Available online at www.sciencedirect.com
ScienceDirect
Knowledge-Based Systems 20 (2007) 671–682
Knowledge-Based SYSTEMS
www.elsevier.com/locate/knosys

Forensic analysis of logs: Modeling and verification ☆

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

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

Received 15 March 2007; accepted 1 May 2007
Available online 6 May 2007

Abstract

Information stored in logs of a computer system is of crucial importance to gather forensic evidence of investigated actions or attacks against the system. 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. The set of logs of a certain system 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 a term algebra. The signature of the algebra is carefully chosen to include all relevant information necessary to conduct the analysis. Properties of the model 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. In order to illustrate the proposed approach, the Windows auditing system is studied. The properties that we capture in our logic include invariant properties of a system, forensic hypotheses, and generic or specific attack signatures. Moreover, we discuss the admissibility of forensics hypotheses and the underlying verification issues.

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

1. Introduction

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 log analysis problem.

One of the most common sources of evidence that an investigator should analyze is logs 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 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.

☆ This research is done in collaboration between the Computer Security Laboratory at Concordia University and Bell Canada under a PROMPT-Québec grant.
* Corresponding author.
E-mail addresses: m_saleh@ciise.concordia.ca (M. Saleh), a_arast@ciise.concordia.ca (A.R. Arasteh), a_sakha@ciise.concordia.ca (A. Sakha), debbabi@ciise.concordia.ca (M. Debbabi).

0950-7051/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2007.05.002

The main 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 [15], 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 [24] 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 [3] involves searching through a database of common vulnerabilities and investigating the case according to the related past and known vulnerabilities. The timeline analysis approach [11] 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 [22] 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.

In [9], Pavel Gladyshev 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 [10,14], 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 [28]:

- Similarity based approaches [6,12,23,27], which group the alerts according to the similarity between alert attributes.
- Predefined attack scenario based approaches [8,16], which detect attacks according to well defined attack scenarios. However, they cannot discover novel attack scenarios.
- Pre/post condition based approaches [7,18,25] that match the post-condition of an attack to the pre-conditions of another attack.
- The multiple information sources based approaches[17, 20,26] that are concerned with distritbuted 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 [19] and operating system-specific analysis [13].

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 [4]. 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. First, 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.

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 application of our approach to the Windows logging system. Finally, conclusions and future work are discussed in Section 5.

2. Modeling approach

We begin by presenting some definitions, then we explain our model.

2.1. Basic definitions

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** : $\mathbb{F} \rightarrow \mathbb{N}$ 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 $f : S_1 \times S_2 \times \cdots \times S_n \rightarrow S$. 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 $\mathbb{T}_\Sigma(S,X)$ of free terms of sort S inductively as follows (the symbol '$\Rightarrow$' is for logic implication):

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

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

where each t_i is a term of sort S_i

The set $\mathbb{T}_\Sigma(X)$ of free terms over the signature Σ is defined as $\bigcup_{S \in \mathbb{S}} \mathbb{T}_\Sigma(S,X)$. The set of ground terms $\mathbb{T}_\Sigma \subseteq \mathbb{T}_\Sigma(X)$ 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 $\mathbb{A} = \bigcup_{S \in \mathbb{S}} \mathbb{A}_S$. Also for each operator symbol $f : S_1 \times S_2 \times \cdots \times S_n \rightarrow S$, $\mathcal{A}$ assigns a function $f_\mathcal{A} : A_{S_1} \times A_{S_2} \times \cdots \times A_{S_n} \rightarrow A_S$, $\mathbb{F}_\mathcal{A}$ 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, $h : \mathcal{A} \rightarrow \mathcal{B}$ is a called a Σ-homomorphism iff:

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

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 $f_{\mathcal{T}_\Sigma(X)} : \mathbb{A}_{S_1} \times \mathbb{A}_{S_2} \times \cdots \times \mathbb{A}_{S_n} \rightarrow \mathbb{A}_S$ are defined as:

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

$$f_{\mathcal{T}_\Sigma(X)}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$$

where, $t_i \in \mathbb{A}_{S_i}$, i.e., each term is taken to be its own interpretation

The term algebra $\mathcal{T}_\Sigma$ can be obtained from $\mathcal{T}_\Sigma(X)$ by removing any terms that contain variables. A substitution $\theta_x : \mathbb{X}_S \rightarrow \mathbb{T}_\Sigma(S,X)$ 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:

$$\theta_x : \mathbb{X}_S \rightarrow \mathbb{T}_\Sigma(S,X) \text{ is extended to}$$

$$\tilde{\theta}_x : \mathbb{T}_\Sigma(S,X) \rightarrow \mathbb{T}_\Sigma(S,X)$$

$$\tilde{\theta}_x(c) = c \text{ where } c \text{ is a constant}$$

$$\tilde{\theta}_x(f(t_1, t_2, \ldots, t_n)) = f(\tilde{\theta}_x(t_1), \tilde{\theta}_x(t_2), \ldots, \tilde{\theta}_x(t_n))$$

Usually we write θ_x for $\tilde{\theta}_x$. As a simple example, consider the signature $\langle \{\mathsf{User}, \mathsf{File}, \mathsf{Bool}\}, \{\mathit{CreateFile}, \mathit{DeleteFile}\} \rangle$, where $\mathit{CreateFile} : \mathsf{User} \rightarrow \mathsf{File}$, and $\mathit{DeleteFile} : \mathsf{User} \times \mathsf{File} \rightarrow \mathsf{Bool}$. Now suppose we have the constants u_1, u_2 of the sort User and l_1, l_2 of the sort File, so we can have the terms $\mathit{CreateFile}(u_1)$, $\mathit{DeleteFile}(u_1, \mathit{CreateFile}(u_1))$.

2.2. Model

In this section, we propose a model for logs that will be later used for the analysis. Analyzing the model amounts to a model checking problem where we express desired model properties using a logic that will be detailed below. The tableau-based proof system of the logic can be further developed into a model checking algorithm. We view a log as a sequence of log entries, each entry consists of a certain term $t \in \mathbb{T}_\Sigma$. 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. For instance, in a computer system, one log can monitor operating system events such as file backup while another log can monitor security related events. We assume that, in the presence of more than one log in the system, all logs are handled in parallel, i.e., there is no synchronization between logs. 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. In order to illustrate this, assume we have a sequence of actions $a.b$ in one log and another sequence $c.d$ in another. 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 having the following traces $\{a, a.c, a.c.b, a.c.d, a.c.b.d, a.c.d.b\}$, i.e., we considered the two possible interleavings of the concurrent events b and d. The same 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 like we described above.

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 [4], 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, 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 has all the expressive power we need for this task including counting properties [4].

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:

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

Here a represents a term of the term algebra ($t \in \mathbb{T}_\Sigma(X)$) and has the form t or $\lceil t \rceil$, where the symbol $\lceil t \rceil$ means a term containing t. For instance $\lceil F \rceil$ could be *DeleteFile*(F, U). In other words, the term $\lceil t \rceil$ is defined as $f(t_1, t_2, \ldots, t_n)$ such that $\exists t_i \,.\, ((t_i = t) \vee (t_i = \lceil t \rceil))$. 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 variables x of the message algebra. The set of sequence variables and the set of terms in r are written $var(r)$ and $trm(r)$, respectively. Moreover, for any pattern r, the symbol $r|_i$ represents the sequence variable or term 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 sequence variables x_r in a sequence pattern to sequences of terms, this is not to be confused with the substitution θ_m that maps variables inside messages into terms of $T_\Sigma(X)$.

We define the predicate $\mathbf{match}(\sigma, r, \theta_m, \theta_r)$, which is true when a sequence $\sigma = s_1.s_2 \ldots s_n$ in the log tree *matches* a pattern r, ϵ is the empty sequence:

$$\mathbf{match}(\epsilon, \epsilon, \theta_m, \theta_r) = true$$
$$\mathbf{match}(\sigma, \epsilon, \theta_m, \theta_r) = false \ \text{ if } s \neq \epsilon$$
$$\mathbf{match}(\sigma, a.r, \theta_m, \theta_r) = (s_1 = a\theta_m) \wedge \mathbf{match}(s_2 \ldots s_n, r, \theta_m, \theta_r)$$
$$\mathbf{match}(\sigma, x_r.r, \theta_m, \theta_r) = \exists j \leqslant n \,.\, (x_r\theta_r = s_1 \ldots s_j) \wedge \mathbf{match}(s_{j+1} \ldots s_n, r, \theta_m, \theta_r)$$

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:

$$\theta(\epsilon) = \epsilon$$
$$\theta(a.r) = \theta_m(a).\theta(r)$$
$$\theta(x_r.r) = \theta_r(x_r).\theta(r)$$

In the above definitions, we 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 $\mathbf{match}(\sigma, r, \theta)$.

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

$$\varphi ::= Z | \neg\varphi | \varphi_1 \wedge \varphi_2 | [r_1 \looparrowright r_2]\varphi | \nu Z.\varphi \tag{2}$$

We require the following two syntactic conditions:

- In $[r_1 \looparrowright r_2]$, $\forall i \,.\, (r_1|_i \in var(r_1) \iff r_1|_i = r_2|_i) \wedge (r_1|_i \in trm(r_1) \iff r_1|_i = r_2|_i \vee r_2|_i = \circledast)$.
- In $\nu Z.\varphi$, 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., the dummy symbol $\circledast$ stands for one single event that has been removed from the sequence. This condition is necessary to ensure that $r_1\theta$ and $r_2\theta$ have the same length since each event that we removed from r_1 is replaced by $\circledast$. Hence, we can replace $r_1\theta$ by $r_2\theta$ and still get a tree, which is written $L' = L[r_2\theta / r_1\theta]$, where L is the tree model of the log. The second condition is necessary for the semantic interpretation function as will be explained in the semantics section.

Intuitively, the formula $[r_1 \looparrowright r_2]\varphi$ is true if there is a sequence in the log tree that matches r_1 and when modified to match r_2 will satisfy φ. The rest of the formulas have their usual meaning in modal μ-calculus [5].

3.2. Semantics

A formula in the logic is interpreted over a log tree. Given a certain log tree L, a substitution θ, and a 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 $S \subseteq L$.

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

From the semantic equations above it can be seen that the meaning of the recursive formula $\nu Z.\varphi$ 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 [5] and hence has a greatest fixpoint.

We use the following shorthand notations:

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

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

Lemma 3.1. $[\![\varphi[\psi/Z]]\!]^L_e = [\![\varphi]\!]^L_{e[Z\mapsto[\![\psi]\!]^L_e]}$

The proof is done by structural induction over φ.

Proof. 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: $\varphi = \nu Z.\varphi'$ and X is free in φ

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

By induction hypothesis $\left([\![\varphi'[\psi/X]]\!]^L_{e[Z\mapsto S]} = [\![\varphi']\!]^L_{e[Z\mapsto S][X\mapsto[\![\psi]\!]^L_{e[Z\mapsto S]}]}\right)$:

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

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

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

Case: $\varphi = [r_1 \looparrowright r_2]\varphi'$

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

By induction hypothesis:

$$
\begin{aligned}
&[\![\varphi[\psi/Z]]\!]^L_e = \left\{\sigma \in L | \forall\theta\,.\,\mathbf{match}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi']\!]^{L'}_{e[Z\mapsto[\![\psi]\!]^L_e]}\right\}\\
&[\![\varphi[\psi/Z]]\!]^L_e = [\![\varphi]\!]^L_{e[Z\mapsto[\![\psi]\!]^L_e]}\Big\} \quad \square
\end{aligned}
$$

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

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

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

For any set of sequences $S \subseteq L$, let $S^c = L \setminus S$. By De Morgan laws, for any two sets A and B:

$$(A \cap B)^c = A^c \cup B^c, (A \cup B)^c = A^c \cap B^c, A \subseteq B \ \Rightarrow\ B^c \subseteq A^c.$$

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

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

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

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, $S \cap (L \setminus S) = \emptyset$. 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 [5] $\prec_I$ as:

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

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 sequnce σ satisfies the property φ. The set H contains elements of the form $\sigma : \nu Z.\varphi$ 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 $H \Rsh \nu Z.\varphi = \{\sigma \in L | \sigma : \nu Z.\varphi \in H\}$. The use of H, and b will be apparent after we state the rules of the proof system:

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

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

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

$$R_{[]} \quad \frac{H, b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi}{\xi_1 \quad \xi_2 \quad \ldots \xi_n} \qquad Condition$$

$$\text{Where, } H' = H \setminus \{\sigma' : \Gamma \mid \nu Z.\varphi \prec \Gamma\}$$

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

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

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 $H,b \vdash \sigma \in \nu Z.\varphi$ is determined only by subformulas of φ [5]. The fourth rule takes care of formulas matching sequences to patterns. 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 $\sigma \in [\![Z]\!]^L_e$.
- $H, \neg \vdash \sigma \in Z$ and $\sigma \notin [\![Z]\!]^L_e$.
- $H, \epsilon \vdash \sigma \in \nu Z.\varphi$ and $\sigma : \nu Z.\varphi \in H$.
- $H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\{\sigma \in L | \exists\theta.\ \mathbf{match}(\sigma, r_1, \theta)\} = \emptyset$.

In the next section, we prove that any formula φ has a finite tableau and that the proof system is sound and complete.

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:

$$[\![\nu Z.\varphi]\!]^{L,H}_e = \left(\nu [\![\varphi]\!]^{L,H}_{e[Z \mapsto S \cup S']}\right) \cup S'$$
$$\text{where, } S' = H \Rsh \nu Z.\varphi$$

In the equation, the greatest fixpoint operator is applied to a function $f(S) = [\![\varphi]\!]^{L,\theta}_{e[Z \mapsto S]}$ 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 provided in Appendix.

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 $R_{\neg}$, $R_{\wedge}$, $R_{[]}$, and R_{ν}. The application of the first three rules results in shorter formulas, while the application of the $R\nu$ results in larger hypothesis sets H. The proof shows that shortening a formulas 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, $\sigma \in [\![\varphi]\!]^{L,H}_e$*

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 $\sigma \in L$, $\sigma \in [\![\varphi]\!]^{L,H}_e$, 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. Windows logging system

To demonstrate the ideas discussed in the previous sections, we consider the Windows logging system as an exam-

ple of an operating system that is popular and hence the target of manu attacks. First, we present the logging system, then we discuss the modeling process and the properties we are able to express in our logic. The overall functionality of the windows logging system [21] is depicted in Fig. 1. Logs are created by the audit process, which monitors the behavior of applications. It creates logs for the defined trigger events. Log entries can also be generated by applications referring to filters that specify what to log. Triggers and filters are defined by group policies which are created and edited using the group policy Microsoft Management Console (MMC). Logs can be examined using the MMC Viewer or the Event Viewer.

The three main log types on the Windows operating system are: System logs, application logs, and security logs. In this paper we focus on the security log, which contains the most pertinent events for a forensic analysis. By reviewing the security logs, information such as the following may be derived: Usage of specific applications, modifications to registry keys, attempts to log on (successful or not), changes to user permissions, access to files by users, creation and termination of processes, etc. It should be noted that if the audit policy is not set up properly, some crucial events will be missing. For this reason, we have identified a set of security critical files and registry keys that should be audited to pinpoint malicious activities. Moreover, a set of trusted executables are defined, which are the only executables that are authorized to modify these critical resources. To be able to analyze the events that are related to these activities and files, we have configured a typical Windows system to log the related events.

4.1. Modeling windows log

In this section, we propose a model for the windows logging system. The model follows the guidelines in Section 2 in that it is a tree labeled by terms of a term algebra that represents different logged actions. We begin by listing the sorts of our algebra and the notation used, then we present different operations of the algebra along with their signatures. Operations are grouped according to the type of actions they represent. The table below shows the sorts of our algebra and the symbols we use for constants and variables of each sort.

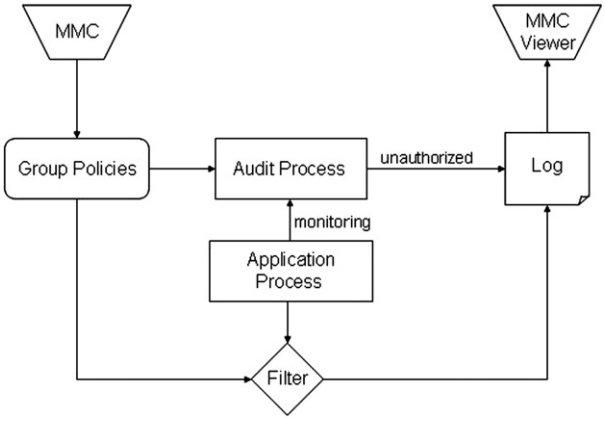


Fig. 1. Windows logging framework.

Sorts	Constants	Variables
user	$u_1, u_2, \ldots, u_n$	$u_x, u_y, u_z, \ldots$
file	$f_1, f_2, \ldots, f_n$	$f_x, f_y, f_z, \ldots$
process	$p_1, p_2, \ldots, p_n$	$p_x, p_y, p_z, \ldots$
object	$o_1, o_2, \ldots, o_n$	$o_x, o_y, o_z, \ldots$
operation	op_{READ}, op_{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}	$pv_x, pv_y, pv_z, \ldots$
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	$r_1, r_2, \ldots, r_n$	$r_x, r_y, r_z, \ldots$
Access	$ac_{NOTIFY}, ac_{ALLOWED}$	$ac_x, ac_y, ac_z, \ldots$
port	pt_{PORT_NO}	$pt_x, pt_y, pt_z, \ldots$
protocol	$proc_{UDP}, proc_{TCP}, \ldots, u_n$	$proc_x, proc_y, proc_z, \ldots$
bool	Tr, Fl	$b_x, b_y, b_Z, \ldots$

In the following, we present a list of operations categorized based on the context they are related to. In most cases, the signature is self-explanatory and in case of ambiguity we will provide explanation about the nature of arguments of the operations.

4.1.1. Process and object related actions

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

$$ProcessBegin : \text{process} \times \text{user} \times \text{caller} \times \text{priviledge} \times \text{executable} \rightarrow \text{bool}$$
$$ProcessExit : \text{process} \times \text{user} \rightarrow \text{bool}$$

Here, executable is the sort of the executable file of the process. 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:

$$ObjectOpen : \text{object} \times \text{user} \rightarrow \text{bool}$$
$$ObjectClose : \text{object} \rightarrow \text{bool}$$
$$ObjectOperation : \text{object} \times \text{operation} \rightarrow \text{bool}$$

It should be noted that not all accesses to files and objects are logged due to the huge amount of logs that would be generated. This is why we assumed a predefined set of important files and registry keys to be audited. Moreover, a set of trusted executables are specified to be allowed to modify these critical system resources.

4.1.2. User actions

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

$$Logon : \text{user} \rightarrow \text{bool}$$
$$Logoff : \text{username} \rightarrow \text{bool}$$

4.1.3. Miscellaneous

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

$ScheduleTask : \mathsf{user} \times \mathsf{process} \rightarrow \mathsf{bool}$

$InstallService : \mathsf{service} \times \mathsf{file} \times \mathsf{service\_type} \times \mathsf{username} \rightarrow \mathsf{bool}$

$UninstallService : \mathsf{service} \times \mathsf{user} \rightarrow \mathsf{bool}$

$DeactivateAntiVirus : \mathsf{user} \rightarrow \mathsf{bool}$

$DeactivateFirewall : \mathsf{user} \rightarrow \mathsf{bool}$

$OpenConnection : \mathsf{process} \times \mathsf{port} \times \mathsf{protocol} \times \mathsf{user} \times \mathsf{access} \rightarrow \mathsf{bool}$

$ClearLogs : \mathsf{user} \rightarrow \mathsf{bool}$

4.2. Expressing properties and scenarios

An important use of the logic is the ability to detect an attack based on the general characterizations of attacks. To 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 would help us identify an attack. The general attack scenario is as follows:

$$Intrusion \rightarrow Compromise \rightarrow Misuse \rightarrow Withdrawal$$

In other words,an attacked system may have a log that satisfies the following property:

$$\langle x_1.Intrusion.x_2.Compromise.x_3.Misuse.x_4.Withdrawal.x_5 \looparrowright \varepsilon \rangle \mathtt{t}\,\mathtt{t}$$

where each of *Intrusion, Compromise, Misuse* and *Withdrawal* is a sequence of events pertaining to the corresponding phase pattern. In the expression above, we use ε as a shorthand for replacing single terms with $\circledast$ and leaving sequence variables x_i as they are, to be consistent with syntax rules. 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 $\equiv$ for syntactic equality:

$$\langle x_1.ProcessBegin(p_x, u_x, p_y, pv_{ADMIN}, e_x).x_2.$$
$$ProcessBegin(p_z, u_y, p_x, pv_z, e_1) \looparrowright \varepsilon \rangle \mathtt{t}\,\mathtt{t}$$
$$where\ e_1 \equiv cmd.exe$$

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. The events in the *Compromise* phase include but are not limited to: Creating a user, changing a group, 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 as follows:

$$\langle x_1.DeactivateFirewall(u_x).x_2.DeactivateAnitvirus(u_x).x_3 \looparrowright \varepsilon \rangle \mathtt{t}\,\mathtt{t}$$

In the expression above the same user 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:

$$\langle x_1.OpenConnection(p_x, pt_x, proc_x, u_x, ac_{NOT\_ALLOWED}).x_2 \looparrowright \varepsilon \rangle \mathtt{t}\,\mathtt{t}$$

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 the particular user.

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

$$\langle x_1.UnistallService(s_x, u_x).x_2.ClearLogs(u_x).x_3 \looparrowright \varepsilon \rangle \mathtt{t}\,\mathtt{t}$$

So far we have shown how an attack scenario can be expressed using our proposed approach. However, this is not the only benefit that our logic provides. Assuming that an investigator wants to look for some specific events or actions. This is possible using our approach by defining formalized properties for a system. We classify three types of properties:

(1) 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.
(2) Signature properties: Signatures of a virus, worm, trojan horse or any malicious code.
(3) 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.

4.2.1. Suspicious facts and invariant properties of a system

Suspicious facts are the clues that an investigator looks for when inspecting a crime scene. The list of such facts is

rather long but we can mention two of them as an example: A trusted executable has been changed, a modification to a security critical resource from an untrusted executable. Moreover, we can also express invariant properties of a system. By invariant properties we mean properties that do not change under normal circumstances. We define, for instance, the invariant property that for each log on there should be a log off:

$$\nu X.\langle x_1.logon(u_x).x_2 \looparrowright \varepsilon\rangle \mathtt{tt} \vee \langle x_1.logoff(u_x).x_2 \looparrowright \varepsilon\rangle \mathtt{tt} \Rightarrow \langle x_1.logon(u_x).x_2.logoff(u_x).x_3 \looparrowright x_1.x_2.x_3\rangle X$$

4.2.2. Signature properties

The signature properties we discuss here are signatures of a virus, worm or other malicious codes. The signatures can be used to look for the traces of the related incidences through the logs. For illustration, we have chosen the most common worm and trojan horse and retrieved the corresponding signatures from the Sophos website [2]. We express the provided signatures using our logic as follows:

4.2.2.1. Troj/Haxdoor-CA Signature. Troj/Haxdoor-CA is a Windows platform trojan horse. It provides a backdoor server by running continuously in the background. Thus allowing a remote intruder to gain access of the infected machine. Troj/Haxdoor-CA installs some files and creates some registry keys under the following hives:

- Reg_Hive1 ≡ HKLM\SOFTWARE\Microsoft\ Windows NT\CurrentVersion\Winlogon\Notify\vistax
- Reg_Hive2 ≡ HKLM\SYSTEM\CurrentControlSet \Services\vistaj\

The trojan horse also disables the firewall and installs a service. These characteristics can be expressed as follows:

$$\langle x_1.DeactivateFirewall(u_x).x_2.InstallSevice(s_x,f_x,st_x,u_x).x_3 \looparrowright \varepsilon\rangle \mathtt{tt} \wedge (\langle x_1.ObjectOpen(o_1,u_x).x_2.ObjectOperation(o_1,op_{WRITE}).x_3 \looparrowright \varepsilon\rangle \mathtt{tt} \wedge (\langle x_1.ObjectOpen(o_2,u_x).x_2.ObjectOperation(o_2,op_{WRITE}).x_3 \looparrowright \varepsilon\rangle \mathtt{tt}$$

where $o_1 \equiv Reg\_Hive_1$ and $o_2 \equiv Reg\_Hive_2$.

4.2.3. Hypothesis properties

Suppose a cyber forensics analyst suspects that an attacker managed to have a malicious file on the victim's machine. When that file is executed it deactivates the firewall and installs a service which will be used as a backdoor. The analyst should be looking for a sequence of events that represents this scenario, which can be expressed in our logic as:

$$\langle x_1.ProcessBegin(p_x,u_x,p_y,pv_x,e_x).x_2. DeactivateFirewall(u_x).x_3.InstallService(s_x,f_x,st_x,u_x).x_4 \looparrowright \varepsilon\rangle \mathtt{tt}$$

Then, by applying a verification algorithm, the analyst will be in a position to validate her hypothesis.

4.3. Example

To demonstrate the use of our approach, we will consider the following simple 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. The network administrator discovers that there was a denial of service attack on the server. The Intrusion Detection System (IDS) SNORT determines a 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. In our particular case, let's suppose the system administrator has reported a 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:

$$\langle x_1.Attack(dt_x,sID_1,sIP_x,dIP_x).x2 \looparrowright \varepsilon\rangle \mathtt{tt}$$
$$Where\ sID_1 = 1:526$$

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 which correlates all logs into a tree is 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.

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

$$\langle x_1.OpenConnection(p_x,pt_x,proc_{TCP},uID_x,ip_1ac_x,dt_1).x_2 \looparrowright \varepsilon\rangle \mathtt{tt}$$

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:

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

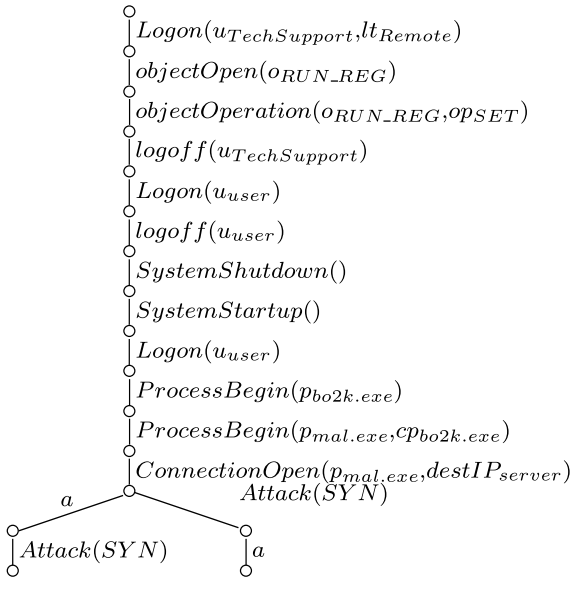
Assume the caller of p_1, i.e., cp_x, is p_2. Using the same formula as the one above but replacing p_1 with p_2 in the term *ProcessBegin*(...), the investigator can find out the name and executable file of p_2. It turns out to be `bo2k.exe`, which is the executable file of backOrifice2000; a backdoor.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

$$\langle x_1.Logon(u_x,uID_x,g_x,d_x,p_x,lt_x).x_2.ProcessBegin(p_x,u_x,cp_x,pv_x.e_1).x_3.$$
$$Logoff(u_x) \looparrowright \varepsilon\rangle \neg\langle x_4.ProcessBegin(p_y,u_x,cp_x,pv_x.e_1).x_5.\circledast.x_6 \looparrowright \varepsilon\rangle \mathtt{tt}$$
Where: $e_1 \equiv$ `Bo2k.exe`

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.

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

From this formula, the investigator will be able to know the session during which the malicious attacker has changed the registry key (since the event representing this change is $objectOperation(o_1, op_{SET})$) 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 algebraic terms, only the relevant arguments are listed:



The Figure above shows two traces, where we assumed the event *a* appeared concurrently with *attack*, hence the existence of two branches. We notice 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.

5. Conclusion

We proposed a model checking approach to the problem of formal analysis of logs. We modeled 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. The Windows logging system was studied as a use case through 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 express signatures of malicious code such as viruses and worms. A future research direction may be the development of an efficient model checking algorithm based on our proof system.

Appendix A. Proof of Finiteness

We will need the following definitions:

Definition A.1 (*Closure*)**.** The closure $CL(\varphi)$ of a formula φ is a set of formulas that is defined as:

$$CL(Z) = \{Z\}$$
$$CL(\neg\varphi) = \{\neg\varphi\} \cup CL(\varphi)$$
$$CL(\varphi_1 \wedge \varphi_2) = \{\varphi_1 \wedge \varphi_2\} \cup CL(\varphi_1) \cup CL(\varphi_2)$$
$$CL([r_1 \looparrowright r_2]\varphi = \{[r_1 \looparrowright r_2]\varphi\} \cup CL(\varphi)$$
$$CL(vZ.\varphi) = \{vZ.\varphi\} \cup CL(\varphi[vZ.\varphi/Z])$$

Definition A.2 (*Size*)**.** The size $|\varphi|$ of a formula is a positive integer:

$$|Z| = 1$$
$$|\neg\varphi| = 1 + |\varphi|$$
$$|\varphi_1 \wedge \varphi_2| = 1 + |\varphi_1| + |\varphi_2|$$
$$|[r_1 \looparrowright r_2]\varphi| = 1 + |\varphi|$$
$$|vZ.\varphi| = 1 + |\varphi|$$

Definition A.3 (*H*-Ordering)**.** Between hypothesis sets, we define the relation $\sqsubseteq_\varphi$, relative to a formula φ, where $\mathcal{H}$ is the set of all hypothesis sets:

$\sqsubseteq_Z = \mathcal{H} \times \mathcal{H}$
$\sqsubseteq_{\neg\varphi'} = \sqsubseteq_{\varphi'}$
$\sqsubseteq_{\varphi_1 \wedge \varphi_2} = \sqsubseteq_{\varphi_1} \cap \sqsubseteq_{\varphi_2}$
$\sqsubseteq_{[r_1 \looparrowright r_2]\varphi'} = \sqsubseteq_{\varphi'}$
$\sqsubseteq_{vZ.\varphi'} = \{(H_1, H_2) \in \sqsubseteq_{\varphi'} | (H_2, H_1) \in \sqsubseteq_{\varphi'} \Rightarrow H_1 \Cap vZ.\varphi' \subseteq H_2 \Cap vZ.\varphi'\}$
$H_1 =_\varphi H_2 \iff H_1 \sqsubseteq_\varphi H_2 \wedge H_2 \sqsubseteq_\varphi H_1$
$H_1 \sqsubset_\varphi H_2 \iff H_1 \sqsubseteq_\varphi H_2 \wedge H_1 \neq_\varphi H_2$
$\forall \varphi' \in CL(\varphi). H_1 \sqsubseteq_{\varphi'} H_2 \Rightarrow H_1 \trianglelefteq H_2$
$H_1 \equiv_\varphi H_2 \iff H_1 \trianglelefteq H_2 \wedge H_2 \trianglelefteq H_1$
$H_1 \triangleleft H_2 \iff H_1 \trianglelefteq H_2 \wedge H_2 \not\trianglelefteq H_1$

From the definitions above, it is clear that $\forall \varphi, \varphi'. \varphi' \prec_1 \varphi \Rightarrow \sqsubseteq_\varphi \subseteq \sqsubseteq_{\varphi'}$. The following results about $\sqsubseteq_\varphi$ are proved in [5]:

- $\sqsubseteq_\varphi$ is reflexive and transitive, i.e., a preorder.
- $H \sqsubset_{vZ.\varphi} H' \cup \{\sigma : vZ.\varphi\}$, where $H' = H \backslash \{\sigma' : \Gamma |\ vZ.\varphi \prec \Gamma\}$.
- $\sqsubset_\varphi$ has no infinite ascending chains.

Definition A.4 (*Sequent ordering*). For any two sequents $\tau_1 = H_1, b_1 \vdash \sigma_1 \varphi_1$ and $\tau_2 = H_2, b_2 \vdash \sigma_2 \varphi_2$, such that $\varphi_2 \in CL(\varphi_1)$, the relation $\tau_1 < \tau_2$ holds, whenever one of the following holds:

- $H_1 \triangleleft_{\varphi_2} H_2$.
- $H_1 \equiv_{\varphi_2} H_2 \wedge |\varphi_2| < |\varphi_1|$.

Lemma A.1. *The sequent ordering relation $<$ has no infinite ascending chain.*

Proof. Suppose we start by $\tau_0 = H_0, b_0 \vdash \sigma \in \varphi_0$, the chain $\tau_0 < \tau_1 < tau_2 \ldots$ cannot ascend infinitely, the proof follows form the facts that $|CL(\varphi_0)|$ is finite, $\triangleleft_{\varphi_0}$ has no infinite ascending chains, and $\forall \varphi' \in CL(\varphi)$. $|\varphi'| < |\varphi|$. □

Theorem A.1 (*Finiteness*). *For any sequent $\tau = H, b \vdash \sigma \in \varphi$ there exists a finite number of finite tableaux.*

Proof. The proof is by well-founded induction on the inverse of the sequent ordering relation, since $<$ has no infinite ascending chain, then $<^{-1}$ has no infinite descending chain.

Induction hypothesis: for all τ' such that $\tau < \tau'$, τ' has a finite number of finite tableaux.

Required: for any τ, the applicable rule of the tableau will produce a finite number of τ', where $\tau < \tau'$.

We consider here two cases: $\tau = H, b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\tau = H, b \vdash \sigma \in vZ.\varphi$ since the other cases are straightforward.

Case 1: $\tau = H, b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$

The only applicable rule is $R_{[]}$, it generates sequents of the form $\tau_i = H_i, b_i \vdash \sigma_i \in \varphi$, where $i \in \mathbb{I} \subseteq \mathbb{N}$, $\mathbb{N}$ is the set of natural numbers. For all i $H_i = H$, moreover $\varphi \in CL([r_1 \looparrowright r_2]\varphi)$ and $|\varphi| < |[r_1 \looparrowright r_2]\varphi|$, hence $\tau < \tau_i$. The set $\mathbb{I}$ is determined by the set $\Theta = \{\theta_i | \textbf{match}(r_1, \sigma, \theta_i) = \mathtt{tt}\}$, where $|\mathbb{I}| = |\Theta|$. Therefore we need to prove that Θ is finite, which follows from the fact that σ is finite since we are dealing with finite models. So we can write $i \in \{0, 1, \ldots n\}$. The other condition of the rule is that $b_1 \times b_2 \times \cdots \times b_n = b$, the number of combinations if finite ($2^{|\mathbb{I}|}$) and each of these combinations produce a different tableau.

Case 2: $\tau = H, b \vdash vZ.\varphi$

The only applicable rule is R_v, which produces only one sequent $\tau' = H', b \vdash \varphi[vZ.\varphi/Z]$, using the results about $\sqsubseteq_\varphi$ above and the definition of the sequent order relation, it is easy to prove that $\tau < \tau'$. □

Appendix B. Proof of Soundness

To prove the soundness, we have to prove that all successful leaves are semantically sound and that all rules of the tableau reserve soundness. We consider two cases here and the rest of the cases can be easily proved.

Theorem B.1 (Soundness). *For any sequent $H, b \vdash \sigma \in \varphi$ with a successful tableau, $\sigma \in [\![\varphi]\!]_e^{L,H}$*

Proof. We consider the two following cases of successful leaves and rules:

Case 1: The sequent $H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$, is a successful leaf when $\forall \theta$. **match** $(\sigma, r_1, \theta)\} = \mathtt{ff}$. This agrees with the semantics $[\![[r_1 \looparrowright r_2]\varphi]\!]_e^{L,H} = \{\sigma \in L | \forall \theta$. $\mathsf{match}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi]\!]_e^{L,H}\}$, since the implication ($\Rightarrow$) will evaluate to $\mathtt{tt}$. Moreover, the rule $R_{[]}$ reserves soundness since it is just an expression of the semantics of $\forall$ from first order logic.

Case 2: The sequent $H, \epsilon \vdash \sigma \in vZ.\varphi$ is a successful leaf when $\sigma : vZ.\varphi \in H$. We recall the definition of R_v and the relativized semantics of $vZ.\varphi$:

$$\frac{H, b \vdash \sigma \in vZ.\varphi}{H' \cup \{\sigma : vZ.\varphi\}, b \vdash \sigma \in \varphi[vZ.\varphi/Z]}, \quad \sigma : vZ.\varphi \notin H$$
$$[\![vZ.\varphi]\!]_e^{L,H} = \left(v[\![\varphi]\!]_{e[Z \mapsto S \cup S']}^{L,H}\right) \cup S'$$

where, $H' = H \backslash \{\sigma' : \Gamma | vZ.\ \varphi \prec \Gamma\}$ and $S' = H \Cap vZ.\varphi$. So, what we would like to prove is that $[\![\varphi[vZ.\varphi/Z]]\!]_e^{L,H' \cup \{\sigma : vZ.\varphi\}} = \left(v[\![\varphi]\!]_{e[Z \to S \cup \{\sigma\}]}^{L,H}\right) \cup \{\sigma\}$. The proof relies on Lemma 3.1 and on the properties of fixpoints. It is detailed in [5]. □

Appendix C. Proof of completeness

The proof depends on the following theorem:

Theorem C.1. *The sequent $\tau = H, b \vdash \sigma \in \varphi$ has a successful tableau if and only if $\tau' = H, b \vdash \sigma \in \neg\varphi$ has no successful tableau*

Proof.

Step 1: ⇒Suppose that both τ and τ' have successful tableaux, then by the soundness theorem $\sigma \in [\![b\varphi]\!]_e^{L,H}$ *and* $\sigma \in [\![b\neg\varphi]\!]_e^{L,H}$, which implies $\sigma \in [\![\neg b\varphi]\!]_e^{L,H}$. From the definition of the semantics, we will have $\sigma \in [\![b\varphi]\!]_e^{L,H}$ and $\sigma \notin [\![b\varphi]\!]_e^{L,H}$, which is a contradiction.

Step 2: ⇐We would like to prove that if τ has no successful tableau then τ' has a successful tableau. We do this by induction on the height of proof tree, starting from the leaves, i.e., prove that unsuccessful leaves imply $\sigma \in \neg\varphi$ and whenever a node in the proof tree implies $\sigma \in \neg\varphi$ its parent implies the same thing. We consider here the case for the unsuccessful leaf $H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\{\sigma \in L | \exists\theta.$ **match** $(\sigma, r_1, \theta)\} = \emptyset$ and the rule $R_{\Box}$. The rest of the cases match those proved in [4]. By definition, we have $H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi \Rightarrow H, \epsilon \vdash \sigma \in \neg[r_1 \looparrowright r_2]\varphi$ which is a successful leaf to prove that $\sigma \in \neg[r_1 \looparrowright r_2]\varphi$. Now for the rule $R_{\Box}$:

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

By the rule $R_{\Box}$, if any $\xi_i = H, b_i \vdash \sigma' \in \varphi$ does not have a successful tableau, then $\tau = H, b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ does not have a successful tableau. In this case, by induction hypothesis, $\xi_i' = H, b_i \vdash \sigma' \in \neg\varphi$ has a successful tableau. By definition, we have $H, \neg b_i \vdash \sigma' \in \varphi$ has a successful tableau. But $b = b_1 \times b_2 \cdots \times b_i \cdots \times b_n$, so the term $b_1 \times b_2 \cdots \times \neg b_i \cdots b_n$ will equal $\neg b$ (by definition of the $\times$ operation), which means that $H, \neg b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$, will have a successful tableau, or in other words $H, b \vdash \sigma \in \neg[r_1 \looparrowright r_2]\varphi$ will have a successful tableau. □

Theorem C.2 (*Completeness*). *If for a sequence* $\sigma \in L$, $\sigma \in [\![\varphi]\!]_e^{L,H}$, *then the sequent* $H, b \vdash \sigma \in \varphi$ *has a successful tableau.*

Proof. The proof follows from Theorems B.1 and C.1 by contradiction. □

References

[1] Snort – sourcefire inc., <http://www.snort.org>, accessed in April, 2007.

[2] Sophos inc., <http://www.sophos.com>, accessed in April, 2007.

[3] Tenable Network Security, <http://www.nessus.org/>, accessed in April, 2007.

[4] K. Adi, M. Debbabi, M. Mejri, A new logic for electronic commerce protocols, International Journal of Theoretical Computer Science TCS 291 (3) (2003) 223–283.

[5] R. Cleaveland, Tableau-based model checking in the propositional mu-calculus, Acta Informatica 27 (8) (1990) 725–748.

[6] F. Cuppens, Managing alerts in a multi-intrusion detection environment, in: Proceedings of the 17th Annual Computer Security Applications Conference, 2001.

[7] F. Cuppens, A. Miege, Alert correlation in a cooperative intrusion detection framework, in: Proceedings of the IEEE Symposium on Security and Privacy, 2002.

[8] H. Debar, A. Wespi, Aggregation and correlation of intrusion-detection alerts, in: Recent Advances in Intrusion Detection, LNCS, vol. 2212, 2001.

[9] P. Gladyshev, A. Patel, Finite state machine approach to digital event reconstruction, Digital Investigation Journal 1 (2) (2004).

[10] P. Gladyshev, A. Patel, Formalising event time bounding in digital investigations, International Journal of Digital Evidence 4 (2) (2005).

[11] C. Hosmer, Time Lining Computer Evidence, available at <http://www.wetstonetech.com/f/timelining.pdf>, 1998.

[12] K. Julisch, Clustering intrusion detection alarms to support root cause analysis, ACM Transactions on Information and System Security 6 (4) (2003).

[13] W. Kruse, J. Heiser, Computer Forensics: Incident Response Essentials, Addison-Wesley, Boston, MA, 2002. http://www.forensics-intl.com/book2.html .

[14] R. Leigland, A.W. Krings, A formalization of digital forensics, Digital Investigation Journal 3 (2) (2004).

[15] K. Monroe, D. Bailey, System Base-lining: A forensic perspective, available at <http://ftimes.sourceforge.net/Files/Papers/baselining.pdf>, 2003.

[16] B. Morin, H. Debar, Correlation of intrusion symptoms: an application of chronicles, in: Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection (RAID03), 2003.

[17] B. Morin, L. Me, H. Debar, M. Ducasse, M2d2: A formal data model for ids alert correlation, in: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), 2002.

[18] Y.C.P. Ning, D.S. Reeves, Constructing attack scenarios through correlation of intrusion alerts, in: Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, D.C., 2002.

[19] C. Peikari, A. Chuvakin, Security Warrior, O'Reilly, 2004.

[20] P. Porras, M. Fong, A. Valdes, A mission-impact-based approach to infosec alarm correlation, in: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), 2002.

[21] R.F. Smith, Windows Server 2003 Security Log Revealed, MTG Press, 2003.

[22] T. Stallard, K. Levitt, Automated Analysis for Digital Forensic Science: Semantic Integrity Checking, in: 19th Annual Computer Security Applications Conference, Las Vegas, NV, USA, 2003.

[23] S. Staniford, J. Hoagland, J. McAlerney, Practical automated detection of stealthy portscans, Journal of Computer Security 10 (1/2) (2002).

[24] P. Stephenson, Modeling of post-incident root cause analysis, International Journal of Digital Evidence 2 (2) (2003).

[25] S. Templeton, K. Levitt., A requires/provides model for computer attacks, in: Proceedings of New Security Paradigms Workshop, 2000.

[26] P.B.V. Yegneswaran, S. Jha., Global intrusion detection in the domino overlay system, in: Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS04), 2004.

[27] A. Valdes, K. Skinner, Probabilistic alert correlation, in: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001), 2001.

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