A Temporal Logic Based Framework for Intrusion Detection

Zanolin Lorenzo

September 4, 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

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.

There are three Intrusion Detection System approaches, according to the literature: *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 from 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 to detect new attacks since an alarm is raised if an observed behavior deviates significantly from pre-learned 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 allowa us to design attacks with fundamentally statistical characteristics; password guessing attacks and ICMP-flood denial of service attacks are two examples. 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 being broken.

In Section 2 we will analyze the basics of EAGLE. In Section 3 we will analyze MONID framework and some evalutations on the performances. In Section 4 we will see some attacks specifications. In section 5 we will conclude.

2 EAGLE, a Temporal Monitoring Logic

According to [2], EAGLE offers a succinct but powerful set of primitives, essentially 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; in the current framework we can build the following definitions:

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

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 Always(Eventually($x > 0$)). Also, the Always operator is defined as 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:

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

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 = 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:

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

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:

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

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

2. For each attack, there is a formula F which 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 part O and a declaration 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 keyword $\underline{\text{mon}}$ specifies the EAGLE formulas to be monitored and cannot have a recursive definition. We will see that these kind of rules will evolve as new events appear. A term indicating whether the interpretation is maximal ($\underline{\text{max}}$) or minimal ($\underline{\text{min}}$) comes before a 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 satisfy 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:

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

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

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

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

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

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 case of a $\underline{\text{min}}$) or a big conjunction (in case of a $\underline{\text{max}}$) and the final value will close the evalutation. Continuing the example, the extended formula will be:

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

and the bold **Property(F)** will be evaluated to 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 informations about past istants. The evolved formula's value is determined 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 $F' = evaluate(F, \alpha_i)$ 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 $value(F)$ at the conclusion of the trace. The definition of the function $evaluate : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$ uses an auxiliary function $update : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$ to pre-evaluate a formula if it is guarded by the previous operator $\bigodot$.

The definitions of *evaluate*, *update* and *value* on the different primitive operators are the following:

$$
\begin{aligned}
evaluate(\underline{\text{true}}, \alpha_i) &= \underline{\text{true}} \\
evaluate(\underline{\text{false}}, \alpha_i) &= \underline{\text{false}} \\
evaluate(\text{exp}, \alpha_i) &= \text{value of } exp \text{ in } \alpha_i \\
evaluate(F_1 \ op \ F_2, \alpha_i) &= evaluate(F_1, \alpha_i) \ op \ evaluate(F_2, \alpha_i) \quad \text{where op can be } \vee, \wedge, \rightarrow \\
evaluate(\neg F, \alpha_i) &= \neg evaluate(F, \alpha_i) \\
evaluate(\bigcirc F, \alpha_i) &= update(F, \alpha_i)
\end{aligned}
$$

$$
evaluate(F_1 \cdot F_2, \alpha_i) = \begin{cases} \text{if } value(F_1) = \underline{\text{false}} \\ \quad evaluate(F_1, \alpha_i) \cdot F_2 \\ \text{else} \\ \quad evaluate(F_1, \alpha_i) \cdot F_2 \vee evaluate(F_2, \alpha_i) \end{cases}
$$

2.4 Relationship to Other Logics

According to [3],

We will use it to create formulas that describe the absence of some attacks and given a trace of events σ generated from the logs we will check whether

3 MONID: Structure analysis

4 Example of attacks

5 Conclusions

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.