Formal Methods for Intrusion Detection of Windows NT Attacks

Sahika Genc
Sensor Informatics Technologies Laboratory, Computing and Decision Sciences Laboratory
General Electric Global Research Center
Niskayuna, NY 12309, USA
gencs@ge.com

***Abstract*—This study develops a method to build Finite State Automaton (FSA) that models the dependencies between the operating-system (OS)-level events recorded in the audit logs of a Windows NT machine. The FSA model contains both sequential and branching relations among audit log events that help the system administrator follow the steps of the intruder. Audit-log-FSA (ALFSA) are relatively proportional to the size of the audit log, hence, may have too many states and transitions. The superset-intrusion-signature-FSA (SISFSA) is extracted using formal methods on ALFSA. The SISFSA accepts a super set of the intruder's steps since not all of its actions may result in system failure, yet it provides the system administrator with a smaller set of patterns compared to ALFSA. Once the intrusion signature is finalized, fault diagnosis methods, developed for detection of faults in systems such as Heat Ventilation and Air-Conditioning (HVAC) systems, are used to detect the intrusion under full and partial-observation. Performance results are provided in the computer intrusion data set collected by Information Systems Technology Group at Massachusetts Institute of Technology Lincoln Laboratory under Defense Advanced Research Projects Agency and Air Force Research Laboratory (IST-MIT-LL data set) in 1999 for Windows NT attacks.**

***Index Terms*—Intrusion detection, discrete-event systems, pattern diagnosis.**

I. Introduction

Intrusion detection systems provide the system administrator with various tools, one of which is audit log that helps the administrator analyze the intruder's steps to develop detection and defense mechanisms against future intrusions. There are three types of Windows NT audit logs: 1) System, 2) Security, and 3) Application. Windows NT audit log files keep records of events defined in the audit policy. Each computer system with Windows NT has an audit policy defined in the Start Menu's Programs, Administrative Tools, User Manager, Policies, Audit, and the Audit Policy dialog box. Full-auditing functionality enables auditing of logon/logoff, file and object access, use of user rights, user and group management, security policy changes, restart, shutdown, and system, process tracking events for all user accounts, and base objects such as files and drivers. Information Systems Technology (IST) Group at Massachusetts Institute of Technology (MIT) Lincoln Laboratory under Defense Advanced Research Projects Agency and Air Force Research Laboratory designed a test bed computer network to collect data sets for evaluation and study of intrusion detection systems in 1998 and 1999. In 1999, three Windows NT machines were added to the test bed: victim, inside- and outside- attacker. Twelve Windows NT attacks including denial-of-service attacks, remote-to-local attacks, and user-to-root attacks were developed. A detailed description of the test bed and Windows NT attacks are shown in [1]. The author successfully used Windows NT audit logs of the victim machine to detect ten out of twelve attacks under full-auditing. He also showed that ten attacks leave a trace in the Windows NT Security Log file, but not in System or Application Log files, and two attacks left no trace in any of the NT audit log files, and developed intrusion signatures based on the traces left in the Windows NT Security Log files collected during the 1999 evaluation.

There have been studies in the literature on FSA based methods for intrusion detection. In [2], the authors monitor privileged processes on a UNIX machines such as TELNETD and LOGIND. They assume that the process is a black box and outputs *observable* (system call) that describes the dynamics characteristic of the program. The abnormal behavior is defined as deviations from the normal behavior and can be quantized through various measures. The authors were able to detect several classes of abnormal behavior such as failed intrusion attempts and error conditions in the UNIX program SENDMAIL. In [3], the authors developed methods that overcome the difficulties in [2], namely complexity in building FSA. The weakness of [3] was in tracing program calls for FORK and TRACE. In [4], the authors present an algorithm for automatically constructing a FSA that accepts the normal behavior and rejects all the other sequences. The algorithm minimizes the FSA by substituting macros for frequently occurring sequences and combining multiple sequences. The results of the experiments on several well-known data sets were promising: perfect detection rate, low False Positive Rate (FPR), and improved performance with respect to [2].

Our approach considers OS-level events logged by Windows NT auditing program, and combines the human intuition with machine automation. That is, one can easily go through the event set and built simple relations such as "*Process A creates process B*". The number of audit-log events is finite and there are few significant events used by the system administrator [5]. This is not much different that what a system administrator would do to identify the intruder's step after a system failure

or detection of abnormal behavior or otherwise during routine checks for system consistency. However, the administrator would have to analyze one or more audit-logs from start to end building these relations on the spot. Our approach automates this processes of parsing through thousands of audit-event logs. Building dependency (relational) graphs (e.g., FSA) to identify the sequences of OS-level events that led to an intrusion has been studied in the literature. In [6], authors describe a tool BackTracker that identifies event sequences leading to intrusion detection point. Intrusion detection point refers to the state on the local computer upon detection of a compromise such as modified system file or port-scan. The authors recommend various tools in the literature and market to detect a compromise. BackTracker works backwards from the intrusion detection point and forms chains of events that lead to the compromise. Then, an administrator may analyze these sequences of events to quickly identify vulnerabilities in the system. Our study follows a more comprehensive approach. First, there is no other software that detects a compromise. Second, our approach requires a collection of audit logs to identify the steps of the intruder, i.e., intrusion signature, and then, simple FSA operations are performed on the collection of audit logs to minimize the set of sequences the administrator works on to understand the vulnerabilities. Our study differs from [1] because in that study extraction of intrusion signatures requires extensive knowledge on the structure of the attack. In this study, signature extraction is automated through series of FSA operations on the collection of audit-log files collected during normal operation and intrusion. The audit-logs with intrusion is used to identify similar abnormal behavior that differs from normal behavior.

The paper is organized as follows. In Section II, a summary of terms, notation, and operations in FSA theory is presented. In the following sections the construction of ALFSA and SISFSA are described, respectively. Then, intrusion signature diagnosis is described. Throughout the paper, the techniques and methods are supported via examples built from the audit-logs and attacks in IST-MIT-LL data set. Finally, some concluding remarks are provided.

II. Preliminaries

Let Σ be a finite set of events. A *string* is a finite-length sequence of events in Σ. Given a string s, the length of s (number of events including repetitions) is denoted by $\parallel s \parallel$. The set of all strings formed by events in Σ is denoted by Σ^*. The set Σ^* is also called the Kleene-closure of Σ. Any subset of Σ^* is called a *language* over Σ. The *prefix-closure* of language L is denoted by $\overline{L}$ and defined as $\overline{L} = \{s \in \Sigma^* : \exists t \in L \text{ such that } st \in L\}$. Given a string $s \in L$, L/s is called the *post-language* of L after s and defined as $L/s = \{t \in \Sigma^* : \exists st \in L\}$. L is *live* if every string in L can be extended to another string in L. Let L be a language over $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$, where Σ_o and Σ_{uo} denote the observable and unobservable event sets, respectively. The projection of strings from L to Σ_o^* is denoted by P. Given a string $s \in L$, $P(s)$ is obtained by removing unobservable events (elements of Σ_{uo}) in s. The inverse projection of a string $s_o \in \Sigma_o^*$ with respect to L is denoted by $P^{-1}(s_o)$ and is equal to the set of strings in L whose projection is equal to s_o.

Given an event $\sigma \in \Sigma$ and a string $s \in \Sigma^*$, we use the set notation $\sigma \in s$ to say that σ appears at least once in s. Given a string of the form $u = stv$ in L where s and t are also strings in L, then s is called a *prefix*, t is called a *substring* of u, and v is called a *suffix* of u. Given a string $s \in L$, a *subsequence* of s is obtained by deleting zero or more events in the string s.

Given a language L and a finite set of bounded strings K over Σ, we define the set $\mathcal{S} \subseteq L$ as $\mathcal{S} = \{s \in L : (\exists u \in K)(u \text{ is a subsequence of } s)\}$ and the set $\mathcal{T} \subseteq L$ as $\mathcal{T} = \{s \in L : (\exists u \in K)(u \text{ is a substring of } s)\}$. We define the set $\Psi_{\mathcal{S}}(K) \subseteq \mathcal{S}$ as $\Psi_{\mathcal{S}}(K) = \{s\sigma \in \mathcal{S} : (\exists u\sigma \in K)(u\sigma \text{ is a subsequence of } s\sigma)\}$ and the set $\Psi_{\mathcal{T}}(K) \subseteq \mathcal{T}$ as $\Psi_{\mathcal{T}}(K) = \{s\sigma \in \mathcal{T} : (\exists u\sigma \in K)(u\sigma \text{ is a substring of } s\sigma)\}$.

A Finite State Automaton (FSA) is a five-tuple

$$G = (Q, \Sigma, \delta, q_0, F) \tag{1}$$

where Q is the set of states, Σ is the finite set of events, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function, q_0 is the initial state, and $F \subseteq Q$ is the set of marked states. We extend δ from domain $Q \times \Sigma$ to domain $Q \times \Sigma^*$ as follows: $\delta(q, \epsilon) = q$, $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$, for $s \in \Sigma^*$ and $\sigma \in \Sigma$. The language *generated* by G is $\mathcal{L}(G) = \{s \in \Sigma^* : \delta(q_0, s) \text{ is defined.}\}$. The language *marked* by G is $\mathcal{L}_m(G) = \{s \in \Sigma^* : \delta(q_0, s) \in F\}$. A set of states $\{q_1, \ldots, q_l\} \subseteq Q$ and a string $\sigma_1 \ldots \sigma_l \in \Sigma^*$ form a *cycle* in G if $q_{i+1} = \delta(q_i, \delta_i)$ for $i = 1, \ldots, l-1$ and $q_1 = \delta(q_l, \sigma_l)$.

Product of two FSA G_1 and G_2 is denoted by $G_1 \times G_2$ and defined as

$$G_1 \times G_2 = ((Q_1 \times Q_2), \Sigma_1 \cap \Sigma_2, \delta, (q_{0,1}, q_{0,2}), (F_1 \times F_2)) \tag{2}$$

where $(Q_1 \times Q_2)$ is the cartesian product of two sets and the state transition function is $\delta((q_1, q_2), \sigma) = \{(\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)), \sigma \in \Sigma_1 \cap \Sigma_2\}$. The language generated by the product of two FSA is $\mathcal{L}(G_1 \times G_2) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ and the language marked is $\mathcal{L}_m(G_1 \times G_2) = \mathcal{L}_m(G_1) \cap \mathcal{L}_m(G_2)$. Product operation synchronizes the event sequences common to both FSA.

Parallel composition of two FSA G_1 and G_2 is denoted by $G_1 || G_2$ and defined as in Equation 2 where the state transition function is

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)), & \sigma \in \Sigma_1 \cap \Sigma_2 \\ (\delta_1(q_1, \sigma), q_2), & \sigma \in \Sigma_1 \setminus \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)), & \sigma \in \Sigma_2 \setminus \Sigma_1 \end{cases} \tag{3}$$

The language generated by the parallel composition is $\mathcal{L}(G_1 || G_2) = P_1^{-1}(\mathcal{L}(G_1)) \cup P_2^{-1}(\mathcal{L}(G_2))$ and the language marked is $\mathcal{L}_m(G_1 || G_2) = P_1^{-1}(\mathcal{L}_m(G_1)) \cup P_2^{-1}(\mathcal{L}_m(G_2))$. Parallel composition of the two FSA synchronizes on the common sequences of events while keeping the rest of the sequences from both FSA. Both product and parallel composition operations can be extended to more than two FSA.

TABLE I
WINDOWS NT AUDIT-LOG EVENT IDS IMPLEMENTED IN THIS STUDY AND THEIR DESCRIPTIONS.

Windows NT Audit-log event ID	Description
528	Logon/Logoff: Successful logon
538	Logon/Logoff: User logoff
576	Privilege use: Special privileges assigned to new logon
577	Privilege use: Privileged Service Called
592	Detailed Tracking: Create Process
593	Detailed Tracking: A process has exited
636	Account Management: Security Enabled Local Group Member Added
639	Account Management: Security Enabled Local Group Changed
642	Account Management: User Account Changed
632	Account Management: Security Enabled Global Group Member Added

Complement of FSA G with respect to an event set Σ is another FSA G^{comp} such the language generated by the complement FSA accepts $\Sigma^* \setminus \mathcal{L}(G)$.

III. AUDIT-LOG FINITE STATE AUTOMATA

Windows NT audit-log files record events with a unique event identification number (ID). Each audit-log event is stored with various fields. A complete list of events, their descriptions, and fields can be found in [7]. The list of audit-log events modeled in this study are provided in Table I. We define (natural) relations based on the fields of audit-event. Fields are specific to each audit-log event type. For example, event *592* logs creation of a new process and stores the new process ID and creator's process ID in two separate fields. Thus, a relation based on these fields can be "*Creator process with ID 2154371904 creates a new process with ID 2153437184*". Each relation is mapped to an FSA in consistent with its natural language description. For example, the relation "*Creator process with ID 2154371904 creates a new process with ID 2153437184*" is mapped to FSA with the set of states $Q = \{$ *Creator process ID*, *New process ID* $\}$, event set $\Sigma = \{$ *create* $\}$, and the state transition function *Creator process ID* $= \delta$*(New process ID, create)*. In this study, we focus on the audit-log event *592*. The ALFSA algorithm composes the FSA built for each audit-log event together through their states like blocks quilted together through their edges. Thus, FSA for each audit-log event type or each audit-log event can be built independently and composed into ALFSA as required. In this section, we describe the ALFSA algorithm formally and provide examples from IST-MIT-LL data set.

Windows NT audit log files of the 1999 evaluation are distributed freely on [8]. We convert the audit log files to ASCII format with *Event Log Explorer* [9]. An example log of a *592* event is shown in Table II. The set of relations for this event is $R = \{$*Creator logs into Domain with User Name and Logon ID, Creator runs Creator process ID, Creator process ID creates new process ID with Image File Name* $\}$. The FSA built is shown in Fig. 1 on the left. The initial state is 0 and there is a hierarchical relation between *Domain, User Name,* and *Logon ID*. Now suppose that there is another *592* logged after the *592* event shown in Table II and have the same *Domain, User Name,* and *Logon ID* but the creator and new process IDs are different. When these two events are composed, the resulting FSA is as shown in Fig. 1 on the right. That is, the dashed (green line) states and transitions are added to the FSA of the *592* event shown on the left. That is, the same user runs another process that creates a new process. The FSA for an audit-log event is sequential. The branching processes are formed through the composition of the FSA.

TABLE II
592: CREATE PROCESS AUDIT-LOG EVENT EXTRACTED FROM ONE OF THE AUDIT-LOG FILES IN IST-MIT-LL DATA SET.

Date	3/29/1999
Time	9:43:08 AM
ID	592
Description	Security Detailed Tracking
Host	HUME
Details	A new process has been created:
	New Process ID: 2153437184
	Image File Name: INSTALL.EXE
	Creator Process ID: 2154371904
	User Name: Administrator
	Domain: EYRIE
	Logon ID: (0x0,0x3AFF)

We now define the composition operation. Suppose that there are N *592* events. Let $G_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, F_i)$ denote the FSA for a single *592* event. ALFSA for *592* events is

$$ALFSA = (Q, \Sigma, \delta, q_0, F) \quad (4)$$

where $Q = \cup_{i=1}^{N} Q_i$, $\Sigma = \cup_{i=1}^{N} \Sigma_i$, $\delta = \cup_{i=1}^{N} \delta_i$, $q_0 = 0$, and $F = \cup_{i=1}^{N} F_i$. That is, ALFSA (except the initial state) is simply the union of all the FSA built for the *592* events. The number of states, events, and transitions of the *592* ALFSA built for the audit-logs in IST-MIT-LL data setis listed in Table III. Audit-logs were collected everyday for five weeks. The first three weeks of the evaluation contain normal data, i.e., no attacks. Attacks were performed during the last two weeks of the evaluation. Two files in the data set did not convert to ASCII files, thus, are excluded. The audit-logs that are not listed in Table III and convert to the ASCII format correctly did not provide additional information and are excluded in our study.

IV. SUPERSET INTRUSION SIGNATURE FINITE STATE AUTOMATA

The construction of SISFSA requires two sets of audit-logs: 1) logs during normal operation and 2) logs collected during intrusion. We assume that the audit-log files are not altered by the intruder. Our goal is to identify the event sequences

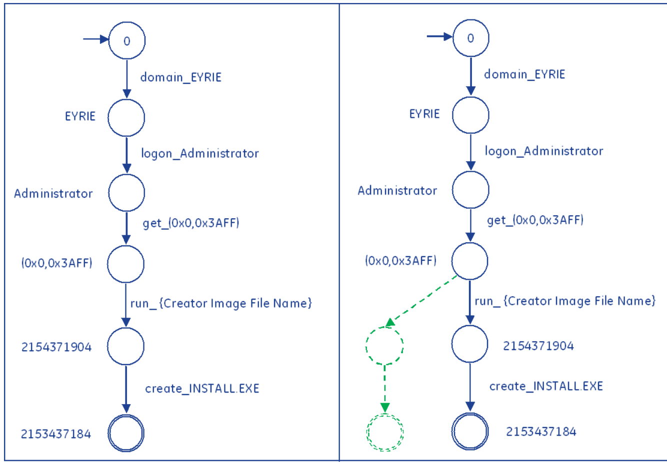

Fig. 1. FSA for *592* and composition of the two FSA for *592* events.

TABLE III
NUMBER OF STATES, EVENTS, AND STATE TRANSITIONS OF *592: Create process-ALFSA* BUILT FOR IST-MIT-LL DATA SET.

Week	Day	Number of states	Number of events	Number of transitions
1	Monday	925	183	1354
1	Wednesday	208	90	302
1	Thursday	178	79	233
2	Thursday	123	77	161
2	Friday	99	69	120
3	Wednesday	52	54	63
3	Friday	74	62	92
4	Monday	123	84	158
4	Tuesday	186	102	248
5	Monday	238	123	325
5	Tuesday	150	87	199
5	Thursday	382	94	437

that appear in the logs collected during the intrusion but are not contained in the logs during normal operation. Thus, we define SISFSA as follows

$$SISFSA = (G_1^A \times \ldots \times G_n^A) \times (G_1^N || \ldots || G_m^N)^{comp} \quad (5)$$

where the superscript A and N stands for ALFSA built from logs collected during intrusion and normal operation, respectively. The product operation synchronizes the sequences common during intrusion. Parallel composition forms a complete picture of the normal behavior by synchronizing on the common sequences and accounting for the individual sequences in each normal ALFSA. The complement operation with respect to the event set of the product of the logs during intrusion builds sequences that are not in the normal behavior. The product of the complement of the normal behavior with the synchronized log collected during intrusion creates the SISFSA.

We now consider two attacks implemented in the 1999 evaluation CrashIIS and Yaga. In [1], the CrashIIS attack is described as a denial-of-service attack against the Windows NT IIS web server. The attacker sends a malformed GET request via telnet to port 80 that crashes the web server, FTP and Gopher daemons. The victim machine was attacked with CrashIIS on Week-4-Tuesday, Week-5-Monday, and Week-5-Tuesday. The normal audit-log files used in the example are listed in Table 3, i.e., the audit-log collected during the first 3 weeks of the evaluation. We automate the calculation of SISFSA via a MATLAB batch file that calls the UMDES Software Library [10] to perform FSA. The SISFSA for CrashIIS in the IST-MIT-LL dataset is shown. When IIS service is turned on, a process called INETINFO.EXE is created and when it crashes DRWTSN32.EXE (debugger for application errors) is started. The CrashIIS SISFSA shown in Figure 3. The CrashIIS SISFSA contains the event sequence INETINFO.EXE followed by DRWTSN32.EXE which indicates that the attacked log files contain the CrashIIS attack.

In [1], the Yaga attack is described as follows: Yaga edits the Registry by replacing lines corresponding to the call for DRWTSN32.EXE when a service crashes. The intruder attacks the victim machine with CrashIIS that causes a service crash and the replaced line in the Registry is executed that gives the intruder administrative privileges in the Domain. The audit-log event *592: Create process* ALFSA identifies the part of the Yaga attack that involves processes. We can get a more complete picture by including the Account Management audit-log events (see Table 1) in ALFSA. The victim machine was attacked with Yaga on Week-4-Monday, Week-5-Thursday. The Yaga SISFSA is shown in Figure 4. When the service crashes (via CrashIIS attack) INETINFO.EXE is run and then the replaced line in the Registry executes NET.EXE to add the intruder to the Domain. In Figure 4, there are two sequences where INETINFO.EXE creates NET.EXE: 1) States 8-12-9 and 2) States 10-19-9. Also, the very same user who probably replaced the DRWTSN.EXE with NET.EXE runs REGEDIT.EXE. The CrashIIS attack used to crash the IIS service is not apparent in this FSA because of a bug in MAILSRV.EXE that was discovered later during the evaluation. The bug in the MAILSRV.EXE results in calling DRWTSN32.EXE. However, it is seen in Figure 4 that the processes (states 8 and 10) that run part of the Yaga attack, i.e., INETINFO.EXE followed by NET.EXE, also runs MAILSRV.EXE followed by DRWTSN.EXE suggesting the link with the service crash.

V. DIAGNOSIS OF INTRUSION SIGNATURE

In the previous section, we described a method to extract a super set of intruder's steps from two sets of audit-logs where one set is collected during normal operation and the other during intrusion. We showed that event sequences in SISFSA might provide enough evidence to identify exactly the intruder's steps. In this section, we present a method for pattern diagnosis that has been successfully used to detect and isolate patterns in partially-observed discrete-event systems (DES) such as HVAC that result in system failure. In HVAC, observable events can be events that are directly recorded by sensors attached to the system. The reason we

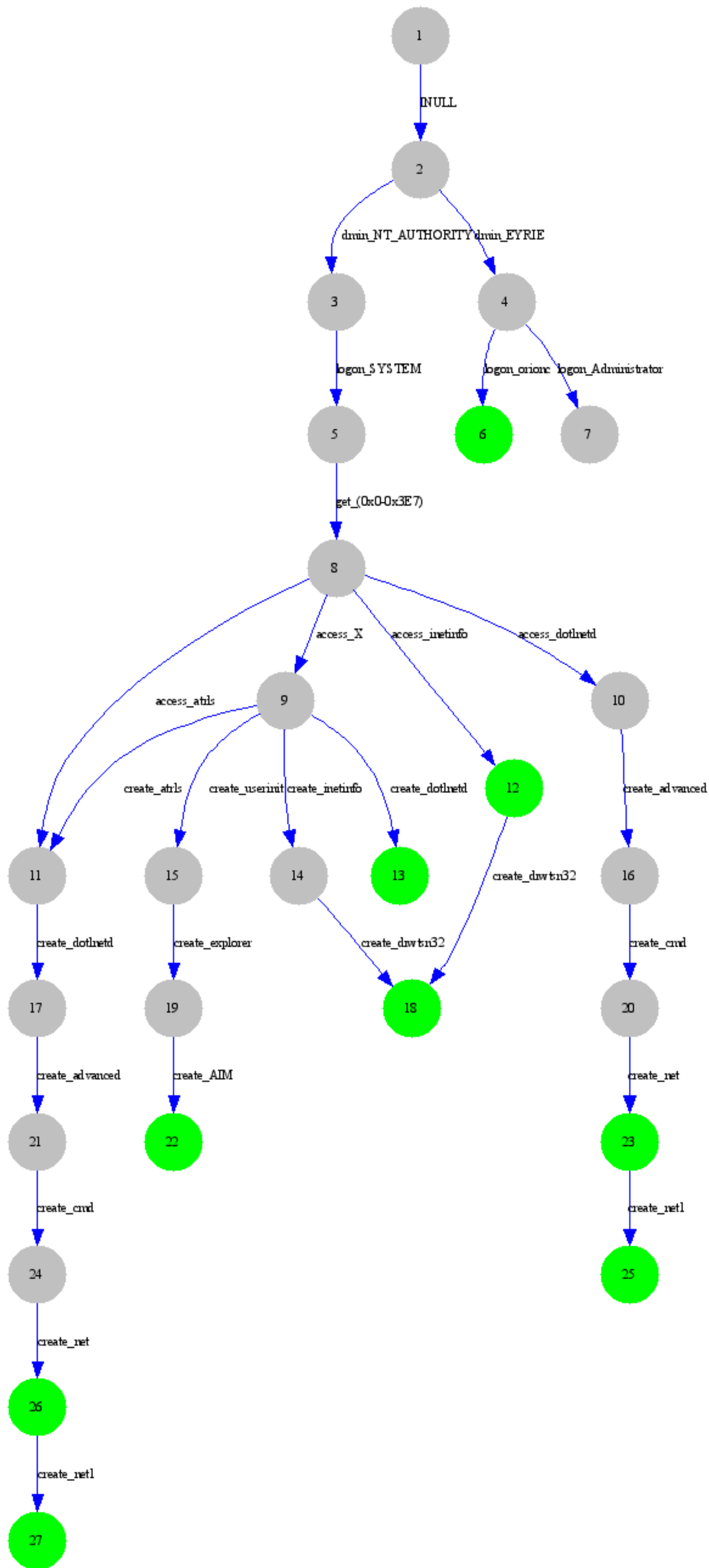

Fig. 2. The ALFSA for the audit-log event 592: Create process built from the audit-event log Week-4-Tuesday.

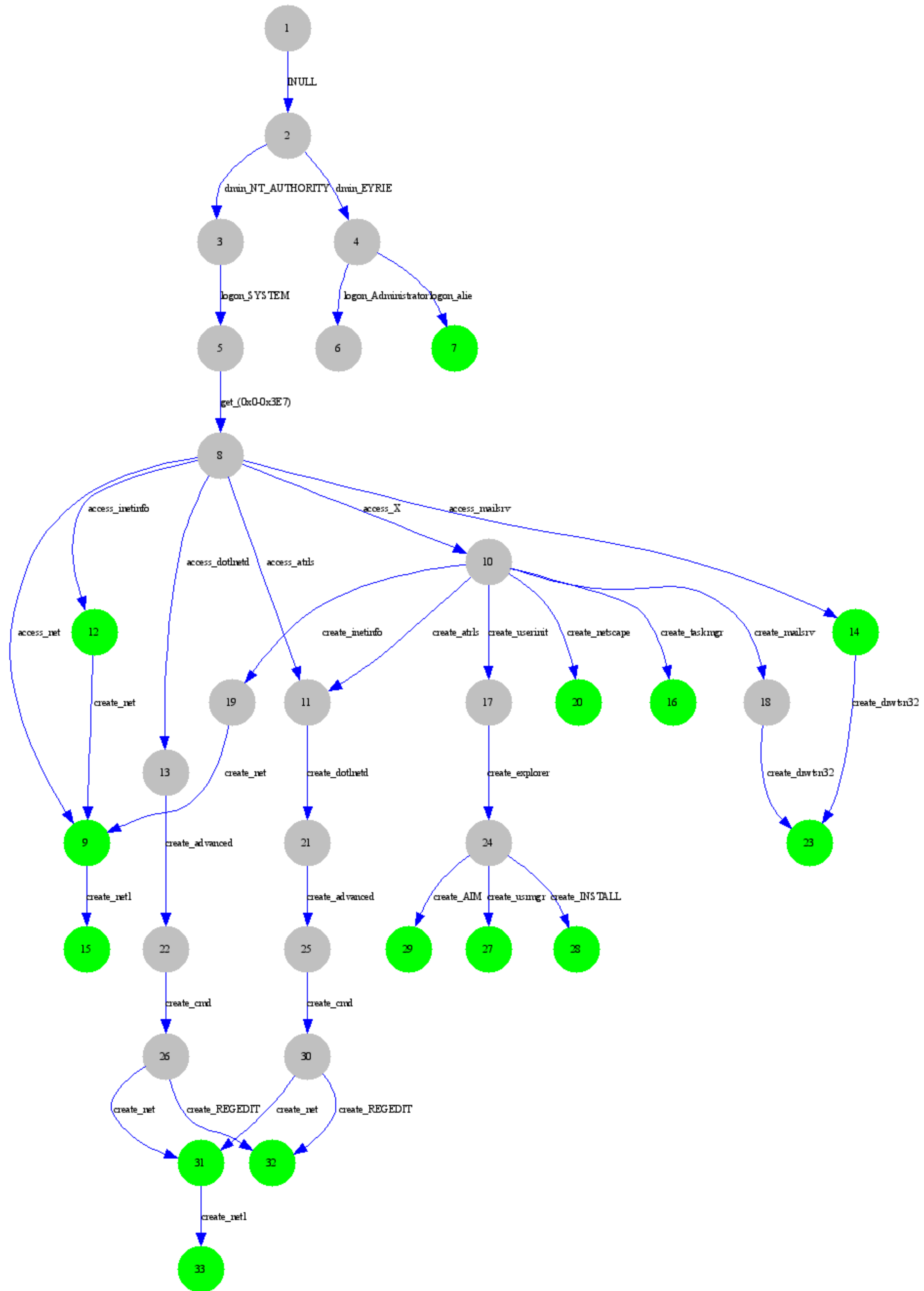

Fig. 3. The ALFSA for the audit-log event 592: Create process built from the audit-event log Week-4-Tuesday.

consider a method for partially-observed systems is as follows: A complete ALFSA built for *all* audit-log events may be very large and the administrator may choose to filter one or more audit-log events to focus his detective work on a selected set of events. Partial-observation methods help us understand the effect of filtering on pattern diagnosis. In this study, observable events are audit-log events that are recorded according to the audit policy or otherwise significant events such as *login/logout* and *create process*.

The problem of fault diagnosis for discrete-event systems has received considerable attention in the last decade and diagnosis methodologies based on the use of discrete-event models have been successfully used in a variety of technological systems ranging from document processing systems to intelligent transportation systems; see [11] and the references therein. The methodology of the Diagnoser Approach introduced in [12] is generalized (from diagnosis of a single event) to event patterns in [13], [14]. The contribution of [13], [14] is two-fold: 1. Off-line verification of the diagnosability property of the system with respect to the pattern, i.e., *if* the system is diagnosable with respect to the pattern. 2. On-line monitoring of the system and diagnosis of the pattern, i.e., *how* to detect the occurrence of the pattern while *partially* observing the behavior of the system.

In [13], two different notions of pattern diagnosability are defined in the context of formal languages: (i) S-type pattern diagnosability and (ii) T-type pattern diagnosability. These two different types stem from different approaches to defining the occurrence of a pattern. In S-type pattern diagnosability, a pattern is detected if all the sequences executed by the system that record the same observed event sequences contain *subsequences* in the pattern. In T-type pattern diagnosability, a pattern is detected if all the sequences executed by the system that record the same observed event sequences contain *substrings* in the pattern. In other words, there could be events interleaved between the events that make up the pattern in the S-type case, but not in the T-type case.

In order to diagnose a T-type (S-type) pattern s in a DES modeled by FSA with event set Σ, we first built FSA $H_{\mathcal{T}}(\Sigma, s)$ such that the projection of the language marked by the T-type (S-type) pattern FSA to the set of observable events contains the intrusion signature as a *substring* (*subsequence*). The necessary and sufficient condition for pattern diagnosability is based on another DES structure called *Observer*. The observer of FSA $G = (Q, \Sigma, \delta, q_0, F)$ is (see, e.g., [15] for further details)

$$\mathbf{Obs}(G) = (X, \Sigma_o, \delta_o, x_o), \tag{6}$$

where $x \in X$ is a set of states in Q,$\Sigma_o \subseteq \Sigma$ is the set of observable events, and x_0 is the initial observer state.

An observer state $x = \{q_1^x, \dots, q_l^x\}$ is *marking-certain* if $q_i^x \in F$ for $i = 1, \dots l$, and *marking-uncertain* if there exists $q_i^x \in F$ and $q_j^x \in Q \setminus F$ for some $i, j \in \{1, \dots, l\}$. Let $\{x_1, \dots, x_l\}$ and $\sigma_{o,1} \dots \sigma_{o,l} \in \Sigma_o^*$ form a cycle in $\mathbf{Obs}(G)$. The cycle in $\mathbf{Obs}(G)$ is a *marking-indeterminate cycle* if the following are satisfied

1) x_i is marking-uncertain for $i = 1, \dots, l$,
2) $\exists q_i^k,\ r_i^l \in x_i$ for all $i = 1, \dots, m$, $k = 1, \dots, M$, and $l = 1, \dots, N$ such that
   a) q_i^k is marked and r_i^l is not marked for all i, k, l,
   b) there are two corresponding cycles in G[1]:

$$\begin{array}{l} q_1^1 \stackrel{\sigma_{o,1} t_1^1}{\longrightarrow} q_2^1 \ \dots \stackrel{\sigma_{o,m-1} t_{m-1}^1}{\longrightarrow} q_m^1 \stackrel{\sigma_{o,m} t_m^1}{\longrightarrow} \\ q_2^2 \ \dots \stackrel{\sigma_{o,m-1} t_{m-1}^2}{\longrightarrow} q_m^2 \dots \ q_1^M \stackrel{\sigma_{o,1} t_1^M}{\longrightarrow} q_2^M \\ \dots \stackrel{\sigma_{o,m-1} t_{m-1}^M}{\longrightarrow} q_m^M \stackrel{\sigma_{o,m} t_m^M}{\longrightarrow} q_1^1 \end{array} \tag{7}$$

and

$$\begin{array}{l} r_1^1 \stackrel{\sigma_{o,1} u_1^1}{\longrightarrow} r_2^1 \ \dots \stackrel{\sigma_{o,m-1} u_{m-1}^1}{\longrightarrow} r_m^1 \stackrel{\sigma_{o,m} u_m^1}{\longrightarrow} \\ r_2^2 \ \dots \stackrel{\sigma_{o,m-1} u_{m-1}^2}{\longrightarrow} r_m^2 \ \dots \ r_1^N \stackrel{\sigma_{o,1} u_1^N}{\longrightarrow} r_2^N \\ \dots \stackrel{\sigma_{o,m-1} u_{m-1}^N}{\longrightarrow} r_m^N \stackrel{\sigma_{o,m} u_m^N}{\longrightarrow} r_1^1 \end{array} \tag{8}$$

where $t_i^k,\ u_i^l \in \Sigma_{uo}^*$ for all i, k, l. □

A union FSA $\mathbf{U}(G_1, G_2)$ of G_1 and G_2 is such that $\mathcal{L}(\mathbf{U}(G_1, G_2)) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ and $s \in \mathcal{L}_m(\mathbf{U}(G_1, G_2))$ if $s \in \mathcal{L}_m(G_1)$ or $s \in \mathcal{L}_m(G_2)$. The extension of the union of two FSA to more than two is a recursive operation.

The necessary and sufficient condition for T-type pattern diagnosability of a regular language with respect to a pattern is as follows:

Theorem 1 (T-type): A prefix-closed, live language $L = \mathcal{L}(G)$ is T-type pattern diagnosable with respect to a pattern K and projection P iff $\mathbf{U}_{s \in K}(G \times H_{\mathcal{T}}(\Sigma, s))$ does not contain any marking-indeterminate cycles.
The condition for S-type is the same except for the construction of the pattern FSA. Pattern diagnosis as defined in this section can be performed in polynomial time with respect to the number of states of ALFSA.

In the following, we present two examples where one attack is a T-type pattern and the other is S-type. The intrusion signature for the CrashIIS attack (studied in the previous section) is a T-type pattern because DRWTSN.EXE is immediately followed by *INETINFO.EXE*. The T-type pattern FSA $H_{\mathcal{T}}$ for CrashIIS is shown in Fig. 4. Suppose that all the audit-log events in the ALFSA are observable, then *CrashIIS* is correctly detected on Week-4-Tuesday, Week-5-Monday, and Week- 5-Tuesday, but not on Week-4-Monday and Week 5-Thursday when launched as part of *Yaga* attack and during the normal operation. If INETINFO.EXE is filtered out and considered as an unobservable event, then *CrashIIS* is wrongly detected on Week-1-Wednesday, Week-2-Thursday, and Week-2-Friday ALFSA that are collected during normal operation. This means that filtering out events in the intrusion signature may result in False Positive (FP) detection of the alarm.

We now consider an S-type intrusion signature. The Casesen attack separated into three separate event sequences and all three need to be diagnosed in the ALFSA. The *Casesen* attack does not require consecutive event occurrences, thus it is an

[1] $q \stackrel{s}{\longrightarrow} q'$ denotes $q' = \delta(q, s)$ where q and q' are states and s is a string.

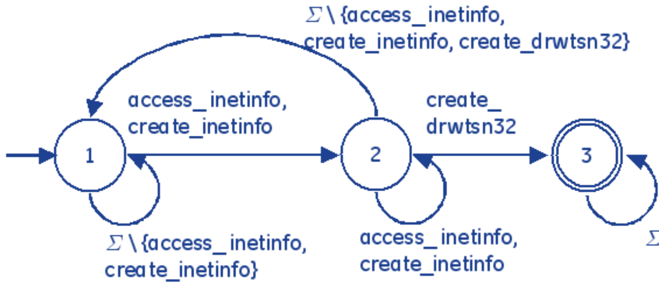

Fig. 4. CrashIIS T-type pattern FSA.

S-Type pattern. The S-type pattern FSA H_S for the Casesen attack is shown in Fig. 5.

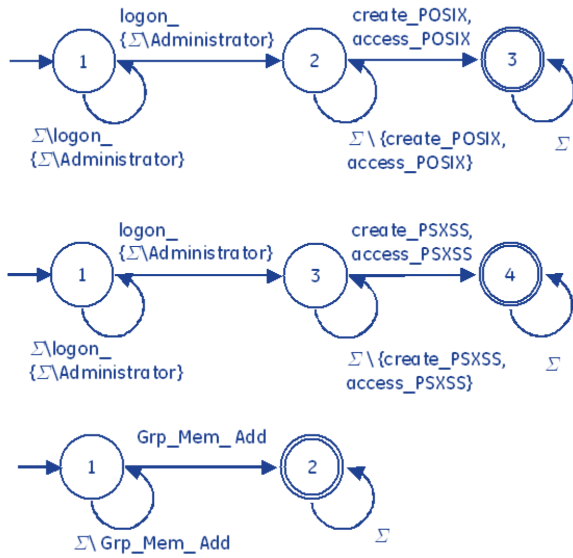

Fig. 5. FSA model of *Casesen* attack.

Suppose that all events are observable, then *Casesen* attack is correctly detected on Week-5-Tuesday and Week-5-Thursday, and there are no FPs for audit logs during normal operation. If all transitions created by *Logon/Logoff* audit-log events are unobservable, then the *Casesen* attack is still detected. There are no indeterminate cycles in Week-5-Tuesday ALFSA, so a sub-attack of *Casesen* is correctly detected. If we consider the very same unobservable events for Week-1-Wednesday, the occurrence of the same sub-attack is ambiguous, i.e., there is an indeterminate cycle. In this case, operator (or administrator) may either consider increasing the set of observable events gradually, act conservatively and declare intrusion, or otherwise declare no attack. The advantage is that if the pattern is diagnosable with a smaller set of observable events, then the operator can declare attack with certainity, and investigate more only if there is an indeterminate cycles. Thus, filtering out audit-log events may result in True Positive (TP) detection of the intrusion, however, may increase the number of FP detections.

VI. Conclusion

In this paper, we described an algorithm to build an FSA from audit-log files that reveals dependency relations between various OS-level events. The algorithm builds ALFSA by stitching smaller FSA built for each audit-log event to each other as it parses through the audit-log. Thus, our approach naturally combines the multiple sequences, unlike [4]. The length of the sequences are limited by the number of dependencies among the processes. This suggests that the length of the sequences are rather short compared to previous methods for construction of FSA.

The SISFSA built for two of the attacks studied in this paper revealed useful information in understanding and tracking the intruder's steps. Finally, we discussed an application of a pattern diagnosis method developed for detecting and isolating patterns that result in system failures in partially-observed FSA. The previous studies have not considered including the audit-logs collected during the same intrusion. The audit-logs collected during the same intrusion from multiple machines can be used in the method. Since the auditing program is universal (with the exception of level of auditing) over the Windows NT machines, there would be few consistency issues.

Partial-observation (and methods that allow the study of partially-observed systems) can be used to reduce the audit-log event set to fewer significant events, after which the detective work is performed on the smaller set of events, at the expense of increasing FPR. A comprehensive study of all attacks in the 1999 evaluation would be beneficial in providing more evidence on the strength of the approach. Also, extending of the study to contemporary attacks and technology would prove very useful in understanding the strengths and limitations of the techniques and methods described here.

References

[1] J. Korba, "Windows nt attacks for the evaluation of intrusion detection systems. Master of Engineering. Massachusetts Institute of Technology, Electrical Engineering and Computer Science," Jun. 2000.

[2] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998. [Online]. Available: citeseer.ist.psu.edu/article/hofmeyr98intrusion.html

[3] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," 2001, pp. 144–155. [Online]. Available: citeseer.ist.psu.edu/sekar01fast.html

[4] K. Wee and B. Moon, "Automatic generation of finite state automata for detecting intrusions using system call sequences," in *Lecture Notes in Computer Science: Computer Network Security*, vol. 2776/2003. Heidelberg, Germany: Springer Berlin, 2004.

[5] R. F. Smith, "Interpreting the NT security log," *Windows & .NET Magazine*, April 2000.

[6] S. T. King and P. M. Chen, "Backtracking intrusions." *ACM Trans. Comput. Syst.*, vol. 23, no. 1, pp. 51–76, Feb. 2005.

[7] Microsoft. Windows 2000 security event descriptions. [Online]. Available: http://support.microsoft.com/?kbid=299475

[8] IST-MIT-LL. Data sets. [Online]. Available: http://www.ll.mit.edu/IST/ideval/data/data_index.html

[9] FSProLabs. Event log expoler. [Online]. Available: http://www.eventlogxp.com/

[10] UMDES. Umdes software library. [Online]. Available: http://www.eecs.umich.edu/umdes/toolboxes.html

[11] S. Lafortune, D. Teneketzis, M. Sampath, R. Sengupta, and K. Sinnamohideen, "Failure diagnosis of dynamic systems: An approach based on discrete event systems," in *Proc. 2001 American Control Conf.*, Jun. 2001, pp. 2058–2071.

[12] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Failure diagnosis using discrete event models," *IEEE Trans. Control Systems Technology*, vol. 4, no. 2, pp. 105–124, Mar. 1996.

[13] S. Genc, "On diagnosis and predictability of partially-observed discrete-event systems. Ph.D. Thesis. University of Michigan, Electrical Engineering: Systems," Apr. 2006.

[14] T. Jeron, H. Marchand, S. Pinchinat, and M. Cordier, "Supervision patterns in discrete event systems diagnosis," in *Proceedings of the 8th International Workshop on Discrete-Event Systems*, 2006.

[15] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.