic, runtime monitoring.

1 Introduction

Despite great progress in research on computer security, fully secure computer systems are still a distant dream. Today any large and complex computer system has many security flaws. *Intrusion detection* involves monitoring the system under concern to identify the misuse of these flaws as early as possible in order to take corrective measures.

There are two main approaches to intrusion detection: *signature-based* [10, 12] and *anomaly-based* [1, 6, 14]. In the signature-based approach, system behavior is observed for known patterns of attacks, while in the anomaly-based approach an alarm is raised if an observed behavior deviates significantly from pre-learned normal behavior. Both these approaches have relative advantages and disadvantages. The signature-based approach has a low false-alarm rate, but it requires us to know the patterns of security attacks in advance and previously unknown attacks would go undetected. In contrast, the anomaly-based approach can detect new attacks, but has a high false-alarm rate.

In this paper, we adopt a *temporal logic* approach to signature-based intrusion detection. One can naturally specify the absence of a known attack pattern as a *safety* formula ϕ in a suitable temporal logic [5]. Such a temporal logic based approach was considered in [16] using a variant of linear temporal logic (LTL) with first order variables. However we consider a more expressive logic in which one can also express attack signatures involving real-time constraints and statistical properties. We show how to automatically monitor the specification ϕ against the system execution and raise an intrusion alarm whenever the specification is violated. We also show how this technique can be used for simple types of anomaly-based intrusion detection. The idea is to specify the intended behavior of security-critical programs as temporal formulas involving *statistical predicates*, and monitor the system execution to check if it violates the formula. If the observed execution violates the formula then an intrusion has occurred, and thus attacks can be detected even if they are previously unknown.

Our approach to intrusion detection is motivated by the success of the relatively new research area called *runtime verification* [8, 18, 20], a major goal of which is to use light-weight formal methods for system monitoring. We use EAGLE, introduced in [4], for specification of attack-safe behavior of a system. EAGLE supports recursively defined temporal formulas, parameterizable by both logical formulas and data-expressions, over a set of three primitive modalities “next”, “previous”, and “concatenation”. The logic enables us to express temporal patterns that involve reasoning about the data-values observed in individual events, and thus allows us to specify attacks whose signatures are inherently statistical in nature. Examples include password guessing attacks or the ICMP-flood denial of service attack. For these attacks there is no clear distinction between an intrusion and a normal behavior, and their detection involves collecting temporal statistics at runtime and making a guess based on the collected statistics.

We use an online algorithm [4] to monitor EAGLE formulas that processes each event as soon as it occurs and modifies the monitored formula to store the relevant summary. If, after any event the modified formula becomes false, an intrusion alarm is raised. Thus the whole procedure works in real-time. We have implemented our proposed approach in a prototype tool called MONID which can detect intrusions either online or offline. Figure 1 illustrates the framework. Information about system-level events, obtained either from relevant log-files (offline) or generated by appropriately instrumented application code (online), are sent to a server. The server merges the events from various sources by timestamp and preprocesses them into an abstract intermediate form to generate a single event trace. Note that detecting certain attacks may require us to observe events from various sources. Our monitor subsequently monitors this event trace against a given specification, and raises an intrusion alarm if the specification is violated.

We show the effectiveness of our approach by specifying several types of attacks and by monitoring them using MONID. Specifically, we perform offline monitoring using the large log-files made available by DARPA exclusively for the task of evaluating intrusion detection systems [13]. We successfully detected the attacks specified with acceptable computational overheads for the monitoring procedure. The experiments suggest that the proposed approach is a viable complement to existing intrusion detection mechanisms.

Following is the layout of the rest of this paper: In Section 2 we discuss related work in the area of intrusion detection and motivate our work. In Section 3, we briefly describe the syntax, semantics, and monitoring algorithm for EAGLE followed by Section 4 where we illustrate several common security-attack patterns specified in EAGLE. In Section 5, we describe the implementation of our tool MONID followed by a sum-

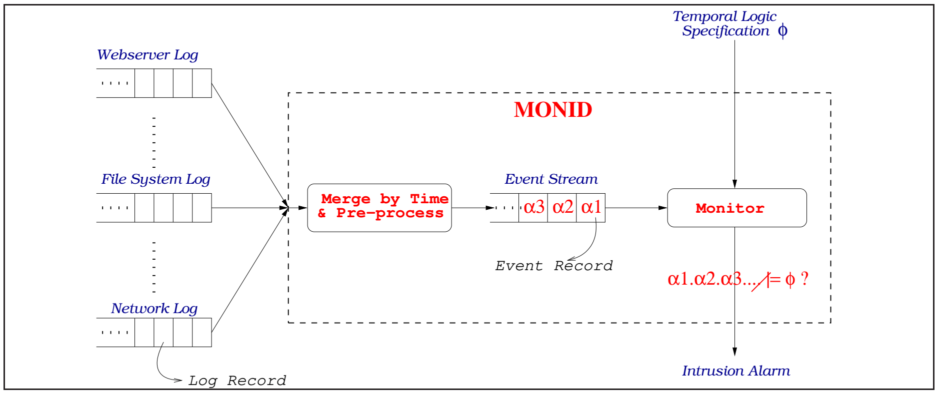

Fig. 1. MONID: A framework for intrusion detection

mary of our experimental results on DARPA log-files. We conclude in Section 6 with a brief discussion about future research directions.

2 Background and Motivation

The area of intrusion detection has seen synthesis of concepts and techniques from a variety of disciplines, including expert systems [1, 17], artificial neural networks [6], data mining [14], and static analysis [22]. A diverse collection of tools based on these various approaches have been deployed and tested [2, 7]. In the following, we elaborate on some of these approaches and clarify how our work fits in this context.

For signature-based approaches there are several languages with varying degrees of expressivity for specifying attack patterns. Roger et al. in [16] used temporal logic and model-checking based approach to detect intrusion. This work is closely related to ours; however, unlike [16] we can express more sophisticated signatures involving statistics and real-time by using powerful monitoring logic EAGLE. Ilgun *et al* [10] propose the use of finite state transition diagrams to specify sequences of actions that would lead the system from a secure initial state to a compromised final state. Ko *et al* [11] introduce a new class of grammars, called parallel environment grammars that are specifically suitable for specifying behavior (traces) of concurrent processes. The expected behavior of security critical programs is specified by a grammar, and an alarm is raised if the observed execution trace is not in the language defined by the grammar. Kumar *et al* [12] propose the use of Colored Petri nets for specifying attack patterns. We note that in comparison to the other approaches, temporal logic specifications of attack signatures tend to be much more compact and simpler to describe.

The state transition diagram and colored Petri net approaches can be seen as special cases of rule-based expert systems [9]. In rule-based expert systems, in general, knowledge about attacks is represented as a collection of `if-then` rules which are fired in response to the observed system execution. The main advantage of this approach is the clear separation of knowledge base from the control mechanism that applies the knowledge base for detecting intrusions.

In contrast to the signature-based approaches such as the above, the anomaly-based approach to intrusion detection does not require a priori knowledge of the attacks. One such approach is to collect statistical information [1, 15] about normal behavior into a user, group or target machine profile, and raise an alarm if the observed behavior deviates significantly from an estimated profile. One of the most rudimentary ones is *threshold detection*, where the idea is to record the number of occurrences of specific events and raise an alarm if the number is not within an expected range. As we will see in Section 3, such threshold detection policies can be elegantly expressed as temporal logic formulas.

Statistical profile-based anomaly detection can be seen as an instance of the general class of intrusion detection systems that *learn* the normal system behavior by constructing some model for it, and use the model to predict the system behavior and detect suspicious deviations. Other approaches in this category include that of time-based inductive generalization [21], artificial neural networks [6], and data-mining[14]. In time-based inductive generalization, the system behavior is modeled as a set of rules that are dynamically modified during the learning phase depending on how the predictions of the rules match with the observed system behavior. In the artificial neural networks approach, a neural net constitutes the model of system behavior and the net is trained with representative normal scenarios. In the data-mining approach, large audit trails are mined for patterns of normal or abusive behavior, which are then used for signature-based or anomaly-based intrusion detection. Self-learning capabilities such as the above are beyond the scope of our approach which is only intended for detecting attacks whose patterns are known a priori.

Anomaly-based intrusion detection systems have the disadvantage of having high false alarm rates due to inaccuracies in the learned model. In contrast, signature-based approaches such as ours have low false alarm rate, but would fail to detect attacks that differ even slightly from the given signature. Ideally, one would like a self-learning intrusion detection system with low false alarm rates. For now, the signature-based systems are quite popular because of their simplicity, accuracy, and ease of use. These systems would in any case be a valuable supplement to build more accurate anomaly-based systems.

3 EAGLE: An Expressive Temporal Monitoring Logic

The patterns for security attacks in software systems are specified formally in the logic EAGLE which is designed to support finite trace monitoring, and contains a small set of operators. The logic EAGLE introduced in [4] supports recursive parameterized equations, with a minimal/maximal fix-point semantics, together with three temporal operators: next-time ($\bigcirc$), previous-time ($\odot$), and concatenation ($\cdot$). Rules which are used to define new temporal operators can be parameterized with formulas and data-values, thus supporting specifications that can involve data, which can span an execution trace. The expressivity of EAGLE, which is indeed very rich, as shown in [3, 4], can express properties involving real-time, statistics and data-values. To make the paper self-contained, in this section, we give an informal introduction to EAGLE followed by its syntax, semantics, and the runtime monitoring algorithm for EAGLE as described in [4].

The logic EAGLE and its monitoring algorithm assumes the following:

1. There is a finite sequence of events σ generated by some executing system. An event is an instance of a record having a pre-specified schema. For example,

$$\texttt{LoginLogoutEvent}\{\mathit{userId}\colon \underline{\text{string}}, \mathit{action}\colon \underline{\text{int}}, \mathit{time}\colon \underline{\text{double}}\}$$

 is the schema of an event and $\{\mathit{userId} = \texttt{"Bob"}, \mathit{action} = \texttt{login}, \mathit{time} = 18.7\}$ is an event representing the fact that user "Bob" has logged in at time 18.7.
2. There is a formula F in EAGLE which specifies the condition for the absence of an attack.

We say that σ is free of the attack specified by F if and only if σ satisfies F.

Now, assume that we want to state a property that *"Whenever there is a login then eventually there is a logout"*. The property can be written in classical future time LTL: $\Box(\mathit{action} = \texttt{login} \rightarrow \Diamond(\mathit{action} = \texttt{logout}))$. The formulas $\Box F$ (always F) and $\Diamond F$ (eventually F), for some property F, satisfy the following equivalences, where the temporal operator $\bigcirc F$ stands for *next* F (meaning '*in next state* F'):

$$\Box F \equiv F \wedge \bigcirc(\Box F) \qquad \Diamond F \equiv F \vee \bigcirc(\Diamond F)$$

One can show that $\Box F$ is a maximal solution of the recursive equivalence $X \equiv F \wedge \bigcirc X$, while $\Diamond F$ is the minimal solution of $X \equiv F \vee \bigcirc X$. In EAGLE one can write the following definitions for the two combinators `Always` and `Eventually`, and the formula to be monitored (M_1):

$$\begin{array}{l}
\underline{\text{max}}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc\texttt{Always}(F) \\
\underline{\text{min}}\ \texttt{Eventually}(\underline{\text{Form}}\ F) = F \vee \bigcirc\texttt{Eventually}(F) \\
\underline{\text{mon}}\ M_1 = \texttt{Always}((\mathit{action} = \texttt{login}) \rightarrow \texttt{Eventually}(\mathit{action} = \texttt{logout}))
\end{array}$$

The `Always` operator is defined as having a maximal fix-point interpretation; the `Eventually` operator is defined as having a minimal interpretation. For further details the readers are referred to [4].

Let us complicate the above property a bit by stating that *"Whenever there is a login by any user x then eventually the user x logs out."* Thus, if "Bob" logs in then eventually "Bob" must logout. Similarly, the property must hold for any user such as "Tom", "Jim" or "Kelly". This property can be expressed by the following LTL formula with data-value bindings:

$$\Box((\mathit{action} = \texttt{login}) \rightarrow \underline{\text{let}}\ k = \mathit{userId}\ \underline{\text{in}}\ \Diamond(\mathit{action} = \texttt{logout} \wedge \mathit{userId} = k))$$

In this formula we use the operator $\underline{\text{let}}$ _ $\underline{\text{in}}$ _ to bind the value of *userId* in the current event to the local variable k whenever $\mathit{action} = \texttt{login}$ in the current event. We then impose the condition that the value of *userId* in some event in future must be same as the user id bound to k and that the action of the event must be `logout`. In EAGLE, we use a parameterized rule to express this property, capturing the value of *userId* as a rule parameter:

$$\begin{array}{l}
\underline{\text{min}}\ \texttt{Bind}(\underline{\text{string}}\ k) = \texttt{Eventually}(\mathit{action} = \texttt{logout} \wedge \mathit{userId} = k) \\
\underline{\text{mon}}\ M_2 = \texttt{Always}((\mathit{action} = \texttt{login}) \rightarrow \texttt{Bind}(\mathit{userId}))
\end{array}$$

Rule `Bind` is parameterized with a string k, and is instantiated in M_2 when $\mathit{action} = \texttt{login}$, hence capturing the value of *userId* at that moment. Rule `Bind` replaces the binding operator $\underline{\text{let}}$ _ $\underline{\text{in}}$ _.

Indeed one can combine the two rules `Bind` and `Eventually` into a single rule `EvLogout` with one parameter to get the same monitor as follows:

$$\underline{\text{min}}\ \texttt{EvLogout}(\underline{\text{string}}\ k) = (\mathit{action} = \texttt{logout} \wedge \mathit{userId} = k) \vee \bigcirc \texttt{EvLogout}(k)$$
$$\underline{\text{mon}}\ M_2 = \texttt{Always}((\mathit{action} = \texttt{login}) \rightarrow \texttt{EvLogout}(\mathit{userId}))$$

Thus by allowing parameterized rules one gets the power of data-value binding in a formula. It can be argued that the introduction of the operator let _ in _ is sufficient to get the power of binding. However, parameterized rules can do more than simple data-binding. For example, suppose we want to express the property that *"Whenever there is a login by any user x then eventually the user x logs out within 100 units of time."* For this property we modify the rule `EvLogout` by introducing two more parameters denoting the time at which the previous event took place and the time left. The modified rule and the monitor is given below:

$$\underline{\text{min}}\ \texttt{EvTimedLogout}(\underline{\text{string}}\ k, \underline{\text{double}}\ t, \underline{\text{double}}\ \delta) = (\delta - (\mathit{time} - t) \geq 0)$$
$$\wedge((\mathit{action} = \texttt{logout} \wedge \mathit{userId} = k) \vee \bigcirc \texttt{EvTimedLogout}(k, \mathit{time}, \delta - (\mathit{time} - t)))$$
$$\underline{\text{mon}}\ M_3 = \texttt{Always}((\mathit{action} = \texttt{login}) \rightarrow \texttt{EvTimedLogout}(\mathit{userId}, \mathit{time}, 100))$$

Note that another simpler alternative to define the rule `EvTimedLogout` is as follows:

$$\underline{\text{min}}\ \texttt{EvTimedLogout}(\underline{\text{string}}\ k, \underline{\text{double}}\ t, \underline{\text{double}}\ \delta) = (\mathit{time} - t \leq \delta)$$
$$\wedge((\mathit{action} = \texttt{logout} \wedge \mathit{userId} = k) \vee \bigcirc \texttt{EvTimedLogout}(k, t, \delta))$$

A possible variation of our requirement for login and logout can be stated as *"Whenever there is a logout by any user x then in the past user x must have logged in"*. This property cannot be expressed concisely using the future time temporal operators only. However, the property can be expressed elegantly by a mixture of past-time and future-time temporal operators as follows:

$$\Box((\mathit{action} = \texttt{logout}) \rightarrow \underline{\text{let}}\ k = \mathit{userId}\ \underline{\text{in}}\ \Diamond\!\!\!\cdot\,(\mathit{action} = \texttt{login} \wedge \mathit{userId} = k))$$

where $\Diamond\!\!\!\cdot\, F$ denotes eventually in past F. This operator can be defined recursively in EAGLE using the primitive operator $\odot$ which is the past-time equivalent of $\bigcirc$. Thus the monitor definition can be written as follows:

$$\underline{\text{min}}\ \texttt{EventuallyInPast}(\underline{\text{Form}}\ F) = F \vee \odot \texttt{EventuallyInPast}(F)$$
$$\underline{\text{min}}\ \texttt{Bind}(\underline{\text{string}}\ k) = \texttt{EventuallyInPast}(\mathit{action} = \texttt{logout} \wedge \mathit{userId} = k)$$
$$\underline{\text{mon}}\ M_4 = \texttt{Always}((\mathit{action} = \texttt{login}) \rightarrow \texttt{Bind}(\mathit{userId}))$$

Thus rules in EAGLE allow us to define customized temporal operators with also the ability to bind and manipulate data. This capability proves to be indispensable for succinctly expressing attack-safe system executions. We recall the syntax and semantics of EAGLE to make the paper self-contained.

3.1 Syntax and Semantics

Syntax A specification S consists of a declaration part D and an observer part O. D consists of zero or more rule definitions R, and O consists of zero or more monitor definitions M, which specify what is to be monitored. Rules and monitors are named (N).

$$\begin{array}{lll} S ::= D\ O & D ::= R^* & O ::= M^* \\ R ::= \{\underline{\text{max}} \mid \underline{\text{min}}\}\ N(T_1\ x_1,\ldots,T_n\ x_n) = F & & \\ M ::= \underline{\text{mon}}\ N = F & & \\ T ::= \underline{\text{Form}} \mid \textit{primitive type} & & \\ F ::= \textit{expression} \mid \underline{\text{true}} \mid \underline{\text{false}} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid & & \\ \quad\ \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1,\ldots,F_n) \mid x_i & & \end{array}$$

A rule definition R is preceded by a keyword indicating whether the interpretation is maximal or minimal. Maximal rules define safety properties (nothing bad ever happens), while minimal rules define liveness properties (something good eventually happens). For us, the difference only becomes important when evaluating formulas at the boundaries of a trace. To understand how this works it suffices to say here that monitored rules evolve as new events are appearing. Assume that the end of the trace has been reached (we are beyond the last event) and a monitored formula F has evolved to F'. Then all applications in F' of maximal fix-point rules will evaluate to true, since they represent safety properties that apparently have been satisfied throughout the trace, while applications of minimal fix-point rules will evaluate to false, indicating that some event did not happen.

The rule parameters are typed and can either be a formula of type Form, or of a primitive type such as int, long, float, etc., or any other composite types such as Set, List, etc.. The body of a rule (or monitor) is a boolean valued formula of the syntactic category *Form* (with meta-variables F, etc.). Any recursive call on a rule must be strictly guarded by a temporal operator. The propositions of this logic are boolean expressions over fields of event. Formulas are composed using standard propositional logic operators together with a next-state operator ($\bigcirc F$), a previous-state operator ($\odot F$), and a concatenation-operator ($F_1 \cdot F_2$). Finally, rules can be applied and their arguments must be type correct. That is, an argument of type Form can be any formula, with the restriction that if the argument is an expression, it must be of boolean type. An argument of a primitive type must be an expression of that type. Arguments can be referred to within the rule body (x_i).

In what follows, a rule N of the form

$$\{\underline{\text{max}}|\underline{\text{min}}\}\ N(\underline{\text{Form}}\ f_1,\ldots,\underline{\text{Form}}\ f_m, T_1\ p_1,\ldots,T_n\ p_n) = B,$$

where $f_1,\ldots f_m$ are arguments of type Form and $p_1,\ldots p_n$ are arguments of primitive type, is written in short as: $\{\underline{\text{max}}|\underline{\text{min}}\}\ N(\overline{\underline{\text{Form}}}\ \overline{f}, \overline{T}\ \overline{p}) = B$, where $\overline{f}$ and $\overline{p}$ represent tuples of type $\overline{\underline{\text{Form}}}$ and $\overline{T}$ respectively. Without loss of generality, in the above rule we assume that all the arguments of type Form appear first.

Semantics An execution trace σ is a finite sequence of events $\sigma = s_1 s_2 \ldots s_n$, where $|\sigma| = n$ is the length of the trace. The i'th event s_i of a trace σ is denoted by $\sigma(i)$. The term $\sigma^{[i,j]}$ denotes the sub-trace of σ from position i to position j, both positions included; if $i \geq j$ then $\sigma^{[i,j]}$ denotes the empty trace. Given a trace σ and a specification $D\ O$, satisfaction is defined as follows:

$$\sigma \models D\ O \ \text{ iff } \ \forall\ (\underline{\text{mon}}\ N = F) \in O\ .\ \sigma, 1 \models_D F$$

That is, a trace satisfies a specification if the trace, observed from position 1 (the first state), satisfies each monitored formula. The definition of the satisfaction relation $\models_D \subseteq (Trace \times \mathbf{nat}) \times \underline{\text{Form}}$, for a set of rule definitions D, is presented below, where $0 \leq i \leq n+1$ for some trace $\sigma = s_1 s_2 \ldots s_n$. Note that the position of a trace can become 0 (before the first state) when going backwards, and can become $n+1$ (after the last state) when going forwards, both cases causing rule applications to evaluate to either true if maximal or false if minimal, without considering the body of the rules at that point.

$$
\begin{array}{lll}
\sigma, i \models_D \mathit{expression} & \text{iff} & 1 \leq i \leq |\sigma| \text{ and } \mathit{evaluate}(\mathit{expression})(\sigma(i)) == \mathit{true} \\
\sigma, i \models_D \underline{\text{true}} & \text{iff} & \sigma, i \not\models_D \underline{\text{false}} \\
\sigma, i \models_D \neg F & \text{iff} & \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff} & i \leq |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \odot F & \text{iff} & 1 \leq i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff} & \exists j \text{ s.t. } i \leq j \leq |\sigma|+1 \text{ and } \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(\overline{F}, \overline{P}) & \text{iff} & \left\{ \begin{array}{l}
\text{if } 1 \leq i \leq |\sigma| \text{ then:} \\
\quad \sigma, i \models_D B[\overline{f} \mapsto \overline{F}, \overline{p} \mapsto \mathit{evaluate}(\overline{P})(\sigma(i))] \\
\quad \text{where } (N(\underline{\text{Form}}\ \overline{f}, \overline{T}\ \overline{p}) = B)\ \in D \\
\text{otherwise, if } i = 0 \text{ or } i = |\sigma|+1 \text{ then:} \\
\quad \text{rule } N \text{ is defined as } \underline{\text{max}} \text{ in } D
\end{array} \right.
\end{array}
$$

An expression (a proposition) is evaluated at the current event in case the position i is within the trace ($1 \leq i \leq n$). In the boundary cases ($i = 0$ and $i = n+1$) a proposition evaluates to false. Propositional operators have their standard semantics in all positions. A next-time formula $\bigcirc F$ evaluates to true if the current position is not beyond the last event and F holds in the next position. Dually for the previous-time formula. The concatenation formula $F_1 \cdot F_2$ is true if the trace σ can be split into two sub-traces $\sigma = \sigma_1 \sigma_2$, such that F_1 is true on σ_1, observed from the current position i, and F_2 is true on σ_2 (ignoring σ_1, and thereby limiting the scope of past time operators). Applying a rule within the trace (positions $1 \ldots n$) consists of replacing the call with the right-hand side of the definition, substituting arguments for formal parameters; if an argument is of primitive type its evaluation in the current state is substituted for the associated formal parameter of the rule, thereby capturing a desired freeze variable semantics.

3.2 The Monitoring Algorithm

We briefly describe the computation mechanism used to check if an EAGLE formula is satisfied by a sequence of events. We assume that the *propositions* or the *expressions* of an EAGLE formula are specified with respect to the fields of the event record. At every event the algorithm evaluates the monitored formula on the event and generates another formula. At the end of the event sequence, the value of the evolved formula is determined; if the value is true the formula is satisfied by the event sequence, otherwise, the formula is violated.

Formally, the evaluation of a formula F at an event $s = \sigma(i)$ results in an another formula $F' = \mathit{eval}(F, s)$ with the property that $\sigma, i \models F$ if and only if $\sigma, i+1 \models F'$. At the end of the trace we compute the boolean function $\mathit{value}(F)$, where F is the evolved formula, such that $value(F)$ is true if and only if $\sigma, |\sigma|+1 \models F$ and false otherwise. Thus for a given trace $\sigma = s_1 s_2 \ldots s_n$ and an EAGLE formula F, σ satisfies F if and only if $value(eval(\ldots eval(eval(F, s_1), s_2) \ldots, s_n)) = true$. The details of the algorithm can be found in [4] which gives the definition of the functions *eval* and *value* along with two other auxiliary functions *update* and *init*. The definition of these four functions forms the calculus of EAGLE. For this paper, to help in understanding, we describe the algorithm informally through an example.

Suppose we want to monitor the following specification, described in Section 3

$$
\begin{aligned}
&\underline{\text{max}}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc\texttt{Always}(F)\\
&\underline{\text{min}}\ \texttt{EvTimedLogout}(\underline{\text{string}}\ k, \underline{\text{double}}\ t, \underline{\text{double}}\ \delta) = (time - t \leq \delta)\\
&\qquad\qquad\qquad\qquad \wedge((action = \texttt{logout} \wedge userId = k) \vee \bigcirc\texttt{EvTimedLogout}(k, t, \delta))\\
&\underline{\text{mon}}\ M_3 = \texttt{Always}((action = \texttt{login}) \rightarrow \texttt{EvTimedLogout}(userId, time, 100))
\end{aligned}
$$

against the sequence of 2 events $e_1 = \{userId = \texttt{"Bob"}, action = \texttt{login}, time = 17.0\}$, $e_2 = \{userId = \texttt{"Bob"}, action = \texttt{logout}, time = 150.0\}$. At the first event the formula M_3 gets modified as follows:

$$
\begin{aligned}
eval(M_3, e_1) = \texttt{EvTimedLogout}(\texttt{"Bob"}, 17.0, 100) \wedge&\\
\texttt{Always}((action = \texttt{login}) \rightarrow \texttt{EvTimedLogout}(userId, time, 100))&
\end{aligned}
$$

Note that the relevant summary of the event, involving the user id "Bob" and the timestamp 17.0 has got assimilated into the modified formula. At the second event the predicate $(time - t \leq \delta)$ gets instantiated to $(150.0 - 17.0 \leq 100)$ which is false. Hence the whole formula becomes false, indicating that the two-event trace violates the property.

One point which is worth stressing on is that the logic EAGLE has a finite trace semantics. The monitoring algorithm as described above can determine the satisfaction or the violation of a formula at the end of a trace only. However, in intrusion detection, end of trace makes no sense as the sequence of events can be theoretically infinite. In that situation we want to raise an alarm as soon as a property is violated. This is done by checking after every event if the formula becomes unsatisfiable. Checking unsatisfiability of a formula in EAGLE is undecidable as it involves data-values. However, note that it is always possible to write a formula, corresponding to the absence of an attack, such that, whenever the attack pattern appears in the event sequence, the formula becomes false. The reason behind this is that we can always specify an attack pattern by a formula ϕ such that ϕ, when evaluated over a sequence of events representing the pattern becomes true. This is called *specifying-bad prefixes* [19]. Once we have the attack pattern specification in terms of ϕ, we can specify the safe behavior of the system as $\Box(\neg\phi)$. Note that this formula becomes false (hence unsatisfiable) whenever a sequence of events representing the attack is detected. Hence, checking unsatisfiability for the evaluation of this formula at any point simply reduces to checking if the evaluated formula is false.

3.3 EAGLE FLIER: Monitoring Engine

We use the monitoring engine EAGLE FLIER to implement our intrusion detection framework, called MONID. The engine EAGLE FLIER, written in Java, is available as a library. The library provides two basic methods that can be called by any client program for the purpose of monitoring. The first method `parse` takes a file containing a specification involving several monitors (sets of monitored formulas) written in EAGLE and compiles them internally into data structures representing monitors. After compilation, the client program calls the method `eval` iteratively for every event. This call internally modifies the monitors according to the definition of *eval* in Subsection 3.2. If at any step the monitored formulas become false, an error message is printed or a pre-specified method is invoked to take a corrective measure.

4 Example Attack Specifications

In this section, we present a few examples of how EAGLE can be used to specify formulas that correspond to desirable properties of execution traces of a system being monitored. An intrusion or an attack in this context is a trace that violates this specification. We draw our examples from real-world attacks to showcase the applicability of our framework. These examples highlight the expressive power of our formalism, as well as exemplify the various features of EAGLE. Moreover, we believe that many attack signatures in practice can be expressed using templates of our examples with minor modifications. In all the examples we use the rule `Always` defined in Section. 3.

Smurf attacks

The first attack we describe is the Smurf IP Denial of Service (DoS) attack. An attacker creates a forged ICMP echo request message (also called a "ping" packet) with the victim's name as the sender and sets the destination IP address to a broadcast IP address. This attack can result in a large amount of ICMP echo reply packets being sent from broadcast hosts that respond to the request, to the victim, which can cause network congestion or outages.

In order to detect this attack, we need to look at network events from a log created by a network auditing tool such as *tcpdump*. The formula for the absence of this attack is given below:

$$\begin{array}{l} \underline{\mathrm{max}}\ \mathtt{Attack}() = (\mathit{type} = \text{“}\mathit{ICMP}\text{”}) \wedge \mathit{isBroadcast}(\mathit{ip}) \\ \underline{\mathrm{mon}}\ \mathtt{SmurfSafety} = \mathtt{Always}(\neg\mathtt{Attack}()) \end{array}$$

In the above example, the record schema of an event contains the *type* field that corresponds to the type of the network packet and the field *ip* that corresponds to the return IP address of a network packet. In the specification, we first specify the attack pattern by the rule `Attack` that checks if the type of the packet is *"ICMP"* and the destination address of the packet is a broadcast IP address (*isBroadcast*), where *isBroadcast* is a predicate over the event which checks if the last two bytes of the IP address provided as argument are 0 or 255. Then a good behavior of the system with respect to this attack can be stated as "*Always there is no attack*".

Cookie-Stealing Scenario

The next example describes what is called the "cookie-stealing" attack. In order to monitor this attack we need to look at a web-server (application-level) log that contains a record of all sessions that the server participates in, along with session-specific state information.

A cookie is a session-tracking mechanism issued by a web-application server to a web-client and store client-specific session information. Clients automatically include these cookies that can act as authentication tokens in their requests. In this example we assume that a session is identified by its IP address. An attack occurs when a malicious user hijacks a session by reusing an old cookie issued to a different IP address. The formula below asserts that a particular cookie must always be used by the same IP address.

$$\begin{array}{l}\underline{\text{min}}\ \texttt{SafeUse}(\underline{\text{string}}\ c, \underline{\text{int}}\ i) = ((\mathit{name} = c) \rightarrow (\mathit{ip} = i)) \wedge \odot\, \texttt{SafeUse}(c, i) \\ \underline{\text{mon}}\ \texttt{CookieSafe} = \texttt{Always}(\texttt{SafeUse}(\mathit{name}, \mathit{ip}))\end{array}$$

A trace that violates this formula therefore encodes a cookie-stealing attack. In the above example, the record schema of an event contains the *name* field which corresponds to name of the cookie, and the *ip* field that corresponds to the IP address of the client using the cookie. The parameterized rule `SafeUse` checks whether the association between a particular cookie identified by the cookie name and an IP address (specified as arguments) is the same as this association in the past. This example highlights the use of value-binding and the previous operator to describe history-based intrusion signatures.

Multi-domain Buffer Overflows

The next example illustrates how we can combine information about events from different logs in a cross-domain intrusion detection exercise. This scenario examines both web-server's access logs, as well as network logs to infer when a buffer-overflow attack has been attempted against the server. Network packets are analyzed, looking for binary data. Subsequently, the web-server's access logs are checked to see if a matching event can be found, where the web-server closes the connection successfully after receiving some binary data. If no matching log record is found, within a specific timeout, then the buffer overflow attack was successful and the web-server is now executing the attackers code. This example is specified by the formula shown next.

$$\begin{array}{l}\underline{\text{min}}\ \texttt{EventuallyClosed}(\underline{\text{long}}\ t, \underline{\text{long}}\ d, \underline{\text{long}}\ i1, \underline{\text{long}}\ i2) = (\mathit{time} - t < d) \wedge \\ \quad ((\mathit{ip1} = i1 \wedge \mathit{ip2} = i2 \wedge \mathit{log} = \mathit{web} \wedge \mathit{type} = \mathit{closed}) \vee \bigcirc \texttt{EventuallyClosed}(t, d, i1, i2)) \\ \underline{\text{mon}}\ \texttt{BufferSafe} = \texttt{Always}((\mathit{log} = \mathit{network} \wedge \mathit{type} = \mathit{binary}) \\ \qquad\qquad\qquad\qquad \rightarrow \texttt{EventuallyClosed}(\mathit{time}, 100, \mathit{ip1}, \mathit{ip2}))\end{array}$$

The record schema for an event contains the *log* field which indicates the name of the log to which the event belongs, the *type* field which can be *binary*, *closed*, etc., the *time* field representing the time at which the event occurred, the *ip1* field representing the source IP address, and the *ip2* representing the destination IP address. In the rule `EventuallyClosed`, the arguments t, d, $i1$, and $i2$ represent the time at which the rule is invoked, the timeout period, the source IP address, and the destination IP address, respectively. The rule `EventuallyClosed` asserts that eventually within time d the connection involving IP addresses $i1$ and $i2$ must get closed. Finally, the monitor `BufferSafe` states that it is always the case that if there is a event of binary access in the network log then eventually within time 100 there must be a matching event in the webserver log that denotes the closing of the connection. Here a connection is identified by the source and destination IP addresses.

Password guessing Attack

The next example illustrates the ability of EAGLE to collect statistics at runtime to detect a potential intrusion. In the password-guessing attack, an unauthorized user uses the *telnet* program to attempt to login into a machine over a network. If a user is allowed to guess an arbitrary number of passwords for a given user-name, it is only a matter of time before the password is broken. Most systems terminate a *telnet* session if a user makes more than three invalid password guesses over a short-time period. Some systems restrict the total number of invalid login attempts over the course of a longer time-period to prevent an attacker from succeeding by initiating multiple short sessions.

In order to detect this attack, we need to have access to the host's audit logs. On Solaris machines for example, auditing can be turned on by running Sun's Basic Security Monitoring (BSM) software. In order to encode this attack, we present a template that can be reused for any signature that specifies a threshold frequency of events of interest in a trace, which when exceeded constitutes an attack:

$$
\begin{array}{l}
\underline{\mathrm{max}}\ \texttt{Failure}() = (\mathit{type} = \mathit{login}) \wedge \neg \mathit{success} \\
\underline{\mathrm{max}}\ \texttt{Guess}(\underline{\mathrm{long}}\ i, \underline{\mathrm{Form}}\ F) = (\mathit{ip} = i) \wedge \texttt{Failure}() \\
\underline{\mathrm{max}}\ \texttt{Counter}(\underline{\mathrm{long}}\ t, \underline{\mathrm{long}}\ d, \underline{\mathrm{int}}\ c, \underline{\mathrm{long}}\ i, \underline{\mathrm{int}}\ C) = (\mathit{time} - t < d) \\
\qquad\qquad\qquad \rightarrow ((\texttt{Guess}(i) \rightarrow (c \leq C \wedge \bigcirc \texttt{Counter}(t, d, c+1, i, C))) \\
\qquad\qquad\qquad \wedge (\neg \texttt{Guess}(i) \rightarrow \texttt{Counter}(t, d, c, i, C))) \\
\underline{\mathrm{mon}}\ \texttt{PassGuessSafe} = \texttt{Always}(\texttt{Failure}() \rightarrow \texttt{Counter}(\mathit{time}, 300, 1, \mathit{ip}, 3))
\end{array}
$$

In the rule `Counter`, the arguments t, d, c, i, and C represent the rule invocation time, the timeout period, the current number of unsuccessful-guesses count, the source IP address doing the guess, and the threshold count. An attack occurs when the number-of-guess count c from the IP address i exceeds the threshold count C within the timeout period d. Whenever there is a `Failure()` in login, the parameterized rule `Counter` starts with the initial count set to 1. For every occurrence of an event, indicating login-failure from the same IP address within the timeout time d, the number-of-guess count is increased by one. The rule `Counter` also checks if within time d whether c exceeds C; in which case the whole rule becomes false indicating an attack. The monitor `PassGuessSafe` asserts that whenever there is a failure of login from an IP address then eventually within time 300 the number of login-failures from the same IP address must be less than or equal to 3.

Port-Sweep Attack

The Port-sweep attack is the most sophisticated example in this section. A port sweep is a surveillance scan through many ports on a single network host. Each port scan is essentially a message sent by the attacker to the victim's port and elicits a response from the victim that indicates the port's status. The aim of the attacker is to determine which services are supported on the host, and use this information to exploit vulnerabilities in these services in the future. Port scans may be legitimate, when a client is trying to determine if a service is being offered. However, when the number of port scans exceeds a certain threshold within a short time-period, the victim machine can assume that the scans are malicious. In order to detect this attack, once again, we need to include a time-period and a frequency explicitly in our formula, and can use the template from the previous example:

$$\begin{array}{l}
\underline{\text{max}}\ \texttt{NewPort}(\underline{\text{long}}\ i1, \underline{\text{long}}\ i2, \underline{\text{Set}}\ S) = (i1 = \mathit{ip1}) \wedge (i2 = \mathit{ip2}) \wedge (\mathit{port} \notin S) \\
\underline{\text{max}}\ \texttt{Counter}(\underline{\text{long}}\ t, \underline{\text{long}}\ d, \underline{\text{int}}\ c, \underline{\text{long}}\ i1, \underline{\text{long}}\ i2, \underline{\text{Set}}\ S, \underline{\text{int}}\ C) = (\mathit{time} - t < d) \rightarrow \\
\qquad ((\texttt{NewPort}(i1, i2, S) \rightarrow (c \leq C \wedge \bigcirc \texttt{Counter}(t, d, c+1, i1, i2, S \cup \{\mathit{port}\}, C))) \\
\qquad \wedge (\neg \texttt{NewPort}(i1, i2, S) \rightarrow \bigcirc \texttt{Counter}(t, d, c, i1, i2, S, C))) \\
\underline{\text{mon}}\ \texttt{PortScanSafe} = \texttt{Always}(\texttt{Counter}(\mathit{time}, 100, 1, \mathit{ip1}, \mathit{ip2}, \{\mathit{port}\}, 10))
\end{array}$$

As before, the arguments t, d, c and C in the `Counter` rule serve the same purpose as place holders for the initial time, the timeout period, the frequency count, and the threshold count. The parameterized `Counter` rule asserts that the number of port scans observed in a tcpdump record between a source and destination IP ($i1$ and $i2$) address pair never exceeds a certain threshold C within time d. Note that in the rule `Counter` we add every new port number scanned to the set S of all port numbers (involving $i1$ and $i2$) that are scanned within time d. The rule `NewPort` checks if the port number, involved in any communication between the IP addresses $i1$ and $i2$, exists in the set S of all port numbers (involved in all communications between $i1$ and $i2$) within the timeout period d. This example shows how we can use EAGLE to collect statistics by using a rich data-structure such as Set.

5 Implementation and Evaluation

5.1 MONID: Monitoring based Intrusion Detection Tool

The intrusion detection tool, MONID, is designed to operate in both online and offline fashion. In the online mode, MONID runs as a server that receives streams of events from various sources. To generate these events, the different logging modules are instrumented so that they filter and send *relevant* events to the server. On receiving various events, the server merges them in increasing order of timestamps and generates a single event-stream, which is passed through a filter to get a projection of the events that are required for monitoring. The filtered event stream is fed to the EAGLE FLIER to see if the event stream violates the normal behavior of the system specified as a set of EAGLE formulas. Note that EAGLE FLIER never stores the event stream while monitoring; instead it collects the essential facts and assimilates them into the monitored formulas by

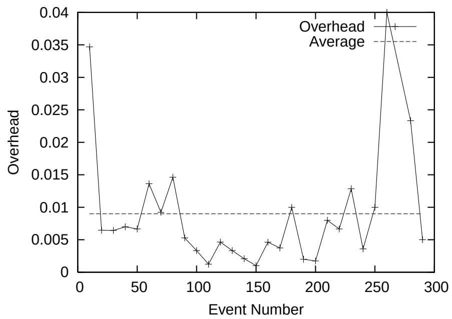

Fig. 2. Performance Overhead of Port-Sweep Attack

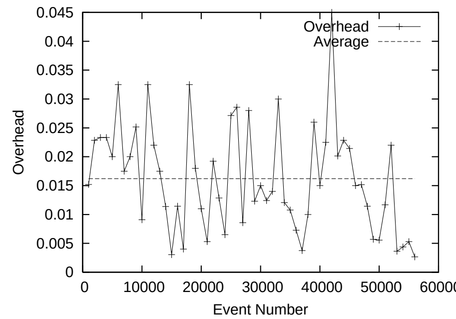

Fig. 3. Performance Overhead of Password-Guess Attack

transforming them into new formulas. This enables us to use EAGLE FLIER for online monitoring. In the offline mode, MONID reads various log files and sends an event corresponding each log entry to the server. The server then processes the event stream as before to detect intrusion. We perform most of our experiments in offline mode.

5.2 Evaluation

We test our MONID tool with the standard DARPA Intrusion Detection Evaluation data set [13] to study the overheads and explore the expressive power of our logic with real-world examples. In our experiments with MONID, we focus on the data sets provided in the 1998 offline Intrusion Detection Evaluation plan [13]. This data set focuses on UNIX workstations. The experimental setup simulates a military base with a private network marked as "Inside" connected to the "Outside" world through a Cisco AGS+ router. Two types of logs are available for analysis. The *tcpdump* logs were collected by running *tcpdump* on this connecting router. This data contains the contents of every packet transmitted between computers inside and outside of the military base, and gives us network sessions of complete TCP/IP connections. The sessions were designed to reflect (statistically) traffic seen on military bases and contain explicit examples of different real attacks.

The other data available is from the Sun Basic Security Module (BSM) from the host *pascal*, which was the victim of the simulated attacks, located on the "Inside" network. This data contains audit information describing system calls made to the Solaris kernel.

We implemented and tested our tool against the smurf, port-sweep and password-guessing as discussed in Section 4, of which we report the results of last two experiments due to space limitation. We do not repeat the details of these attacks here. We ran our tool on a 2.0 GHz Pentium M laptop with 1GB RAM, simulating the behavior of a dedicated-monitor that passively observes traffic on the "Inside" network and processes events from our victim host offline. The aim of our experiments is to demonstrate that the tool can detect intrusions, and the monitoring and processing overheads of our prototype tool are very low.

Our experiments detected 5 password-guess attack and 2 port-sweep attack in the logs. The performance overheads for monitoring the Port-Sweep and Password-Attacks are given in Figure 2 and 3, respectively. The X-axis in both graphs shows the number of events we are monitoring, as they are obtained from our logs. Each data point is the average overhead calculated for intervals of 10 and 1000 events respectively. The Y-axis plots the ratio between the time spent by the monitor vs. the time between the generation of the events in the actual log. As long as this ratio is less than 1, our monitoring is feasible. The results, show that the average overhead, is around 0.009 for Port-Sweep attacks and 0.016 for Password-Guessing attacks, suggesting that online mode is feasible and efficient.

6 Conclusion

We have proposed a framework for intrusion detection using a temporal logic approach. In contrast to other such temporal logic based approaches [16], we use an expressively rich logic in which one can express both signature based and simple anamoly based attack specifications. We automatically monitor such attack specifications with respect to the system execution. An intrusion is detected when the observed execution violates the formula being monitored. We demonstrate this approach by specifying formulas for detecting several types of well-known attacks, and by testing a prototype implementation of our monitoring algorithm over large event-logs made available by DARPA for evaluation purposes. We believe that our examples are generic and can be used as template for specifying a large number of other attacks.

Our approach opens up several interesting directions for future research. We plan to conduct a more systematic performance study and categorize the overheads more precisely. With the aim to exploit the expressiveness of our logic, we plan explore the use of ideas introduced in [18, 19] for *predicting* security failures from successful executions in multi-threaded programs. Specifically, in addition to monitoring a given specification against the currently observed trace, we can also compare the specification with all the traces that correspond to different interleavings of the same partial order (causality relation) between the underlying events. Another problem of interest is to use the distributed monitoring framework introduced in [20] for detecting attacks that involve multiple hosts on a network. We believe that our approach can complement existing intrusion detection mechanisms and provide support for more expressive attack specifications.

References

[1] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion detection expert system. Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1995.

[2] S. Axelsson. Intrusion detection systems: A taxonomy and survey. Technical Report 99–15, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, 2000.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'04) (Satellite workshop of IPDPS'04)*, Santa Fe, New Mexico, USA, April 2004. (To Appear).

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57, Venice, Italy, January 2004. Springer-Verlag.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *IEEE Computer Society Symposium on Research on Security and Privacy*, pages 240–250, May 1992.

[7] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion detection systems. *Computer Networks*, 31(8):805–822, April 1999.

[8] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.

[9] J.P. Ignizio. *Introduction to Expert Systems-the Development and Implementation of Rule-Based Expert System*. McGraw-Hill Science, 1991.

[10] K. Ilgun, R. Kemmerer, and P. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[11] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.

[12] S. Kumar and E. Spafford. A pattern matching model for misuse intrusion detection. In *National Computer Security Conference*, pages 11–21, 1994.

[13] MIT Lincoln Laboratory. DARPA intrusion detection evaluation. http://www.ll.mit.edu/IST/ideval/.

[14] W. Lee. A datamining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, pages 120–132, May 1999.

[15] P. Porras and P. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, 1997.

[16] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*. IEEE, 2001.

[17] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert systems in intrusion detection: A case study. In *National Computer Security Conference*, pages 74–81, 1998.

[18] K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 337–346, Helsinki, Finland, September 2003. ACM.

[19] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 123–138, Barcelona, Spain, March 2004.

[20] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 418–427, Edinburgh, UK, May 2004. IEEE.

[21] H. Teng, K. Chen, and S. Lu. Security audit trail analysis using inductively generated predictive rules. In *Conference on Artificial Intelligence Applications*, pages 24–29. IEEE Computer Society Press, March 1990.

[22] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.