Analyzing Multiple Logs for Forensic Evidence

By

Ali Reza Arasteh, Mourad Debbabi, Assaad Sakha, and Mohamed Saleh

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2007 USA

Pittsburgh, PA (Aug 13th - 15th)

available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Analyzing multiple logs for forensic evidence[☆]

Ali Reza Arasteh, Mourad Debbabi, Assaad Sakha, Mohamed Saleh*

Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada

Keywords:
Forensic analysis
Log analysis
Formal methods
Model checking
Logging systems
Log correlation

ABSTRACT

Information stored in logs of a computer system is of crucial importance to gather forensic evidence of investigated actions or attacks. Analysis of this information should be rigorous and credible, hence it lends itself to formal methods. We propose a model checking approach to the formalization of the forensic analysis of logs. A set of logs is modeled as a tree whose labels are events extracted from the logs. In order to provide a structure to these events, we express each event as a term of algebra. The signature of the algebra is carefully chosen to include all relevant information necessary to conduct the analysis. Properties of the model, attack scenarios, and event sequences are expressed as formulas of a logic having dynamic, linear, temporal, and modal characteristics. Moreover, we provide a tableau-based proof system for this logic upon which a model checking algorithm can be developed. We use our model in a case study to demonstrate how events leading to an SYN attack can be reconstructed from a number of system logs.

1. Introduction and motivation

Attacks on IT systems are increasing in number, and sophistication at an alarming rate. These systems now range from servers to mobile devices and the damage from such attacks is estimated in billions of dollars. However, due to the borderless nature of cyber attacks, many criminals/offenders have been able to evade responsibility due to the lack of supporting evidence to convict them. In this context, cyber forensics plays a major role by providing scientifically proven methods to gather, process, interpret, and use digital evidence to bring a conclusive description of cyber crime activities. The development of forensics IT solutions for law enforcement has been limited. Although outstanding results have been achieved for forensically sound evidence gathering, little has been done on the automatic analysis of the acquired evidence. Furthermore, limited efforts have been made into formalizing the digital forensic science. In many cases, the forensic procedures employed are constructed in an ad hoc manner that impedes the effectiveness or the integrity of the investigation. In this paper, we contribute with an automatic and formal approach to the problem of analyzing logs with the purpose of gathering forensic evidence.

One of the most common sources of evidence that an investigator should analyze is logged events from the activities of the system that is related to the incident in question. Indeed, having the logs from all system events during the incident will reduce the process of forensics analysis to event reconstruction. However, log analysis depends largely on the analyst's skills and experience to effectively decipher complex log patterns and determine what information is pertinent and useful to support the case at hand. Despite the paramount importance of this aspect, not much research effort has been dedicated to the automation of forensic log analysis. The main

[☆] The research reported in this paper is the result of a fruitful collaboration with Bell Canada under the PROMPT Quebec research partnership program.

* *Corresponding author.*

E-mail addresses: a_araste@ciise.concordia.ca (A.R. Arasteh), debbabi@ciise.concordia.ca (M. Debbabi), a_sakha@ciise.concordia.ca (A. Sakha), m_saleh@ciise.concordia.ca (M. Saleh).

1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.013

intent of this paper is to introduce a formal and automatic log analysis technique. The advocated approach caters for:

- Modeling of log events and logical representation of properties that should be satisfied by the traces of system events.
- Formal and automatic analysis of the logs looking for a specific pattern of events or verifying a particular forensic hypothesis.

In spite of the few research results on formal and automatic analysis of forensic and digital evidence, there are some important proposals that we detail hereafter.

Current research efforts on cyber forensic analysis can be categorized into baseline analysis, root cause analysis, common vulnerability analysis, timeline analysis, and semantic integrity check analysis. The baseline analysis, proposed in Monroe and Bailey (2003), uses an automated tool that checks for differences between a baseline of the safe state of the system and the state during the incident. The work presented in Stephenson (2003) proposes an approach to post-incident root cause analysis of digital incidents through a separation of the information system into different security domains and modeling the transactions between these domains. Common vulnerability analysis (Tenable Network Security, 2007) involves searching through a database of common vulnerabilities and investigating the case according to the related past and known vulnerabilities. The timeline analysis approach (Hosmer, 1998) consists of analyzing logs, scheduling information, and memory to develop a timeline of the events that led to the incident. Finally, the semantic integrity checking approach (Stallard and Levitt, 2003) uses a decision engine that is endowed with a tree to detect semantic incongruities. The decision tree reflects pre-determined invariant relationships between redundant digital objects.

Gladyshev and Patel (2004) proposed a formalization of digital evidence and event reconstruction based on finite state machines. In his work, the behavior of the system is modeled as a state machine and a recursive model checking procedure is proposed to verify the admissibility of a forensic hypothesis. However, in the real world, modeling the behavior of a complex system such as an operating system as a state machine diagram is burdensome and sometimes impossible to achieve because of complexity issues. Other research on formalized forensic analysis include the formalization of event time binding in digital investigation (Gladyshev and Patel, 2005; Leigland and Krings, 2004), which proposes an approach to constructing formalized forensic procedures. Nevertheless, the science of digital forensics still lacks a formalized approach to log analysis.

As for log analysis and correlation, some research has been done on alert correlation, which can be classified into four categories (Xu, 2007):

- Similarity based approaches (Cuppens, 2001; Julisch, 2003; Staniford et al., 2002; Valdes and Skinner, 2001), which group the alerts according to the similarity between alert attributes.
- Predefined attack scenario based approaches (Debar and Wespi, 2001; Morin and Debar, 2003), which detect attacks according to well-defined attack scenarios. However, they cannot discover novel attack scenarios.
- Pre-/post-condition based approaches (Cuppens and Miege, 2003; Ning et al., 2002; Templeton and Levitt, 2000) that match the post-condition of an attack to the pre-conditions of another attack.
- The multiple information sources based approaches (Morin et al., 2002; Porras et al., 2002; Yegneswaran et al., 2004) that are concerned with distributed attack discovery.

However, these approaches are mainly concerned with correlation, and intrusion detection, while formal log analysis and hypothesis verification is of paramount importance to forensic science. As an example, invariant properties of the system cannot be modeled and analyzed through the above approaches. The absence of a satisfactory and a general methodology for forensic log analysis has resulted in ad hoc analysis techniques such as log analysis (Peikari and Chuvakin, 2004) and operating system-specific analysis (Kruse and Heiser, 2002).

In this paper, we propose a new approach for log analysis that is based on computational logic and formal automatic verification. We start by developing a model of logs based on traces of events. Each event is actually an abstract view of a piece of information stored in the log. The structure of an event is carefully chosen to convey the necessary information needed for the analysis. To this end, events are represented as terms of a multi-sorted term algebra whose operation symbols are chosen such that *they* faithfully convey the information stored in the actual events stored in the log. For instance the term $DeleteFile(F, U)$ with operation $DeleteFile: File \times User \rightarrow Bool$ represents the deletion of file F by a user U. Using this approach, we can reason about log events irrespective of the specific syntax of the log, which is usually different for different systems. Each log in the system is thus modeled as a trace of terms. Moreover, in the presence of several logs to which information is written concurrently, the whole logging system is modeled as a tree that represents possible different interleavings of events from the logs.

To express properties of the model, we resort to a temporal, dynamic, modal and linear logic. This logic is an accommodated version of ADM logic that has been initially proposed in Adi et al. (2003). The motivation behind this choice is that ADM comes with many features and attributes that make it very suitable for what we intend to achieve since it deals with properties of traces. The modified version that we present in this paper deals with properties of terms and trees. Also, ADM is very compact in its syntax, elegant and formal in its semantics and high in terms of expressiveness. Actually, it is temporal (through the use of modal operators), dynamic (through the use of patterns as arguments in the modalities) and linear (by allowing model modifications in the logic semantics). Besides, it comes with fixpoint operators à la modal μ-calculus, which allows for the specification of properties that are finite encodings of infinite logical formulas. All this expressiveness is extremely useful in capturing forensic properties, hypotheses and system invariants. Moreover, we present a tableau-based proof system that defines a compositional model checking algorithm. All these features provide a rigorous and provable logical support, which is a necessity for an investigation to be admitted in courts of law.

This paper is organized in five sections. In Section 2, we present our approach to the formal modeling of logs. Section 3 is devoted to the logic used to express properties of the model. Section 4 contains an example of how logs can be modeled and their properties expressed in our framework. Section 5 presents a case study dealing with an SYN attack. Finally, conclusions and future work are discussed in Section 6.

2. Model for logs

In this section, we describe our model for system logs which is a tree of algebraic terms. First, we briefly recall concepts from universal algebra (Wechler, 1992) then we present the model.

A multi-sorted signature Σ is a pair $\langle \mathbb{S}, \mathbb{F} \rangle$, where $\mathbb{S}$ is a set of sorts and $\mathbb{F}$ is a set of operator symbols. For each operator symbol f, the function **arity** $: \mathbb{F} \to \mathbb{N}$ maps f to a natural number called the arity of f. Moreover, for any operator symbol $f \in \mathbb{F}$ of arity n, we have the functions $\mathbf{dom}(f) \in \mathbb{S}^n$ and $\mathbf{cod}(f) \in \mathbb{S}$, where $\mathbb{S}^0 = \emptyset$, $\mathbb{S}^1 = \mathbb{S}$, and $\mathbb{S}^{n+1} = \mathbb{S} \times \mathbb{S}^n$. For an operator symbol f, where $\mathbf{dom}(f) = S_1 \times S_2 \times \cdots \times S_n$, and $\mathbf{cod}(f) = S$, f is called an operator of sort S and we write $f : S_1 \times S_2 \times \cdots \times S_n \to S$. In the case where $\mathbf{arity}(f) = 0$, and $\mathbf{cod}(f) = S$, f is called a constant symbol of sort S, the set of constants of sort S is written $\mathbb{C}_S$. Also, for each sort S, we define an infinite set $\mathbb{X}_S$ of countable elements called variables of sort S, such that $\mathbb{X}_S \cap \mathbb{C}_S = \emptyset$. For a certain signature Σ, we define the set $\mathbb{T}_{\sum}(S, X)$ of free terms of sort S inductively as follows (the symbol '$\Rightarrow$' is for logic implication):

$$(\mathbb{X}_S \cup \mathbb{C}_S) \subseteq \mathbb{T}_\Sigma(S, X)$$

$$f : S_1 \times S_2 \times \cdots \times S_n \to S \Rightarrow f(t_1, \ldots, t_n) \in \mathbb{T}_\Sigma(S, X)$$

where each t_i is a term of sort S_i

The set $\mathbb{T}_{\sum}(X)$ of free terms over the signature Σ is defined as $\cup_{s \in \mathbb{S}} \mathbb{T}_{\sum}(S, X)$. The set of ground terms $\mathbb{T}_{\sum} \subseteq \mathbb{T}_{\sum}(X)$ includes all terms that do not contain any variables.

An algebra $\mathcal{A}$ of a signature Σ (called a Σ-algebra) is a pair $\langle \mathbb{A}, \mathbb{F}_{\mathcal{A}} \rangle$. The Σ-algebra $\mathcal{A}$ assigns to each sort S in the signature Σ a set $\mathbb{A}_S$ called the carrier of sort S, where $\mathbb{A} = \cup_{S \in \mathbb{S}} \mathbb{A}_S$. Also for each operator symbol $f : S_1 \times S_2 \times \cdots \times S_n \to S$, $\mathcal{A}$ assigns a function $f_{\mathcal{A}} : A_{S_1} \times A_{S_2} \times \ldots \times A_{S_n} \to A_S$, $\mathbb{F}_{\mathcal{A}}$ is the set of all functions $f_{\mathcal{A}}$. A homomorphism is a function between Σ-algebras that reserves their structure. If $\mathcal{A}$ and $\mathcal{B}$ are two Σ-algebras having the same signature, $h : \mathcal{A} \to \mathcal{B}$ is a called a Σ-homomorphism iff:

$$\forall f \in \mathbb{F} \cdot h(f_{\mathcal{A}}(a_1, a_2, \ldots, a_n)) = f_{\mathcal{B}}(h(a_1), h(a_2), \ldots, h(a_n))$$

A Σ-algebra provides an interpretation for a certain signature, where each sort is interpreted as a set and each operator symbol as a function. The free term algebra $\mathcal{T}_\Sigma(X)$ associated with a signature Σ is a special kind of Σ-algebra in which each carrier set $\mathbb{A}_S$ of the sort S and each function $f_{\mathcal{T}_\Sigma(X)} : \mathbb{A}_{S_1} \times \mathbb{A}_{S_2} \times \cdots \times \mathbb{A}_{S_n} \to \mathbb{A}_S$ are defined as:

$$\forall t \cdot t \in \mathbb{T}_\Sigma(S, X) \Leftrightarrow t \in \mathbb{A}_S$$

$$f_{\mathcal{T}_\Sigma(X)}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n) \quad \text{where } t_i \in \mathbb{A}_{S_i}$$

The term algebra $\mathcal{T}_\Sigma$ can be obtained from $\mathcal{T}_\Sigma(X)$ by removing any terms that contain variables. A substitution $\theta_x : \mathbb{X}_S \to \mathbb{T}_\Sigma(S, X)$ is a mapping from variables to terms of the same sort. A ground substitution maps variables to ground terms. A substitution is generally extended to a homomorphism in the following way:

$$\theta_x : \mathbb{X}_S \to \mathbb{T}_\Sigma(S, X) \text{ is extended to}$$
$$\tilde{\theta}_x : \mathbb{T}_\Sigma(S, X) \to \mathbb{T}_\Sigma(S, X)$$
$$\tilde{\theta}_x(c) = c \text{ where } c \text{ is a constant}$$
$$\tilde{\theta}_x(f(t_1, t_2, \ldots, t_n)) = f\left(\tilde{\theta}_x(t_1), \tilde{\theta}_x(t_2), \ldots, \tilde{\theta}_x(t_n)\right)$$

Usually we write θ_x for $\tilde{\theta}_x$.

We view a log as a sequence of log entries, each entry consists of a certain term t of a term algebra $\mathcal{T}_\Sigma$. The signature Σ is chosen such that each event monitored by the logging system can be represented as a term $t \in \mathbb{T}_\Sigma$. We define the log model L as a finite sequence over $\mathbb{T}_\Sigma$, i.e., $L \in \mathbb{T}_\Sigma^*$. In other words, the log model is a sequence that represents logged events in the system ordered temporally. Graphically, the log model is a trace whose edges are labeled by terms of $\mathbb{T}_\Sigma$. However, many systems may have more than one log, where each log is dedicated to a certain category of events. Moreover, we may be faced by situations where it is necessary to inspect logs from different machines, e.g., in the case of a network. In order to model a log system consisting of more than one log, and in the presence of concurrent actions in the logs, the model becomes a tree. To illustrate this, assume we have a sequence of actions $a \cdot b$ in log L_1 and another sequence $c \cdot d$ in log L_2. Temporally, we assume that a occurred first then c, then b and d occurred at the same time. The combined trace of the two logs will be a tree L, which is the following set of traces $L = \{a, a \cdot c, a \cdot c \cdot b, a \cdot c \cdot d, a \cdot c \cdot b \cdot d, a \cdot c \cdot d \cdot b\}$, i.e., we considered the two possible interleavings of the concurrent events b and d. The same basic idea is used to construct synchronization trees of process calculi. We define the interleaving of two logs L_1 and L_2 to be $L_1 \| L_2 \subseteq (\mathbb{T}_{\Sigma 1} \cup \mathbb{T}_{\Sigma 2})^*$ which represents all possible interleavings of concurrent events from both logs. Here, we assumed that each log has its own signature. The definition can be easily generalized in the case of a finite number of logs. Defined this way, a log system of more than one log is graphically represented as a tree whose branches are labeled by terms of term algebras. Our model consists of this log tree along with a mapping **orig** that maps terms to their respective logs. For instance, in the example above, we have $\mathbf{orig}(a) = L_1$. For a sequence $s \in L$, we define $s \upharpoonright \mathbb{P}$, where $\mathbb{P}$ is a set of individual logs, to be the sequence obtained from s by removing any terms t, such that $\mathbf{orig}(t) \notin \mathbb{P}$. We overload the $\upharpoonright$ operator by defining $L \upharpoonright \mathbb{P}$ to be the set $\{s \upharpoonright \mathbb{P} | s \in L\}$. In the example above, we have: $L \upharpoonright L_1 = \{a,\ a \cdot b\}$, $L \upharpoonright L_2 = \{c,\ c \cdot d\}$, and $L \upharpoonright \{L_1,\ L_2\} = L$. We note here that $L \upharpoonright \mathbb{P}$ is also a tree.

3. Logic for log properties

In this section, we present a new logic for the specification of properties of the log model. The logic is based on ideas from the ADM logic (Adi et al., 2003), with some basic differences. First, ADM is trace-based while the logic we present is tree-based, therefore we can quantify existentially and universally over traces. Moreover, this gives us the opportunity to express branching-time properties. Second, we add a "sub-tree" operator that will allow us to quantify with respect to individual logs, e.g., all sequences of the tree that contain events from logs L_1 and L_2 only. Third, the actions in ADM are atomic symbols whereas the actions in our logic have a structure since they are terms of a term algebra. The choice of ADM in the first place is motivated by the fact that is based on modal μ-calculus with its known expressive power. Most importantly, the properties we would like to express are over traces of the log tree model, ADM deals with traces and has all the expressive power we need for this task including some counting properties (Adi et al., 2003).

3.1. *Syntax*

Before presenting the syntax of formulas, we present the concept of a sequence pattern r. A sequence pattern has the following syntax:

$$r ::= \epsilon | a \cdot r | x_r \cdot r \quad a ::= t | \lceil t \rceil \tag{1}$$

Here a represents a term in $\mathbb{T}_\Sigma(X)$ and has the form t or $\lceil t \rceil$. Intuitively, a represents a term in the log tree model, where this term is either t or a term containing $t(\lceil t \rceil)$. The term $\lceil t \rceil$ is defined as t or $f(t_1, t_2, \ldots, t_n)$ such that $\exists t_i \cdot t_i = \lceil t \rceil$. Here $f \in \Sigma$ is any function symbol in the signature of the algebra. The sequence variable x_r represents a sequence of terms of zero or any finite length, the subscript r is added to avoid confusion with term variables x of the message algebra. The sets of sequence variables and terms in r are written $var(r)$ and $trm(r)$, respectively. Moreover, for any pattern r, the symbol $r|_i$ represents the variable or move at position i of r, where $i \in \{1, \ldots, n\}$.

We define the substitution $\theta_r : var(r) \to \mathbb{T}^*_\Sigma$ that maps variables x_r in a sequence pattern to sequences of terms, this is not to be confused with the substitution θ_x that maps variables inside messages into terms of $T_\Sigma(X)$.

We define the predicate **match**(σ, r, θ_m, θ_r), which is true when a sequence $\sigma = s_1 \cdot s_2 \ldots s_n$ in the log tree *matches* a pattern r, ϵ is the empty sequence:

$$\begin{aligned}
&\mathbf{match}(\epsilon, \epsilon, \theta_m, \theta_r) = \mathit{true}\\
&\mathbf{match}(\sigma, \epsilon, \theta_m, \theta_r) = \mathit{false} \quad \text{if } \sigma \neq \epsilon\\
&\mathbf{match}(\sigma, a \cdot r, \theta_m, \theta_r) = (s_1 = a\theta_m) \wedge \mathbf{match}(s_2 \ldots s_n, r, \theta_m, \theta_r)\\
&\mathbf{match}(\sigma, x_r \cdot r, \theta_m, \theta_r) = \left(\exists j \leq n \cdot \left(x_r\theta_r = s_1 \ldots s_j\right)\right)\\
&\qquad\qquad\qquad\qquad \wedge \mathbf{match}\left(s_{j+1} \ldots s_n, r, \theta_m, \theta_r\right)
\end{aligned}$$

A substitution $\theta : \mathcal{R} \to \mathbb{T}^*_\Sigma$ from patterns to sequences of terms, where $\theta = \theta_m \cup \theta_r$, is defined as follows:

$$\begin{aligned}
&\theta(\epsilon) = \epsilon\\
&\theta(a \cdot r) = \theta_m(a) \cdot \theta(r)\\
&\theta(x_r \cdot r) = \theta_r(x_r) \cdot \theta(r)
\end{aligned}$$

We will follow the usual notation for substitutions and write $r\theta$ for $\theta(r)$. From the definitions of the predicate **match** and the substitution θ above, we notice that the condition for a match between a pattern and a sequence is the existence of one or more substitutions θ, we can therefore write the predicate as **match**(σ, r, θ).

The syntax of a formula ϕ is expressed by the following grammar:

$$\varphi ::= Z | \neg\varphi | \varphi_1 \wedge \varphi_2 | [r_1 \looparrowright r_2]\varphi | \langle\langle \mathcal{L} \rangle\rangle \varphi | \nu Z \cdot \varphi \tag{2}$$

We require the following two syntactic conditions:

- In $[r_1 \looparrowright r_2]$: $\forall i \cdot (r_1|_i \in var(r_1) \Leftrightarrow r_1|_i = r_2|_i) \wedge (r_1|_i \in trm(r_1) \Leftrightarrow r_1|_i = r_2|_i \vee r_2|_i = \circledast)$.
- In $\nu Z \cdot \varphi$, any free Z in φ appears under the scope of an even number of negations.

The first condition above means that r_2 is obtained from r_1 by replacing some of the terms of r_1 by the dummy symbol $\circledast$, where $\theta(\circledast) = \circledast$, i.e., sequence variables of r_1 and r_2 are the same. This condition is necessary to ensure that $r_1\theta$ and $r_2\theta$ have the same length. Hence, in a tree model L, if $r_1\theta \in L$, we can replace $r_1\theta$ by $r_2\theta$ and still get a tree, which is written $L' = L[r_2\theta / r_1\theta]$. The second condition is necessary for the semantic interpretation function as will be explained in the semantics section.

Intuitively, a log tree model L satisfies the formula $[r_1 \looparrowright r_2]\varphi$ if for all sequences $\sigma \in L$ that matches r_1, i.e., $r_1\theta = \sigma$, when σ is modified to match r_2 (by removing some of its terms) then $\sigma' = r_2\theta$ satisfies φ. The model L satisfies $\langle\langle \mathcal{L} \rangle\rangle \varphi$, where $\mathcal{L}$ is a set of logs, if $L \restriction \mathcal{L}$ satisfies φ. The rest of the formulas have their usual meanings in modal μ-calculus (Cleaveland, 1990).

3.2. *Semantics*

A formula in the logic is interpreted over a log tree. Given a certain log tree L, and an environment e that maps formulae variables to sequences in L, the semantic function $[\![\varphi]\!]^L_e$ maps a formula φ to a set of sequences $S \subseteq L$ that satisfy this formula.

$$\begin{aligned}
&[\![Z]\!]^L_e = e(Z)\\
&[\![\neg\varphi]\!]^L_e = L \setminus [\![\varphi]\!]^L_e\\
&[\![\varphi_1 \wedge \varphi_2]\!]^L_e = [\![\varphi_1]\!]^L_e \cap [\![\varphi_2]\!]^L_e\\
&[\![[r_1 \looparrowright r_2]\varphi]\!]^L_e = \left\{\sigma \in L | \forall \theta \cdot \mathbf{match}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi]\!]^{L'}_e\right\},\\
&\text{where } \sigma' = r_2\theta \text{ and } L' = L[r_2\theta / r_1\theta]\\
&[\![\langle\langle \mathcal{L} \rangle\rangle \varphi]\!]^L_e = (L \restriction \mathcal{L}) \cap [\![\varphi]\!]^L_e\\
&[\![\nu Z \cdot \varphi]\!]^L_e = \cup \left\{S \subseteq L | S \subseteq [\![\varphi]\!]^L_{e[Z \mapsto S]}\right\}
\end{aligned} \tag{3}$$

From the semantic equations above it can be seen that the meaning of the recursive formula $\nu Z \cdot \varphi$ is taken to be the greatest fixpoint of a function $f : 2^L \to 2^L$, where $f(S) = [\![\varphi]\!]^L_{e[Z \mapsto S]}$. The function f is defined over the lattice $(2^L, \subseteq, \cup, \cap)$, the syntactic condition on φ (X appears under the scope of an even number of negations) ensures that $f(S)$ is monotone (Cleaveland, 1990) and hence has a greatest fixpoint.

We use the following shorthand notations:

$$\begin{aligned}
&\neg(\neg\varphi_1 \wedge \neg\varphi_2) \equiv \varphi_1 \vee \varphi_2\\
&\neg\varphi_1 \vee \varphi_2 \equiv \varphi_1 \Rightarrow \varphi_2\\
&\neg[r_1 \looparrowright r_2]\neg\varphi \equiv \langle r_1 \looparrowright r_2 \rangle \varphi\\
&\neg\nu Z \cdot \neg\varphi[\neg Z / Z] \equiv \mu Z \cdot \varphi
\end{aligned}$$

We also define $\nu Z \cdot Z$ to be `tt`, where $[\![\mathtt{tt}]\!]^L_e = L$ and $\mu Z \cdot Z$ to be `ff`, where $[\![\mathtt{ff}]\!]^L_e = \emptyset$. In the following, we prove some important results regarding the logic.

Lemma 3.1.

$$[\![\varphi[\psi/Z]]\!]^L_e = [\![\varphi]\!]^L_{e[Z \mapsto [\![\psi]\!]^L_e]}$$

Proof. The proof is done by structural induction over φ. Base case: $\varphi = Z$

$$[\![\psi/Z]\!]^L_e = [\![\psi]\!]^L_e$$

But: $[\![Z]\!]^L_e = e(Z)$, so $[\![\psi/Z]\!]^L_e = [\![Z]\!]^L_{e[Z \mapsto [\![\psi]\!]^L_e]}$.

We demonstrate two cases and the other cases can be easily proved:

Case 1. $\varphi = \nu Z\cdot\varphi'$ and X is free in φ

$$[\![\varphi[\psi/X]]\!]^L_e = \cup\left\{S\subseteq L\middle|S\subseteq[\![\varphi'[\psi/X]]\!]^L_{e[Z\mapsto S]}\right\}$$

By induction hypothesis:

$$[\![\varphi[\psi/X]]\!]^L_e = \cup\left\{S\subseteq L\middle|S\subseteq[\![\varphi']\!]^L_{e[Z\mapsto S]\ [X\mapsto[\![\psi]\!]^L_{e[Z\mapsto S]}]}\right\}$$

Since ψ does not contain Z as free variable, which can be assured by renaming of the bound variable Z, we have:

$$[\![\varphi[\psi/X]]\!]^L_e = \cup\left\{S\subseteq L|S\subseteq[\![\varphi'[\psi/X]]\!]^L_{e[Z\mapsto S][X\mapsto[\![\psi]\!]^L_e]}\right\}$$

$$[\![\varphi[\psi/X]]\!]^L_e = [\![\varphi]\!]^L_{e[X\mapsto[\![\psi]\!]^L_e]}$$

Case 2. $\varphi = [r_1 \looparrowright r_2]\varphi'$

$$[\![\varphi[\psi/Z]]\!]^L_e = \left\{\sigma\in L|\forall\theta\cdot\mathbf{match}\ \ (\sigma,r_1,\theta)\Rightarrow\sigma'\in[\![\varphi'[\psi/Z]]\!]^{L'}_e\right\}$$

By induction hypothesis:

$$[\![\varphi[\psi/Z]]\!]^L_e = \left\{\sigma\in L|\forall\theta\cdot\mathbf{match}(\sigma,r_1,\theta)\Rightarrow\sigma'\in[\![\varphi']\!]^{L'}_{e[Z\mapsto[\![\psi]\!]^L_e]}\right\}$$

$$[\![\varphi[\psi/Z]]\!]^L_e = [\![\varphi]\!]^L_{e[Z\mapsto[\![\psi]\!]^L_e]} \qquad \square$$

As a result, we have $[\![\upsilon Z\cdot\varphi]\!]^L_e = [\![\varphi[\upsilon Z\cdot\varphi/Z]]\!]^L_e$. This follows from the fact that $[\![\upsilon Z\cdot\varphi]\!]^L_e = [\![\varphi]\!]^L_{e[Z\mapsto T]}$, where $T = \cup\left\{S\subseteq L|S\subseteq [\![\varphi]\!]^L_{e[Z\mapsto S]}\right\} = [\![\upsilon Z\cdot\varphi]\!]^L_e$.

We can now prove that the semantics of the expression $\mu Z\cdot\varphi$ defined earlier as $\neg\nu Z\cdot\neg\varphi[\neg Z/Z]$ is the least fixpoint of the function $f(S) = [\![\varphi]\!]^L_{e[Z\mapsto S]}$.

$$\begin{aligned}[\![\neg\nu Z\cdot\neg\varphi[\neg Z/Z]]\!]^L_e &= L\setminus\cup\left\{S\subseteq L|S\subseteq[\![\neg\varphi[\neg Z/Z]]\!]^L_{e[Z\mapsto S]}\right\}\\ &= L\setminus\cup\left\{S\subseteq L\middle|S\subseteq L\setminus[\![\varphi]\!]^L_{e[Z\mapsto[\![\neg Z]\!]^L_{e[Z\mapsto S]}]}\right\}\\ &= L\setminus\cup\left\{S\subseteq L\middle|S\subseteq L\setminus[\![\varphi]\!]^L_{e[Z\mapsto L\setminus S]}\right\}\end{aligned}$$

For any set of sequences $S \subseteq L$, let $S^c = L\backslash S$. By De Morgan laws, for any two sets A and B: $(A \cap B)^c = A^c \cup B^c$, $(A \cup B)^c = A^c \cap B^c$, $A \subseteq B \Rightarrow B^c \subseteq A^c$.

$$\begin{aligned}[\![\neg\nu Z\cdot\neg\varphi[\neg Z/Z]]\!]^L_e &= \left(\cup\{L\setminus S^c\subseteq L|S\subseteq\left([\![\varphi]\!]^L_{e[Z\mapsto S^c]}\right)^c\}\right)^c\\ &= \left(\cup\{L\setminus S^c\subseteq L|[\![\varphi]\!]^L_{e[Z\mapsto S^c]}\subseteq S^c\}\right)^c\\ &= \cap\left(\left\{L\setminus S^c\subseteq L|[\![\varphi]\!]^L_{e[Z\mapsto S^c]}\subseteq S^c\right\}\right)^c\\ &= \cap\left\{S^c\subseteq L|[\![\varphi]\!]^L_{e[Z\mapsto S^c]}\subseteq S^c\right\}\end{aligned}$$

Moreover, we investigate the semantics of the expression $\langle r_1 \looparrowright r_2\rangle\varphi$ as defined above:

$$\begin{aligned}[\![\langle r_1\looparrowright r_2\rangle\varphi]\!]^L_e &= [\![\neg[r_1\looparrowright r_2]\neg\varphi]\!]^L_e\\ &= \left\{\sigma\in L|\neg\forall\theta\cdot\mathbf{match}(\sigma,r_1,\theta)\Rightarrow\sigma'\in L\setminus[\![\varphi]\!]^{L'}_e\right\}\\ &= \left\{\sigma\in L|\neg\forall\theta\cdot\mathbf{match}(\sigma,r_1,\theta)\Rightarrow\neg\sigma'\in[\![\varphi]\!]^{L'}_e\right\}\\ &= \left\{\sigma\in L|\neg\forall\theta\cdot\neg\left(\mathbf{match}(\sigma,r_1,\theta)\wedge\sigma'\in[\![\varphi]\!]^{L',\theta}_e\right)\right\}\\ &= \left\{\sigma\in L|\exists\theta'\cdot\left(\mathbf{match}(\sigma,r_1,\theta)\wedge\sigma'\in[\![\varphi]\!]^{L',\theta}_e\right)\right\}\end{aligned}$$

In the derivation above we used the sequent $\psi \Rightarrow \neg\varphi \vdash \neg(\psi\wedge\varphi)$, which can be easily proved by propositional calculus. We also used the fact that for any set of sequences S, $S \cap (L\backslash S) = \emptyset$. It is worth noting here that the semantics of $\langle r_1 \looparrowright r_2\rangle\varphi$ is consistent with the definition of the modality $\langle\rangle$ from modal μ-calculus.

3.3. *Tableau-based proof system*

Before we present the rules of the tableau, we define the immediate subformula relation (Cleaveland, 1990) $\prec_I$ as:

$\varphi\prec_I\neg\varphi \quad \varphi\prec_I[r_1\looparrowright r_2]\varphi$
$\varphi_i\prec_I\varphi_1\wedge\varphi_2 \quad i\in\{1,2\} \quad \varphi\prec_I\nu Z\cdot\varphi$

We define $\prec$ to be the transitive closure of $\prec_I$ and $\preceq$ to be its transitive and reflexive closure. A tableau-based proof system starts from the formula to be proved as the root of a proof tree and proceeds in a top down fashion. In every rule of the tableau, the conclusion is above the premises. Each conclusion of a certain rule represents a node in the proof tree, whereas the premises represent the children to this node. In our case, the proof system proves sequents of the form $H, b\vdash\ \sigma\in\varphi$, which means that under a set H of hypotheses and the symbol b, then the sequence σ satisfies the property φ. The set H contains elements of the form σ: $\nu Z\cdot\varphi$ and is needed for recursive formulas. Roughly, the use of H is to say that in order to prove that a sequence σ satisfies a recursive formula φ_{rec}, we must prove the following: Under the hypothesis that σ satisfies φ_{rec}, then σ also satisfies the unfolding of φ_{rec}. We also define the set $H \Rsh \nu Z\cdot\varphi = \{\sigma\in L|\sigma{:}\nu Z\cdot\varphi\in H\}$. The use of H, and b will be apparent after we state the rules of the proof system:

$$R_\neg \quad \frac{H, b\vdash\sigma\in\neg\varphi}{H, \neg b\vdash\sigma\in\varphi}$$

$$R_\wedge \quad \frac{H, b, \vdash\sigma\in\varphi_1\wedge\varphi_2}{H, b_1\vdash\sigma\in\varphi_1\ \ H, b_2\vdash\sigma\in\varphi_2} \quad b_1\times b_2 = b$$

$$R_\nu \quad \frac{H, b\vdash\sigma\in\nu Z\cdot\varphi}{H'\cup\{\sigma:\nu Z\cdot\varphi\}, b\vdash\sigma\in\varphi[\nu Z\cdot\varphi/Z]} \quad \sigma:\nu Z\cdot\varphi\notin/H$$

$$R_{[]} \quad \frac{H, b\vdash\sigma\in[r_1\looparrowright r_2]\varphi}{\xi_1\xi_2\cdots\xi_n} \quad \text{condition}$$

$$R_{\langle\langle\rangle\rangle} \quad \frac{H, b\vdash\sigma\in\langle\langle\mathcal{L}\rangle\rangle\varphi}{H, b\vdash\sigma\restriction\mathcal{L}\in\varphi}$$

where

$$H' = H\setminus\{\sigma':\Gamma|\nu Z\cdot\varphi\prec\Gamma\}$$

$$\xi_i = H, b_i\vdash r_2\theta_i\in\varphi$$

$$\text{condition} = \begin{cases}\forall\theta_i\cdot\mathbf{match}(\sigma, r_1, \theta_i)\\ \wedge\ b_1\times b_2\times\cdots\times b_n = b\\ \wedge\ n>0\end{cases}$$

The first rule concerns negation of formulas where $b\in\{\epsilon, \neg\}$ serves as a “memory” to remember negations, in this case $\epsilon\varphi=\varphi$. We define $\epsilon\epsilon=\epsilon$, $\epsilon\neg=\neg\epsilon=\neg$, and $\neg\neg=\epsilon$. Moreover, we define $\epsilon\times\epsilon=\epsilon$, $\epsilon\times\neg=\neg\times\epsilon=\neg$, and $\neg\times\neg=\neg$. The second rule says that in order to prove the conjunction, we have to prove both conjuncts. The third rule concerns proving a recursive formulas, where the construction of the set H, via H', ensures that the validity of the sequent $H, b\vdash\ \sigma\in\nu Z\cdot\varphi$ is determined only by subformulas of φ (Cleaveland, 1990). The fourth rule takes care of formulas matching sequences to patterns. Finally, the fifth rule deals with formulas dealing with subtrees of the model tree L. Starting from the formula to be proved at the root of the proof tree, the tree grows downwards until we hit a node where the tree cannot be extended anymore, i.e., a leaf node. A formula is proved if it has a successful tableau, where a successful tableau is one whose all leaves are successful. A successful leaf meets one of the following conditions:

- $H, \epsilon \vdash \sigma \in Z$ and $\sigma \in \llbracket Z \rrbracket_e^L$.
- $H, \neg \vdash \sigma \in Z$ and $\sigma \notin \llbracket Z \rrbracket_e^L$.
- $H, \epsilon \vdash \sigma \in \nu Z.\varphi$ and $\sigma : \nu Z.\varphi \in H$.
- $H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\{\sigma \in L | \exists \theta \cdot \mathbf{match}(\sigma, r_1, \theta)\} = \emptyset$.

In the next section, we list the most important properties of the tableau-based proof system.

3.4. Properties of tableau system

We would like to prove three main properties, namely the finiteness of the tableau for finite models, the soundness, and the completeness. Soundness and completeness are proved with respect to a relativized semantics that takes into account the set H of hypotheses. The new semantics is the same as the one provided above for all formulas except for recursive formulas where is it defined as:

$$\llbracket \nu Z \cdot \varphi \rrbracket_e^{L,H} = \left(\nu \llbracket \varphi \rrbracket_{e[Z \mapsto S \cup S']}^{L,H} \right) \cup S' \quad \text{where } S' = H \Rsh \nu Z \cdot \varphi$$

In the equation, the greatest fixpoint operator is applied to a function $f(S) = \llbracket \varphi \rrbracket_{e[Z \mapsto S]}^{L,\theta}$ whose argument is $S \cup S'$. Since the function is monotone over a complete lattice, as mentioned earlier, then the existence of a greatest fixpoint is guaranteed. We now list some results regarding the proof system. The detailed proofs are not provided due to space limitation.

Theorem 3.1. *Finiteness. For any sequent $H, b \vdash \sigma \in \varphi$ there exists a finite number of finite tableaux.*The idea of the proof is that for any formula at the root of the proof tree we begin applying the rules $R_\neg$, $R_\wedge$, $R_{[]}$, $R_{\langle\langle\rangle\rangle}$, and R_ν. The application of the first four rules results in shorter formulas, while the application of the R_ν rule results in larger hypothesis sets H. The proof shows that shortening a formula and increasing the size of H cannot continue infinitely. Hence no path in the tree will have infinite length. Branching happens in the proof tree whenever we have an expression of the form $\varphi_1 \wedge \varphi_2$ or $[r_1 \looparrowright r_2]\varphi$. Finite branching is guaranteed in the first case by the finite length of any expression and in the second case by the finiteness of the model.

Theorem 3.2. *Soundness. For any sequent $H, b \vdash \sigma \in \varphi$ with a successful tableau, $\sigma \in \llbracket \varphi \rrbracket_e^{L,H}$*

The idea behind the proof is to show that all the successful leaves described above are valid and that the application of the rules of the tableau reserves semantic validity.

Theorem 3.3. *Completeness. If for a sequence $\sigma \in L$, $\sigma \in \llbracket \varphi \rrbracket_e^{L,H}$, then the sequent $H, b \vdash \sigma \in \varphi$ has a successful tableau.*The proof relies on showing that we cannot have two successful tableaux for the sequents $H, b \vdash \sigma \in \varphi$ and $H, b \vdash \sigma \in \neg\varphi$.

4. Example of logs and properties

In this section we give an example of constructing a model of actual log systems and expressing some general properties. The methodology we apply here can be easily adapted to various systems of logs. We begin by listing the signature of our algebra and the notation used for constants and variables. The sorts of the signature are given in Table 1, whereas the operations are given in Sections 4.1 and 4.2. These operations are categorized based on the context to which they are related. In most cases, the notation we use for the names of operations and sorts reveals their intended meaning and in case of ambiguity we will provide explanation. So, in order to model a single log, a parser can be developed that will parse the log and construct its model as a sequence of algebraic terms, where each term corresponds to a certain log event. The model for a system of several logs will be a tree whose labels are terms as explained in Section 2. Once the model is constructed, the desired properties can be expressed as formulas of the logic we presented. The model and logic can then be submitted to a model checker that will check if the model satisfies the formula. The presence of a tableau-based proof system for the logic serves as a starting point for the implementation of a model checking algorithm. Such a framework can be used in event reconstruction based on expressing scenarios and hypotheses as properties, and then verify their validity, this will be explained in Table 1.

Table 1 – Sorts of the algebra

Sorts	Constants	Variables
File	$f_1, f_2, \ldots, f_n$	$f_x, f_y, f_z, \ldots$
User	$u_1, u_2, \ldots, u_n$	$u_x, u_y, u_z, \ldots$
Object	$o_1, o_2, \ldots, o_n$	$o_x, o_y, o_z, \ldots$
objectedID	$oID_1, oID_2, \ldots, oID_n$	$oID_x, oID_y, oID_z, \ldots$
Process	$p_1, p_2, \ldots, p_n$	$p_x, p_y, p_z, \ldots$
Service	$s_1, s_2, \ldots, s_n$	$s_x, s_y, s_z, \ldots$
service_type	$st_1, st_2, \ldots, st_n$	$st_x, st_y, st_z, \ldots$
registry_key	$r_1, r_2, \ldots, r_n$	$r_x, r_y, r_z, \ldots$
Operation	$op_{\mathrm{READ}}, op_{\mathrm{WRITE}}$	$op_x, op_y, op_z, \ldots$
Executable	$e_1, e_2, \ldots, e_n$	$e_x, e_y, e_z, \ldots$
Privilege	pv_{ADMIN}	$pv_x, pv_y, pv_z, \ldots$
Access	$ac_{N\_Allowed}, ac_{\mathrm{ALLOWED}}$	$ac_x, ac_y, ac_z, \ldots$
Port	$pt_{\mathrm{PORT\_NO}}$	$pt_x, pt_y, pt_z, \ldots$
Protocol	$proc_{\mathrm{UDP}}, proc_{\mathrm{TCP}}, \ldots, u_n$	$proc_x, proc_y, proc_z, \ldots$
ipAddress	$ip_1, ip2, \ldots, ip_n$	$ip_x, ip_y, ip_z, \ldots$
dateTime	$dt_1, dt_2, \ldots, dt_n$	$dt_x, dt_y, dt_z, \ldots$
logonType	$lt_{\mathrm{INTERACTIVE}}, lt_{\mathrm{REMOTE}}, \ldots$	$lt_x, lt_y, lt_z, \ldots$
logonID	$lID_1, lID_2, \ldots, lID_3$	$lID_x, lID_y, lID_z, \ldots$
Size/type	$s\_t_{\mathrm{POD}} \ldots$	$s\_t_x, s\_t_y, s\_t_z, \ldots$
args	$a_1, a_2, \ldots, a_n$	$a_x, a_y, a_z, \ldots$
error	$er_1, er_2, \ldots, er_n$	$er_x, er_y, er_z, \ldots$
bool	Tr, Fl	$b_x, b_y, b_Z, \ldots$

4.1. *Machine events logs*

In this section, we discuss logs of events related to a single machine. This machine could be a stand alone system or a node in a network, however, the events we consider here are internal events related to a specific machine. It is not our intention to give an exhaustive list of all possible events but to demonstrate the use of our model using various categories of events. Each type of log events is modeled by an operation of the algebra as will be demonstrated below.

4.1.1. *Process related actions*

These are the events related to the creation and destruction of a process.

$$\begin{aligned}&\mathit{ProcessBegin}:\mathtt{process}\times\mathtt{user}\times\mathtt{caller}\times\mathtt{privilege}\\&\qquad\times\mathtt{executable}\rightarrow\mathtt{bool}\\&\mathit{ProcessExit}:\mathtt{process}\times\mathtt{user}\rightarrow\mathtt{bool}\end{aligned}$$

For instance, the term *ProcessExit*(p_x, u_x) models an actual log event in which any user (u_x) terminated any process (p_x), since u_x and p_x are variables. On the other hand, the term *ProcessExit* (p_x, u_1) models an actual log event in which a certain user (u_1) terminated any process (p_x). Table 1 presents our notation for variables and constants.

4.1.2. *Object related actions*

In a Windows system, an object is either a file or a registry key. The events which are related to objects are opening, modifying and closing the objects. These events are modeled as actions as follows:

$$\begin{aligned}&\mathit{ObjectOpen}:\mathtt{object}\times\mathtt{process}\times\mathtt{processName}\times\mathtt{type}\\&\qquad\times\mathtt{logonID}\rightarrow\mathtt{bool}\\&\mathit{ObjectClose}:\mathtt{object}\rightarrow\mathtt{bool}\\&\mathit{ObjectOperation}:\mathtt{object}\times\mathtt{operation}\rightarrow\mathtt{bool}\end{aligned}$$

4.1.3. *User actions*

These are actions related to creation, deletion, or modifications to user accounts and their rights as well as logons and logoffs.

$$\begin{aligned}&\mathit{Logon}:\mathtt{user}\times\mathtt{userID}\times\mathtt{group}\times\mathtt{domain}\times\mathtt{process}\\&\qquad\times\mathtt{logonType}\rightarrow\mathtt{bool}\\&\mathit{Logoff}:\mathtt{username}\rightarrow\mathtt{bool}\end{aligned}$$

4.1.4. *Miscellaneous*

Events that are not specific to any category are mentioned here.

$$\begin{aligned}&\mathit{ScheduleTask}:\mathtt{user}\times\mathtt{process}\times\mathtt{caller}\times\mathtt{file}\\&\qquad\times\mathtt{dateTime}\rightarrow\mathtt{bool}\\&\mathit{InstallService}:\mathtt{service}\times\mathtt{file}\times\mathtt{service\_type}\\&\qquad\times\mathtt{username}\rightarrow\mathtt{bool}\\&\mathit{UninstallService}:\mathtt{service}\times\mathtt{user}\times\mathtt{dateTime}\rightarrow\mathtt{bool}\\&\mathit{DeactAntiVirus}:\mathtt{user}\times\mathtt{dateTime}\rightarrow\mathtt{bool}\\&\mathit{DeactFirewall}:\mathtt{user}\times\mathtt{dateTime}\rightarrow\mathtt{bool}\\&\mathit{OpenConnection}:\mathtt{process}\times\mathtt{port}\times\mathtt{protocol}\times\mathtt{user}\\&\qquad\times\mathtt{access}\times\mathtt{dateTime}\rightarrow\mathtt{bool}\\&\mathit{ClearLogs}:\mathtt{user}\times\mathtt{dateTime}\rightarrow\mathtt{bool}\end{aligned}$$

4.2. *Network events logs*

In this section we consider logs that monitor network events, i.e., related to more than one machine.

4.2.1. *Intrusion detection system*

Here we use SNORT (Snort, 2007) as an example for our modeling. The SID shown below is the attack ID used by SNORT to detect a known signature.

$$\mathit{Attack}:\mathtt{dateTime}\times\mathtt{SID}\times\mathtt{sourceIP}\times\mathtt{destIP}\rightarrow\mathtt{bool}$$

4.2.2. *Web server*

$$\begin{aligned}&\mathit{GetError}:\mathtt{dateTime}\times\mathtt{error}\times\mathtt{ipAddress}\rightarrow\mathtt{bool}\\&\mathit{PostError}:\mathtt{dateTime}\times\mathtt{error}\rightarrow\mathtt{bool}\\&\mathit{Get}:\mathtt{dateTime}\times\mathtt{args}\times\mathtt{ipAddress}\rightarrow\mathtt{bool}\\&\mathit{Post}:\mathtt{dateTime}\times\mathtt{args}\times\mathtt{ipAddress}\rightarrow\mathtt{bool}\\&\mathit{UUCP}:\mathtt{dateTime}\times\mathtt{ipAddress}\rightarrow\mathtt{bool}\end{aligned}$$

Here args is the sort of arguments for the *Post* and *Get* actions.

4.3. *Properties and scenarios*

An important use of the logic is the ability to express an attack or a certain scenario of events using a pattern based on the general characterizations of this attack or scenario. In other words, an investigator can express the attack or the events he is looking for as a pattern in a logic formula using the syntax of the logic described above. Then, using a model checker, he can search the log model for such an attack or scenario of events in order to reconstruct the sequence of events that lead to the incidence under investigation. To further illustrate how this can be used, we have defined a general attack scenario which is divided into four phases. Under each phase we have grouped the possible events that an investigator would want to search for. The general attack scenario is as follows:

$$\mathit{Intrusion}\rightarrow\mathit{Compromise}\rightarrow\mathit{Misuse}\rightarrow\mathit{Withdrawal}$$

where each of *Intrusion*, *Compromise*, *Misuse* and *Withdrawal* is a certain sequence of events. The *Intrusion* sequence consists of the events that are captured during the time the attacker breached the system using different techniques such as buffer overflow, malicious codes, etc. Events in this category can be extracted from the logs of an intrusion detection system. Some patterns can be recognized as signs of intrusions such as when a service executing with administrative privileges starts a command shell can allude to a buffer overflow attack. This pattern is demonstrated below, where we use ϵ as a shorthand for replacing single terms with ⊛ and leaving sequence variables x_i as they are, to be consistent with syntax rules, also we use ≡ for syntactic equality:

$$\begin{aligned}&\langle x_1\cdot\mathit{ProcessBegin}\left(p_x,u_x,p_y,pv_{\mathrm{ADMIN}},e_x\right)\cdot x_2\cdot\mathit{ProcessBegin}\\&\quad\times\left(p_z,u_y,p_x,pv_z,e_1\right)\looparrowright\epsilon\rangle\mathtt{tt},\\&\quad\text{where }\ e_1\equiv\mathit{cmd}\cdot\mathit{exe}\end{aligned}$$

In the expression above the process p_x started with administrative privileges and then it called p_z which is the command shell.

The *Compromise* sequence consists of events pertaining to the preparation for malicious activity. During this phase the attacker is setting up what is needed to carry out the intended attack. In addition, the attacker tries to provide himself with convenient settings to return to the system at will. These include but are not limited to: Creating a user, changing group or password, installing or uninstalling a service, deactivating antivirus and firewall software, changing important files or registries, etc. These events can be constructed in different sequences, one sequence can be:

$$\langle x_1 \cdot \mathit{DeactFirewall}(u_x, dt_x) \cdot x_2 \cdot \mathit{DeactAnitvirus}(u_x, dt_y) \cdot x_3 \looparrowright \epsilon \rangle \mathtt{tt}$$

In the expression above the same user (u_x) deactivated the firewall followed by a deactivation of the antivirus, which is suspicious. Of course, it is possible to elaborate more complex pattern but we are just using expressions for demonstrative purposes.

The *Misuse* sequence represents the phase during which the attacker uses the compromised system for his own malicious purposes. The events considered for this phase are: local or remote login using the created accounts in the previous phases, opening a connection to the system, etc. As an example, consider the following property of a log sequence:

$$\langle x_1 \cdot \mathit{OpenConnection}(p_x, pt_x, proc_x, u_x, ac_{\mathrm{N\_ALLOWED}}, dt_x) \cdot x_2 \looparrowright \epsilon \rangle \mathtt{tt}$$

In the expression above, a user tried to open a connection that was refused, maybe because the user does not have the required privilege. A repeated pattern like this indicates suspicious behavior of this particular user.

The *Withdrawal* sequence represents the phase during which the attacker tries to cover his tracks cleaning the system from his malicious activities. The events may be: deleting files, uninstalling a service, clearing the log files, etc. An example for how a withdrawal may be expressed:

$$\langle x_1 \cdot \mathit{UnistallService}(s_x, u_x, dt_x) \cdot x_2 \cdot \mathit{ClearLogs}(u_x, dt_y) \cdot x_3 \looparrowright \epsilon \rangle \mathtt{tt}$$

So far we have shown how an attack scenario can be expressed using our proposed approach. To further elaborate on the usefulness of this approach we demonstrate two types of properties:

(A) Suspicious facts and invariant properties of a system
Suspicious facts are events that, when triggered, may be indications of an attack; such as, a trusted executable being executed by an untrusted object. Invariant properties are conditions that should be always satisfied. For example, for every logon there should be a logoff:

$$\begin{aligned} &\nu X \cdot \langle x_1 \cdot \mathit{logon}(u_x, uID_x, g_x, d_x, p_x, lt_x) \cdot x_2 \looparrowright \epsilon \rangle \mathtt{tt} \vee \langle x_1 \cdot \mathit{logoff} \\ &\quad \times (uID_x) \cdot x_2 \looparrowright \epsilon \rangle \mathtt{tt} \Rightarrow \langle x_1 \cdot \mathit{logon}(u_x, uID_x, g_x, d_x, p_x, lt_x) \cdot x_2 \cdot \mathit{logoff} \\ &\quad \times (u_x) \cdot x_3 \looparrowright x_1 \cdot x_2 \cdot x_3 \rangle X \end{aligned}$$

(B) Hypothesis properties
During the investigation, the investigator should be able to verify the admissability of a claim, a hypothesis or a speculation by expressing it in the form of a property and submitting the logic expression to a model checker. An example is the assumption that a file, when executed, will deactivate the firewall and install a service which will be used as a backdoor:

$$\begin{aligned} &\langle x_1 \cdot \mathit{ProcessBegin}(p_x, u_x, p_y, pv_x, e_x) \cdot x_2 \cdot \mathit{DeactFirewall} \\ &\quad \times (u_x, dt_x) \cdot x_3 \cdot \mathit{InstallService}(s_x, f_x, st_x, u_x, dt_y) \cdot x_4 \looparrowright \epsilon \rangle \mathtt{tt} \end{aligned}$$

5. Case study

To better comprehend the power of our approach, we will consider the following case: in an office environment using Windows-based machines, there are several hosts connected to a database server. An intrusion detection system is setup as a security measure for the server. The server can be accessed from outside the company so that the employees can connect to the server from client offices. Fig. 1 depicts the scenario.

The network administrator discovers that there was a denial of service attack on the server. The intrusion detection system (IDS) SNORT determines an SYN attack which caused the server to halt. An investigator is called upon to investigate this incident.

The investigator gathers the server network logs and starts searching for possible clues. Generally, when analyzing logs, an investigator will either scan through the logs manually using a simple editor which may provide some filtering capabilities, or create a script to serve his purpose. Using our methodology, the investigator only defines a pattern for a trace he wants to look for, and since all logs are combined in a single model, he does not need to consider logs separately. Moreover, if the investigator wishes to focus on a single log or a number of logs, he can use the $\langle\langle\rangle\rangle$ operator in the logic to do the job. In our particular case, let us suppose the system administrator has reported an SYN attack which is detected by SNORT and logged with SID = 1:526 (1). To locate the occurrence of this attack the administrator uses the following trace from the SNORT logs:

$$\begin{aligned} &\langle\langle \mathit{SNORT} \rangle\rangle \langle x_1 \cdot \mathit{Attack}(dt_x, sID_1, sIP_x, dIP_x) \cdot x2 \looparrowright \epsilon \rangle \mathtt{tt}, \\ &\quad \text{where } sID_1 = 1:526 \end{aligned}$$

Once the investigator finds the event that logged the attack, he examines it and realizes that this IP is within the same network of the company. Knowing the date and time of the first occurrence of the attack, the investigator looks for connections opened on the same date and time from that host to the server. Until now, the events have been mostly gathered from the IDS logs. The advantage and practicality of our model

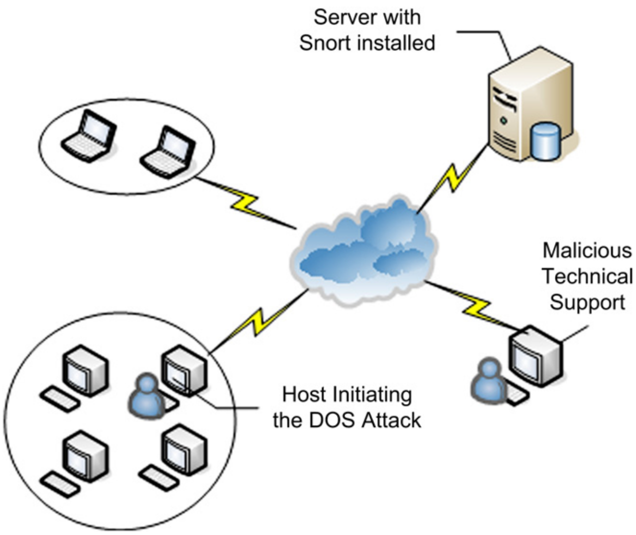


Fig. 1 – Case scenario.

which correlates all logs into a tree are shown here since the Windows system, application, and security logs are correlated along with the network logs. The logs being correlated into the same tree, we can define the patterns of traces that we are looking for using a logic formula and the model checker will search for them through the tree. However, to improve efficiency, we can specify which log we want to examine thus reducing the scope of our search. This can be done by adding ⟪*LogSource*⟫ right before the corresponding pattern or trace. *LogSource* in this case being SNORT (like in the pattern above), SEC (security log), SYS (system log), or APP (application log). It is possible to specify more then one log, e.g., ⟪*SYS, SEC*⟫, or all logs excluding one specific log, e.g., $\langle\langle \mathcal{L} \setminus SNORT\rangle\rangle$.

Getting back to our case, the event reflecting opening a connection can be found using:

$$\langle\langle SEC, SYS, APP\rangle\rangle\ \langle x_1 \cdot OpenConnection \times (p_x, pt_x, proc_{TCP}, uID_x, ip_1 ac_x, dt_1) \cdot x_2 \looparrowright \epsilon\rangle \mathtt{tt}$$

where ip_1 is the server's IP address and dt_1 is the time, these two values can be obtained from SNORT logs as explained above. Looking at the process that opened the connection (the variable p_x), the investigator finds out that p_x is a certain process p_1. The investigator now wants to know information about the launching of this process:

$$\langle\langle SEC\rangle\rangle\ \langle x_1 \cdot ProcessBegin(p_1, u_x, cp_x, pv_x, e_x) \cdot x_2 \looparrowright \epsilon\rangle \mathtt{tt}$$

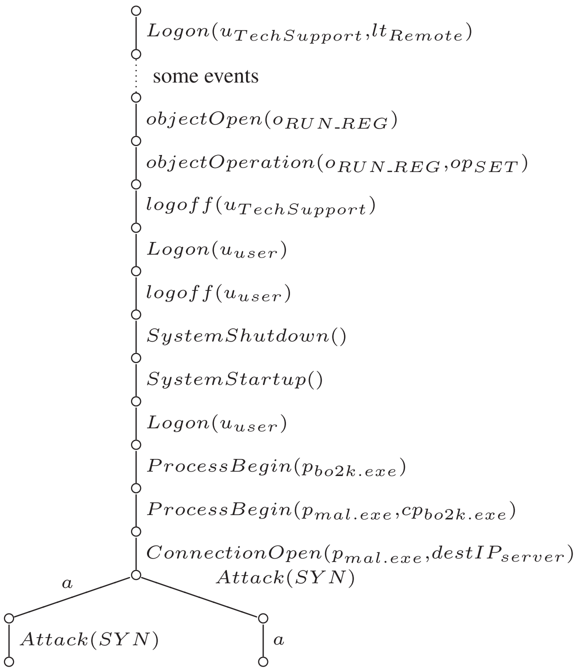
Assume the caller of p_1, i.e., cp_x, is p_2. Using the same formula as the one above but replacing p_1 with p_2 in the term *ProcessBegin*(…), the investigator can find out the name and executable file of p_2. It turns out to be `bo2k.exe`, which is the executable file of backOrifice2000; a backdoor. The next step is to track down sessions in which the back door has been executed which can be determined by the following:

$$\begin{aligned}&\langle\langle SEC\rangle\rangle\ \langle x_1 \cdot Logon(u_x, uID_x, g_x, d_x, p_x, lt_x) \cdot x_2 \cdot ProcessBegin\\ &\quad \times (p_x, u_x, cp_x, pv_x \cdot e_1) \cdot x_3 \cdot Logoff(u_x) \looparrowright \epsilon\rangle \neg\langle x_4 \cdot ProcessBegin\\ &\quad \times (p_y, u_x, cp_x, pv_x \cdot e_1) \cdot x_5 \cdot \circledast \cdot x_6 \looparrowright \epsilon\rangle \mathtt{tt}\\ &\quad \text{where } e_1 \equiv Bo2k \cdot exe\end{aligned}$$

The formula above looks for all sessions (a session is bounded by a logon and a logoff) in which the malicious executable e_1 is launched. Suppose in all of these sessions, at each logon of a certain user *mal* to the system, the back door program is executed, the investigator suspects that there is an entry in one of the start-up registry keys for `Bo2k.exe`. The following formula looks for the first session during which the run registry key has been modified before the first execution of the back door program.

$$\begin{aligned}&\langle\langle SEC\rangle\rangle\langle x_1 \cdot Logon(u_x, uID_x, g_x, d_x, p_x, lt_x) \cdot x_2 \cdot\\ &\quad ProcessBegin(p_x, u_x, cp_x, pv_x \cdot e_1) \cdot x_3 \cdot Logoff(u_x) \looparrowright \epsilon\rangle\\ &\quad (\neg\langle x_4 \cdot ProcessBegin(p_y, u_x, cp_x, pv_x \cdot e_1) \cdot x_5 \cdot \circledast \cdot x_6 \looparrowright \epsilon\rangle \mathtt{tt}\\ &\quad \wedge\langle x_6 \cdot logon(u_y, uID_y, g_y, d_y, p_y, lt_y) \cdot x_7 \cdot objectOpen\\ &\quad (o_1, p_z, pn_z, tp_z, lID_z) \cdot x_8 \cdot objectOperation(o_1, op_{SET}) \cdot x_9 \cdot logoff\\ &\quad (u_y) \cdot x_{10} \cdot \circledast \cdot x_{11} \looparrowright \epsilon > \mathtt{tt})\\ &\quad \text{where } o_1 \equiv RUN\_REG\end{aligned}$$

From this formula, the investigator will be able to know the session during which the malicious attacker has changed the registry key (*objectOperation*(o_1, op_{SET})) and the username of the attacker, which may turn out to be different from the user *mal* mentioned earlier. Such a situation is depicted by the following traces where, in terms, only the relevant arguments are listed:


$Logon(u_{TechSupport}, lt_{Remote})$
some events
$objectOpen(o_{RUN\_REG})$
$objectOperation(o_{RUN\_REG}, op_{SET})$
$logoff(u_{TechSupport})$
$Logon(u_{user})$
$logoff(u_{user})$
$SystemShutdown()$
$SystemStartup()$
$Logon(u_{user})$
$ProcessBegin(p_{bo2k.exe})$
$ProcessBegin(p_{mal.exe}, cp_{bo2k.exe})$
$ConnectionOpen(p_{mal.exe}, destIP_{server})$
a
$Attack(SYN)$
$Attack(SYN)$
a

The figure above shows two traces, where we assumed the event *a* appeared concurrently with *attack*, hence the existence of two branches. The idea we would like to express here is that although the SYN attack may be coming from the machine of the user u_{user}, a careful forensic analysis using our logic reveals that the registry key was modified by the user $u_{TechSupport}$, which is the real responsible.

6. Conclusion and future research

In this paper we proposed a model checking approach to the problem of formal analysis of logs. We model the log as a tree labeled by terms from a term algebra that represents the different actions logged by the logging system. The properties of such a log are expressed through the use of a logic that has temporal, modal, dynamic and computational characteristics. Moreover, the logic is provided by a sound and complete tableau-based proof system that can be the basis for verification algorithms. An example was given in which we presented ideas about properties of logs such as attack traces and invariant properties. We also demonstrated how our system can be used to reconstruct the sequence of events leading to SYN attack. A future research direction may be the development of an efficient model checking algorithm based on our proof system.

REFERENCES

Adi K, Debbabi M, Mejri M. A new logic for electronic commerce protocols. Int J Theor Comput Sci, TCS 2003; 291(3):223–83.

Cleaveland R. Tableau-based model checking in the propositional Mu-calculus. Acta Inform 1990;27(8):725–48.

Cuppens F. Managing alerts in a multi-intrusion detection environment. In: Proceedings of the 17th annual computer security applications conference, December 2001.

Cuppens F, Miege A. Alert correlation in a cooperative intrusion detection framework. In: Proceedings of the 2002 IEEE symposium on security and privacy, May 2003.

Debar H, Wespi A. Aggregation and correlation of intrusion-detection alerts. In: Recent advances in intrusion detection. LNCS 2212; 2001.

Gladyshev P, Patel A. Finite state machine approach to digital event reconstruction. Digit Investig J 2004;1(2).

Gladyshev P, Patel A. Formalising event time bounding in digital investigations. Digit Investig J 2005;4(2).

Hosmer C. Time lining computer evidence. Available from: <http://www.wetstonetech.com/f/timelining.pdf>; 1998 [Visited on May 22, 2007].

Julisch K. Clustering intrusion detection alarms to support root cause analysis. ACM Trans Inform Syst Secur Nov 2003;6(4): 443–71.

Kruse W, Heiser J. Computer forensics: incident response essentials. Boston, MA: Addison-Wesley; 2002 [Visited on May 22, 2007].

Leigland R, Krings AW. A formalization of digital forensics. Digit Investig J 2004;3(2).

Monroe K, Bailey D. System base-lining: a forensic perspective. Available from: <http://ftimes.sourceforge.net/Files/Papers/baselining.pdf>; 2003 [Visited on May 22, 2007].

Morin B, Debar H. Correlation of intrusion symptoms: an application of chronicles. In Proceedings of the sixth international conference on recent advances in intrusion detection (RAID'03), September 2003.

Morin B, Me L, Debar H, Ducasse M. M2D2: a formal data model for IDS alert correlation. In: Proceedings of the fifth international symposium on recent advances in intrusion detection (RAID 2002), 2002.

Ning P, Cui Y, Reeves DS. Constructing attack scenarios through correlation of intrusion alerts. In: Proceedings of the nineth ACM conference on computer and communications security, Washington, DC, November 2002. p. 245–54.

Peikari C, Chuvakin A. Security warrior. O'Reilly; 2004.

Porras P, Fong M, Valdes A. A mission-impact-based approach to INFOSEC alarm correlation. In: Proceedings of the fifth international symposium on recent advances in intrusion detection (RAID 2002); 2002.

Stallard T, Levitt K. Automated analysis for digital forensic science: semantic integrity checking. In: 19th annual computer security applications conference, Las Vegas, NV, USA, December 2003.

Staniford S, Hoagland J, McAlerney J. Practical automated detection of stealthy portscans. J Comput Secur December 2002;10(1/2):105–36.

Stephenson P. Modeling of post-incident root cause analysis. Int J Digit Evid 2003;2(2).

Snort – sourcefire Inc., <http://www.snort.org> [Visited on May 22, 2007].

Tenable Network Security, <http://www.nessus.org/> [Visited on May 22, 2007].

Templeton S, Levitt K. A requires/provides model for computer attacks. In: Proceedings of new security paradigms workshop, September 2000.

Valdes A, Skinner K. Probabilistic alert correlation. In: Proceedings of the fourth international symposium on recent advances in intrusion detection (RAID 2001); 2001.

Wechler W. Universal algebra for computer scientists. Springer; 1992.

Xu D. Alert correlation through triggering events and common resources, <http://citeseer.ist.psu.edu/742919.html> [Visited on May 22, 2007].

Yegneswaran V, Barford P, Jha S. Global intrusion detection in the domino overlay system. In: Proceedings of the 11th annual network and distributed system security symposium (NDSS'04), Feburary 2004.

Ali Reza Arasteh is a master student in the Department of Computer Science at Concordia University. He holds B.Sc. degree from Amirkabir University, Tehran, Iran. His main research interests are forensics memory analysis, log analysis and intrusion detection systems.

Mourad Debbabi is a Full Professor and the Associate Director of the Concordia Institute for Information Systems Engineering at Concordia University. He is also a Concordia Research Chair Tier I in Information Systems Security. He holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published several research papers in international journals and conferences on computer security, cyber forensics, formal semantics, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory and specification and verification of safety-critical systems. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Corporate Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France.

Assaad Sakha is a M.Sc. student at Concordia Institute of Information System Engineering, Concordia University working in the field of Digital Forensics. The main area of his research is forensic log analysis. He obtained his B.Sc. in Computer Information Systems at Notre Dame University, Lebanon. In addition he is a member of the Information System Audit and Control Association (ISACA).

Mohamed Saleh is a Ph.D. student at Concordia University working in IT security. His main research interest is in the use of formal methods in the specification and verification of security protocols. He has also conducted some research work in application security, namely using Java.