A Temporal Logic Based Framework for Intrusion Detection

Zanolin Lorenzo

September 20, 2023

Abstract

The purpose of this paper is to introduce MONID, which is a framework created for system intrusion detection. This framework uses EAGLE, a rich and effectively monitorable logic, to express intrusion patterns using temporal logic formulas; EAGLE's ability to include data values and parameterized recursive equations makes it possible to represent security threats that include complex temporal event sequences and attacks with intrinsically statistical signatures succinctly. This tool can be used in offline and real-time scenarios. The implementation uses an algorithm for online monitoring that matches descriptions of the lack of an assault with indications of system execution; an alarm is set off whenever the standard is broken.

Contents

1 Introduction **2**

2 EAGLE, a Temporal Monitoring Logic **2**

2.1 Basics 3

2.2 Syntax and Semantics 4

2.3 Evaluation algorithm 5

2.4 Relationship to LTL+P 8

3 MONID: Structure analysis **8**

4 Example of attacks **9**

5 Conclusions **13**

1 Introduction

Even with all of the advances in computer security research, totally safe computer systems remain a long way off. Almost every large and complicated computer system nowadays contains vulnerabilities. Intrusion detection means maintaining constant surveillance on a system in order to detect any misuse of these weak areas as soon as feasible so that they can be repaired.

According to the literature, there are three approaches to intrusion detection systems: *signature-based*[4], *anomaly-based*[6] and *hybrid*[5]. The first approach aims to identify patterns and match them with known signs of intrusions, relying on a database of previous intrusions. If activity within the network matches the "signature" of an attack or breach in the database, the detection system creates an alert. This 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, *anomaly-based* is capable of detecting new attacks since an alarm is raised if an observed behavior deviates significantly from prelearned normal behavior. Finally, a *hybrid system* combines the best of both worlds by looking at patterns and one-off events. A Hybrid Intrusion Detection system can flag new and existing intrusion strategies.

In this paper we will focus on *signature-based* approach using temporal logic; we use EAGLE[3, 1] to specify a system's attack-safe behavior. EAGLE allows recursively built temporal formulas, parameterizable by both logical formulas and data expressions, across three primitive modalities: "next", "previous", and "concatenation". The logic allows us to describe temporal patterns involving reasoning about data-values observed in individual events. Unlike LTL, EAGLE allows us to design attacks with fundamentally statistical characteristics; password guessing attacks and ICMP-flood denial of service attacks are two examples. We will monitor EAGLE formulas using an online algorithm that processes each event as it happens and updates the monitored formula to store a relevant summary. An intrusion alarm is triggered if, for any reason, the updated formula turns out to be false; as a result, the entire process operates in real-time. We'll see how the implementation, MONID, examines the event stream to see if the monitored formula (which is used to hold the pertinent summary of the system) is broken.

In Section 2, we'll deeply explore EAGLE, covering everything from its syntax to semantics and the evaluation algorithm, with the addition of practical examples. Moving on to Section 3, we'll delve into the MONID framework, which employs EAGLE to construct an intrusion detection system. Section 4 will focus on specific EAGLE rules tailored for the detection of various types of attacks and finally, in Section 5, we will draw our conclusions.

2 EAGLE, a Temporal Monitoring Logic

According to [2], EAGLE offers a succinct but powerful set of primitives, supporting recursive parameterized equations with a minimal/maximal fix-point

semantics together with three temporal operators: next-time ($\bigcirc$), previous-time ($\odot$), and concatenation ($\cdot$).

2.1 Basics

In EAGLE recursion definitions are supported; for example, in the current framework we can build the following definitions:

$$\underline{\text{min}}\ \texttt{Next}(\underline{\text{Form}}\ F) = \bigcirc F$$
$$\underline{\text{max}}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc \texttt{Always}(F)$$
$$\underline{\text{min}}\ \texttt{Eventually}(\underline{\text{Form}}\ F) = F \vee \bigcirc \texttt{Eventually}(F)$$
$$\underline{\text{min}}\ \texttt{Until}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigcirc \texttt{Until}(F_1, F_2))$$

As we can see, rules are parameterized by an EAGLE formula (of type Form), which means that we will be able to write EAGLE formulas such as $\texttt{Always}(\texttt{Eventually}(x > 0))$. Also, the `Always` operator is defined as the maximal solution of the equation $X = F \wedge \bigcirc X$, while the `Eventually` operator represents the minimal solution to the equation $X = F \vee \bigcirc X$.

Assume that we want to state the following property: "*Whenever there is a login by any user x, then eventually the user x logs out*". In LTL we can write the following formula:

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

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 $(action = \texttt{login})$ 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 can express the same property with the following rules:

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

As a result, rules in EAGLE give us the power to create specific temporal operators as well as to bind and modify data. This property turns out to be crucial for succinctly expressing executions of attack-safe systems. As we can see, each rule begins with a term that describes its type (min, max, mon); we will read more about it later.

Lastly, two assumptions must be made:

1. There is a finite sequence of events called $\sigma = \alpha_1, \ldots, \alpha_n$ that is a merge of the system registered logs organized by ascending time. The structure of an event record α_i is the following:

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

   An example of event could be: $\{userId : \text{"}\texttt{Lori}\text{"},\ action : \texttt{login},\ time : \texttt{20}\}$

2. For each attack, there is a formula F that specifies the absence of it.

To make the paper self-contained, we will present the syntax and semantics of EAGLE.

2.2 Syntax and Semantics

Syntax. A specification S is made up of an observer's part O and a declaration's part D. O is made up of zero or more monitor definitions M, which define what will be watched, while D is made up of zero or more rule definitions R. The names of rules and monitors are (N).

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

The types T_i of parameters include primitive types like $\underline{\text{int}}$, $\underline{\text{long}}$, $\underline{\text{float}}$, etc. or formulas of type $\underline{\text{Form}}$. Definitions starting with the keyword $\underline{\text{mon}}$ specify the EAGLE formulas to be monitored and cannot have a recursive definition. We will see that these kinds of rules evolve as new events occur. A term indicating whether the interpretation is maximal ($\underline{\text{max}}$) or minimal ($\underline{\text{min}}$) comes before the rule specification R. While minimal rules establish *liveness properties* (something good eventually happens), maximal rules define *safety properties* (nothing bad ever happens); we shall see that the difference becomes important only when we are evaluating the at the boundaries of a trace. Finally, recursive definitions of rules are permitted as long as they are tightly guarded by a temporal operator.

Semantics. The semantics of the logic is defined in terms of the *satisfaction* relation $\models$ which defines whether a finite execution trace σ satisfies the specification $\varphi = D\,O$. The i'th state 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:

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

In other words, a trace satisfies a specification if it fulfills each monitored formula ($\underline{\text{mon}}$) when monitoring from position 1, which is the initial state. The definition of the satisfaction relation $\models_D \subseteq (Trace \times \mathbb{N}) \times \underline{\text{Form}}$, for a set of rule definitions

D, is the following:

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

where $\overline{F}$ and $\overline{P}$ represent tuples of type $\overline{\text{Form}}$ and $\overline{T}$ respectively.

An **important** note is that, given a trace $\sigma = \alpha_1, \ldots, \alpha_n$ the index i of the trace can become 0 (before the first state) and $n+1$, thus going beyond the limits; both of these situations result in rule applications evaluating to either true if maximal or false if minimal, without first taking the rules body into account. Let us explain why using an example. Given the definition

$$\underline{\text{min}}\ \texttt{Property}(\underline{\text{Form}}\ F) = F \vee \bigcirc \texttt{Property}(F)$$

and the sequence $\sigma = \alpha_1, \ldots, \alpha_n$, `Property` will evaluate to *true*

$$evaluate(\ldots evaluate(evaluate(\texttt{Property}, \alpha_1), \alpha_2), \ldots, \alpha_n) = true$$

if and only if the rule is true at some given event α_i. Once the sequence has been completely analyzed, we will obtain a big disjunction (in the case of a $\underline{\text{min}}$) or a big conjunction (in the case of a $\underline{\text{max}}$) and the final value will close the evaluation. Continuing the example, the extended formula will be:

$$F \vee \bigcirc(F \vee \bigcirc(\ldots F \vee \bigcirc \mathbf{Property}(\mathbf{F})))$$

and **Property**(**F**) will be evaluated as false, since it is the last of a big disjunction.

2.3 Evaluation algorithm

As already mentioned, monitored formulas ($\underline{\text{mon}}$) evolve at every event to store relevant information about past events. The evolved formula's value is deter-

mined at the conclusion of the event series; if it is true, the formula is satisfied by the event sequence; otherwise, it is violated. Formally, a formula $F' = \mathit{evaluate}(F, \alpha_i)$ is created when a formula F is evaluated at an event $\alpha_i = \sigma(i)$ and has the condition that $\sigma, i \models F$ if and only if $\sigma, i+1 \models F'$. With F being the evolved formula, we compute the boolean function $\mathit{value}(F)$ at the conclusion of the trace. The function $\mathit{evaluate} : \underline{\text{Form}} \times \underline{\text{State}} \to \underline{\text{Form}}$ uses an auxiliary function $\mathit{update} : \underline{\text{Form}} \times \underline{\text{State}} \to \underline{\text{Form}}$ to pre-evaluate a formula if it is guarded by the previous operator $\bigodot$. Note that Form in the practical use represents the monitored formula, while State represents the current event. At the end (or at the beginning) of a trace, the function $\mathit{value} : \underline{\text{Form}} \to \{\underline{\text{true}}, \underline{\text{false}}\}$, when applied to F, returns true if and only if $\sigma, |\sigma| + 1 \models F$ (or $\sigma, 0 \models F$), and returns false otherwise.

Calculus. According to [3], the *evaluate*, *update* and *value* functions are defined a priori for all operators except for the rule application, which gets generated based on the definition of rules in the specification. The definitions of these functions on the different primitive operators (except the previous operator $\odot$) are the following (*op* can be $\vee, \wedge, \to$):

$$
\begin{aligned}
\mathit{evaluate}(\underline{\text{true}}, \alpha_i) &= \underline{\text{true}} \\
\mathit{evaluate}(\underline{\text{false}}, \alpha_i) &= \underline{\text{false}} \\
\mathit{evaluate}(\mathit{exp}, \alpha_i) &= \text{value of } \mathit{exp} \text{ in } \alpha_i \\
\mathit{evaluate}(F_1 \ \mathit{op}\ F_2, \alpha_i) &= \mathit{evaluate}(F_1, \alpha_i)\ \mathit{op}\ \mathit{evaluate}(F_2, \alpha_i) \\
\mathit{evaluate}(\neg F, \alpha_i) &= \neg \mathit{evaluate}(F, \alpha_i) \\
\mathit{evaluate}(\bigcirc F, \alpha_i) &= \mathit{update}(F, \alpha_i) \\
\mathit{evaluate}(F_1 \cdot F_2, \alpha_i) &= \begin{cases} \mathit{evaluate}(F_1, \alpha_i) \cdot F_2 & \text{if } \mathit{value}(F_1) = \underline{\text{false}} \\ \mathit{evaluate}(F_1, \alpha_i) \cdot F_2 \vee \mathit{evaluate}(F_2, \alpha_i) & \text{else} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{value}(\underline{\text{true}}) &= \underline{\text{true}} \\
\mathit{value}(\underline{\text{false}}) &= \underline{\text{false}} \\
\mathit{value}(\text{exp}) &= \underline{\text{false}} \\
\mathit{value}(F_1 \text{ op } F_2) &= \mathit{value}(F_1) \text{ op } \mathit{value}(F_2) \\
\mathit{value}(\neg F) &= \neg \mathit{value}(F) \\
\mathit{value}(\bigcirc F) &= \begin{cases} F & \text{if at the beginning of a trace} \\ \underline{\text{false}} & \text{if at the end of a trace} \end{cases} \\
\mathit{value}(F_1 \cdot F_2, \alpha_i) &= \mathit{value}(F_1) \wedge \mathit{value}(F_2) \\
\mathit{value}(\mathtt{R}(F_1, \ldots, F_n)) &= \begin{cases} \underline{\text{true}} & \text{if } \mathtt{R} \text{ is } \underline{\text{max}} \\ \underline{\text{false}} & \text{if } \mathtt{R} \text{ is } \underline{\text{min}} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{update}(\underline{\text{true}}, \alpha_i) &= \underline{\text{true}} \\
\mathit{update}(\underline{\text{false}}, \alpha_i) &= \underline{\text{false}}
\end{aligned}
$$

$$
\begin{aligned}
update(\text{exp}, \alpha_i) &= \text{exp} \\
update(F_1 \text{ op } F_2, \alpha_i) &= update(F_1, \alpha_i) \text{ op } update(F_2, \alpha_i) \\
update(\neg F, \alpha_i) &= \neg update(F, \alpha_i) \\
update(\bigcirc F, \alpha_i) &= update(F, \alpha_i) \\
update(F_1 \cdot F_2, \alpha_i) &= update(F_1, \alpha_i) \cdot F_2
\end{aligned}
$$

Focusing of *Future Time Operators*, we have:

$$
\begin{aligned}
evaluate(\texttt{Next}(F), s) &= evaluate(\bigcirc F, s) \\
evaluate(\texttt{Always}(F), s) &= evaluate(F \wedge \bigcirc \texttt{Always}(F), s) \\
evaluate(\texttt{Eventually}(F), s) &= evaluate(F \vee \bigcirc \texttt{Eventually}(F), s) \\
evaluate(\texttt{Until}(F_1, F_2), s) &= evaluate(F_2 \vee (F_1 \wedge \bigcirc \texttt{Until}(F_1, F_2)), s)
\end{aligned}
$$

$$
\begin{aligned}
update(\texttt{Next}(F), s) &= \texttt{Next}(update(F, s)) \\
update(\texttt{Always}(F), s) &= \texttt{Always}(update(F, s)) \\
update(\texttt{Eventually}(F), s) &= \texttt{Eventually}(update(F, s)) \\
update(\texttt{Until}(F_1, F_2), s) &= \texttt{Until}(update(F_1, s), update(F_2, s))
\end{aligned}
$$

while, for the *Past Time Operators* the $\odot$ operator is removed and instead a rule `Previous` is introduced; more details can be found in [2].

Let us introduce an example to better understand the algorithm. Given the following rules

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

and a trace $\sigma = \alpha_1, \alpha_2$, where $\alpha_1 = \{userId : \text{"}\texttt{Lori}\text{"},\ action : \texttt{login},\ time : \texttt{17.0}\}$ and $\alpha_2 = \{userId : \text{"}\texttt{Lori}\text{"},\ action : \texttt{logout},\ time : \texttt{150.0}\}$, the initial formula M_3 gets modified. At time 1, when α_1 is analyzed, M_3 changes as follows:

$$
\begin{aligned}
evaluate(M_3, \alpha_1) = {} & \texttt{EvTimedLogout}(\text{"}\texttt{Lori}\text{"}, \texttt{17.0}, \texttt{100}) \\
& \wedge \texttt{Always}((action = \texttt{login}) \rightarrow \texttt{EvTimedLogout}(userId, time, 100))
\end{aligned}
$$

At the second event the predicate $(time - t \leq \delta)$ gets instantiated to $(150.0 - 17.0 \leq 100)$ which is false; thus, the whole formula becomes false, indicating that the two-event trace violates the property M_3.

It's essential to emphasize that the logic EAGLE operates with a *finite* trace semantics; the monitoring algorithm, as written before, can check whether the formula is satisfied or not only at the conclusion of a trace. However, in intrusion detection scenarios, the notion of the "end of a trace" makes no sense since event

sequences can theoretically be infinite. In such cases, the aim is to trigger an alarm as soon as a property is violated and this is done by continuously checking the formula's satisfaction status after each event. In practice, once we have the specification of the attack pattern in terms of φ, we can express the system's safe behavior as $\Box(\neg\varphi)$. Note that if a sequence of events corresponding to the attack is discovered, this formula becomes false.

2.4 Relationship to LTL+P

According to [3, 1], we can define both **Future Time LTL** and **Past Time LTL** operators. The future modalities are written in section 2.1, therefore let us list the past ones:

$$\underline{\min}\ \texttt{Previous}(\underline{\text{Form}}\ F) = \odot F$$
$$\underline{\max}\ \texttt{AlwaysInPast}(\underline{\text{Form}}\ F) = F \wedge \odot\texttt{AlwaysInPast}(F)$$
$$\underline{\min}\ \texttt{EventuallyInPast}(\underline{\text{Form}}\ F) = F \vee \odot\texttt{EventuallyInPast}(F)$$
$$\underline{\min}\ \texttt{Since}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \odot\texttt{Since}(F_1, F_2))$$

By combining the definitions for the future and past time LTL as defined above, we obtain a temporal logic over the future, present and past, in which one can freely intermix the future and past time modalities; thus, we can state that EAGLE is expressive at least as LTL+P.

3 MONID: Structure analysis

MONID is a prototype that can detect intrusions either online or offline. System-level event data is delivered to MONID server either directly from relevant log files (offline) or online via application code that has been properly instrumented. In the online mode, MONID runs as a server that receives streams of events from various sources and to generate these events, the different logging modules are instrumented so that they filter and send relevant events to the server. To create a single event trace, the server combines events from numerous sources according to timestamps and preprocesses them into an abstract intermediate form. Note that sometimes it is necessary to watch events from multiple sources in order to detect certain attacks; then, the MONID monitor checks this event trail against a predetermined criteria, and if the specification is broken, an intrusion alarm is raised. While in offline mode, MONID reads various log files and sends an event corresponding to each log entry to the server. The server then processes the event stream as before to detect an intrusion.

The entire framework is represented in Figure 1. In paper [9], the implementation was written in Java and available as a library; there were two available methods used. The first one is *parse* which takes a file containing a list of EAGLE specifications and monitor formulas and compiles them into a data structure used to represent the *monitor* represented in Figure 1. The client software iteratively invokes the method *evaluate* for each event after compilation; this

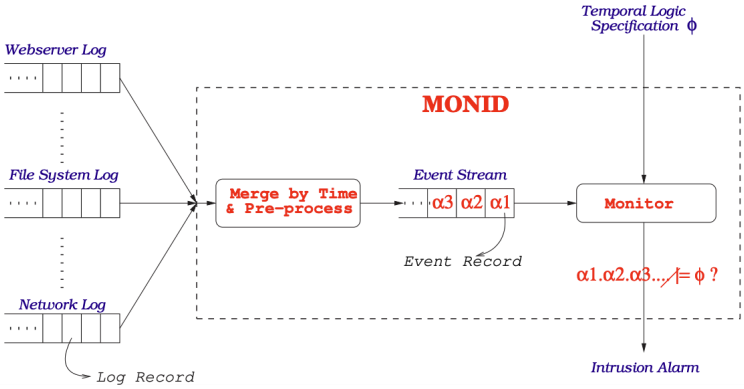

Figure 1: MONID Architecture.

call internally adjusts the monitor formulas in accordance with Subsection 2.3. An error notice is produced or a pre-specified mechanism is triggered to take a corrective action if the monitored formulas become false at any stage.

Some evaluations were also done in [9], using the DARPA Intrusion Detection Evaluation dataset [8]. The experiment was done using offline mode against some of the attacks presented in Section 4. More specifically, against the smurf, port-sweep and password-guessing attacks, the experiments detected 5 password-guess attack and 2 port-sweep attack in the logs and the performance overheads for monitoring the Port-Sweep and Password-Attacks are reported 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. Although the used hardware is now outdated and obsolete, the overhead was rather modest, thus this method should still be theoretically helpful today.

4 Example of attacks

In this section, we give some examples of how EAGLE can be utilized to build formulas that correlate to the desired execution trace characteristics of a system under observation. In this context, a trace σ that violates this specification is referred to as an intrusion or an attack; to explain how the framework may be used, we use examples from actual attacks illustrated in [9]. These illustrations demonstrate the expressive power of EAGLE formalism as well as its numerous features. In all the following examples, the rule `Always` is the one defined in

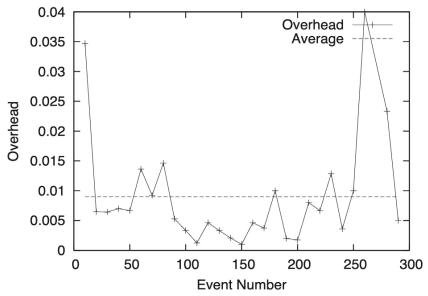

Figure 2: Performances overhead of Port-Sweep attacks.

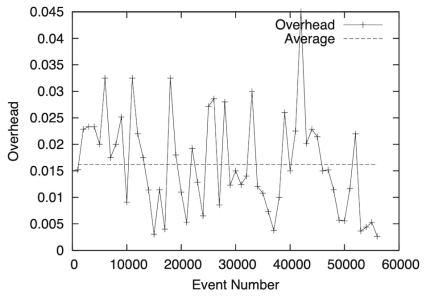

Figure 3: Performances overhead of Password-Guessing attacks.

Section 2.1.

Smurf attacks. A smurf attack [11] is a type of Distributed Denial of Service (DDoS) attack that targets computer networks; in this scenario, the attacker sends a large volume of Internet Control Message Protocol (ICMP) echo request packets (commonly known as "ping" packets) to a network's broadcast address. This broadcast address is the one that reaches all devices on the targeted network. The attacker spoofs the source IP address of these ICMP packets to make it appear as if they are originating from the victim's IP address and since the broadcast address sends the request to all devices on the network, all devices will respond with ICMP echo replies to the victim's IP address. This results in a flood of traffic overwhelming the victim's network and causing it to become unreachable or severely degraded in performance. To detect this kind of attack, we need to work on network logs; we will use a network tool called *tcpdump*. The monitored formula that describes the absence of the attack is the following:

$$\underline{\max}\ \texttt{SmurfAttack}() = (type = \text{"ICMP"}) \wedge isBroadcast(ip)$$
$$\underline{\text{mon}}\ \texttt{SmurfSafety} = \ \texttt{Always}(\neg\ \texttt{SmurfAttack}())$$

as we can see, the *type* field of a record schema describes the type of network packet, while the *ip* field is the destination of the packet. The `SmurfAttack` rule determines whether the packet is a ping packet and if the destination's IP address is a broadcast ip, i.e. all the host bits are set to 1. Thus, *"Always there is no attack"* can be used to describe how the system behaves well in response to this attack.

Cookie-stealing attacks. A cookie is a technique for tracking sessions that a web application server sends to a web client and uses to keep track of client-specific session data. These cookies that serve as authentication tokens are automatically included in requests by clients and in this scenario we assume that a session is exclusively identified by the client's IP address. When a malicious user uses an old cookie provided by a different IP address to take over a session,

it is considered an 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. According to the formula below, a specific cookie must always be used by the same IP address.

$$\underline{\text{min}}\ \texttt{Hijack}(\underline{\text{string}}\ c, \underline{\text{string}}\ i) = ((name = c) \wedge \neg(ip = i)) \vee \odot\ \texttt{Hijack}(c, i)$$
$$\underline{\text{mon}}\ \texttt{CookieSafe} = \ \texttt{Always}(\neg\texttt{Hijack}(name, ip))$$

A trace that violates this formula therefore encodes a cookie-stealing attack. The rule `Hijack` checks whether it occurs a situation in which we have the same cookie c associated with two different ip i; if it's true, then it means that a cookie has been stolen and reused by an attacker. With `CookieSafe` we want to check that it never occurs a situation just like the one described.

Multi-domain Buffer Overflows attacks. According to [10], a buffer-overflow attack is an attack that (possibly implicitly) uses memory-manipulating operations to overflow a buffer which results in the modification of an address to point to malicious or unexpected code. In this case, we will integrate data from various log sources; to be more exact, we will examine both network and web server access logs to determine when a server buffer overflow attack has been performed. We search through network packets for binary data; then the web server's access logs are examined to see if a corresponding event, in which the web server correctly terminates the connection after receiving some binary data, can be located. If no matching log record is discovered within a predetermined timeout, it means that the buffer overflow attack was successful. The following formulas capture this scenario:

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

In each record we have the *log* field which indicates the source of the log, the *time* represents the instant in which the event occurs, $ip1$ represents the source address, $ip2$ represents the destination IP address and finally the *type* field can be $binary, closed, \ldots$ We assume that a connection is identified by the tuple $(ip1, ip2)$. In order, the `EventuallyClosed` rule states that the connection between IP addresses $i1$ and $i2$ must end eventually within time d; finally, the monitor rule `BufferSafe` states that it is always the case that if there is an 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.

Password guessing attacks. Assuming the system accepts telnet connections to remote login; it won't take long until the password is compromised if an attacker is permitted to guess any number of passwords for a specific user name. If a user guesses an incorrect password more than three times in a short

amount of time, most systems end the telnet session. To prevent an attacker from succeeding by starting numerous temporary sessions, some systems cap the total number of unsuccessful login attempts over a longer time frame; in this case we need to have access to the host's audit logs. The formulas that capture this scenario are the following:

$$\underline{\max}\ \texttt{Failure}() = (type = login) \wedge \neg success$$

$$\underline{\max}\ \texttt{Guess}(\underline{\text{string}}\ i) = (ip = i) \wedge \texttt{Failure}()$$

$$\begin{aligned}\underline{\max}\ \texttt{Counter}(&\underline{\text{long}}\ t, \underline{\text{long}}\ d, \underline{\text{int}}\ c, \underline{\text{string}}\ i, \underline{\text{int}}\ C) = (time - t < d)\\ &\rightarrow ((\texttt{Guess}(i) \rightarrow (c \leq C \wedge \bigcirc\texttt{Counter}(t, d, c+1, i, C)))\\ &\wedge (\neg\texttt{Guess}(i) \rightarrow \texttt{Counter}(t, d, c, i, C)))\end{aligned}$$

$$\underline{\text{mon}}\ \texttt{PassGuessSafe} = \texttt{Always}(\texttt{Failure}() \rightarrow \texttt{Counter}(time, 300, 1, ip, 3))$$

as we can see, the rules here are numerous. In `Counter`, the arguments t, d, c, i, 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. When the number of guesses from the IP address i exceeds the threshold count C within the timeout period d, an attack has taken place. The parameterized rule `Counter` starts with the initial count c set to 1 whenever there is a `Failure` during login. The number of guesses is increased by one for each instance of an event signifying a failed login attempt from the same IP address within the timeout period d. The rule `Counter` also determines whether the local counter c surpasses the threshold C within time d, and if it does, the entire rule becomes false, indicating an attack. Finally, with the rule `PassGuessSafe` we assure that whenever there is a failed login from a specific IP address, we make sure that within 300 seconds the number of login-failures from the same IP address must be less than or equal to 3.

Port Scanning attacks. According to [7], Port scanning attack is a method for discovering exploitable communication channels that has been used for a long time. The key idea is to probe the network ports and then keep the information about them that is useful for an attack. Hackers frequently employ a port scan approach to find gaps or weak spots in a network. Attackers can use a port scan attack to identify open ports and determine if they are accepting or rejecting data; additionally, it can show whether a company uses firewalls or other active security measures. Assuming that our server has multiple network interfaces, thus multiple IP addresses; the victim machine can presume that port scans are malicious when the counter of reached ports overcomes a specific threshold in a short amount of time. The formulas are the following:

$$\underline{\max}\ \texttt{NewPort}(\underline{\text{string}}\ i1, \underline{\text{string}}\ i2, \underline{\text{Set}}\ S) = (i1 = ip1) \wedge (i2 = ip2) \wedge (port \notin S)$$

$$\begin{aligned}\underline{\max}\ \texttt{Counter}(&\underline{\text{long}}\ t, \underline{\text{long}}\ d, \underline{\text{int}}\ c, \underline{\text{string}}\ i1, \underline{\text{string}}\ i2, \underline{\text{Set}}\ S, \underline{\text{int}}\ C) = (time - t < d)\\ &\rightarrow ((\texttt{NewPort}(i1, i2, S) \rightarrow (c \leq C \wedge \bigcirc\texttt{Counter}(t, d, c+1, i1, i2, S \cup \{port\}, C)))\\ &\wedge (\neg\texttt{NewPort}(i1, i2, S) \rightarrow \texttt{Counter}(t, d, c, i1, i2, S, C)))\end{aligned}$$

$$\underline{\text{mon}}\ \texttt{PortScanSafe} = \texttt{Always}(\texttt{Counter}(time, 100, 1, ip1, ip2, \{port\}, 10))$$

in this case we have two IPs; $ip1$ is the source IP, while $ip2$ is the destination one. The parameterized `Counter` rule states that the number of port scans between a source and destination IP address pair $(i1, i2)$ recorded does not surpass a threshold C 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.

5 Conclusions

In conclusion, we have introduced a new temporal logic named EAGLE, which serves as the foundation for the MONID framework, which is a useful tool for system intrusion detection. MONID continuously monitors these attack specifications during system execution and when the observed execution contradicts the monitored formula, we detect an intrusion. We have showcased this by creating formulas to identify several common attacks and testing our monitoring algorithm on large event logs provided by DARPA. These examples can serve as templates for specifying many other types of attacks. The obtained results were excellent, both in terms of detected attacks and system overhead.

References

[1] Howard Barringer et al. "EAGLE can do Efficient LTL Monitoring". In: *Fossacs Ortacas*. 2003.

[2] Howard Barringer et al. "Program monitoring with LTL in EAGLE". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE. 2004, p. 264.

[3] Howard Barringer et al. "Rule-based runtime verification". In: *Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004 Venice, Italy, January 11-13, 2004 Proceedings 5.* Springer. 2004, pp. 44–57.

[4] Wei Gao and Thomas H Morris. "On cyber attacks and signature based intrusion detection for modbus based industrial control systems". In: *Journal of Digital Forensics, Security and Law* 9.1 (2014), p. 3.

[5] Akash Garg and Prachi Maheshwari. "A hybrid intrusion detection system: A review". In: *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. IEEE. 2016, pp. 1–5.

[6] VVRPV Jyothsna, Rama Prasad, and K Munivara Prasad. "A review of anomaly based intrusion detection systems". In: *International Journal of Computer Applications* 28.7 (2011), pp. 26–35.

[7] Urupoj Kanlayasiri, Surasak Sanguanpong, and Wipa Jaratmanachot. "A rule-based approach for port scanning detection". In: *Proceedings of the 23rd electrical engineering conference, Chiang Mai Thailand*. Citeseer. 2000, pp. 485–488.

[8] John McHugh. "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory". In: *ACM Transactions on Information and System Security (TISSEC)* 3.4 (2000), pp. 262–294.

[9] Prasad Naldurg, Koushik Sen, and Prasanna Thati. "A temporal logic based framework for intrusion detection". In: *Formal Techniques for Networked and Distributed Systems–FORTE 2004: 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004. Proceedings 24*. Springer. 2004, pp. 359–376.

[10] Krerk Piromsopa and Richard J Enbody. "Buffer-overflow protection: the theory". In: *2006 IEEE International Conference on Electro/Information Technology*. IEEE. 2006, pp. 454–458.

[11] Gholam Reza Zargar and Peyman Kabiri. "Identification of effective network features to detect Smurf attacks". In: *2009 IEEE Student Conference on Research and Development (SCOReD)*. IEEE. 2009, pp. 49–52.