SUPERVISORY CONTROL OF MALICIOUS EXECUTABLES[†]

V.V. Phoha\$, Xin Xu
Louisiana Tech University
Ruston, LA 71272
\$ Email: phoha@acm.org

A. Ray‡, S. Phoha
The Pennsylvania State University
University Park, PA 16802
‡ Email: axr2@psu.edu

Abstract: This paper presents a systems-theoretic approach to profile, model, and control malicious executables in computer software. By treating the structural profile of malicious codes as a generator of formal languages, the language recognizer serves as a supervisory controller in the sense that the spread of malicious executables is arrested with the goal of making the virus ineffective. The theoretical foundation and the approach presented in this paper are applicable to a wide class of malicious executables. The controller can be designed as a separate program or as a background process to run on individual machines to monitor other processes. Simulation experiments on supervisory control of a *file virus* are presented as examples. *Copyright© 2003 IFAC*

Keywords: Discrete event system, Supervisory control, Automata theory, Detection systems, Software safety.

1. INTRODUCTION

Wide-spread connectivity provided by the Internet and an increase in use of computers have lead to development and rapid spread of malicious programs, such as viruses, Trojan horses, and logic bombs. This paper introduces a novel approach to control the spread of viruses and presents a technique to make them ineffective.

There are various approaches to detection and prevention of virus spread. Currently, the signature-based method is the most commonly used anti-virus solution in industry (Sandeep and Eugene, 1992; Matthew, et al., 2001). The signature-based method relies on prior knowledge about the structural or behavioral profile of a particular virus. This knowledge can be a signature consisting of a unique sequence of bytes at a known offset, or a special system call, or a set of system calls made when the virus is executing (Eugene, 1992). The shortcoming of this method is that users need to update the tools when a new virus comes out. Since about eight to ten malicious programs are created every day, on the average, and most of them cannot be detected until signatures have been analyzed, the computer system becomes vulnerable to attacks during this period (Matthew, et al., 2001). Many researchers (Baudouin, et al., 1995; Matthew, et al., 2001) are trying to devise *generic methods* to detect new viruses. This class of methods looks for virus-like behavior rather than specific viruses and warns the user once a virus-like behavior is detected.

The approach, introduced and adopted in the present paper, models the virus-like behavior to develop controllers that make viruses ineffective while the software is in execution, and thus largely complements the existing methods for virus protection of software, which are essentially off-line. We have applied discrete-event Supervisory Control Theory (SCT) to detect and prevent the spread of computer viruses where prior knowledge of the specific virus signatures is not required. SCT is a well-studied paradigm and has been used in many applications. Examples are: software systems (Karsai, et al., 2001), workflow management paradigm to schedule concurrent tasks through scheduling controllers (Wallace, et al., 1996) as well as for protocol converters to ensure consistent

[†]This work has been supported in part by the Army Research Office under Grant No. DAAD19-01-1-0646.

communications in heterogeneous network environments (Kumar and Fabian, 1997).

From the perspectives of software operation and control, we briefly review the relevant literature on system call and state transition analysis. The idea of tracing system calls has been extensively used for intrusion detection (ID) by many researchers (Christina, et al., 1999; Steven et al., 1998; Eleazar & Wenke, 2001; Wenke & Salvatore 1998). Their work shows that the use of system calls is critical to appropriate usage of system resources. Those methods monitor the system calls used by active, privileged processes, such as *sendmail* and *lpr*, and then, distinguish the abnormal executions from the normal ones.

Several researchers (Nuansri et al., 1999; Koral, et al., 1995; Michael and Anup, 2000; Steven et al., 2000) have developed ID techniques based on the principle of state transition analysis to detect new intrusions, but state transition analysis has been seldom used for virus detection. To the knowledge of authors, only Le Charlier et al. (1995) have used state transition diagrams to represent infection patterns for virus detection in the framework of rule-based expert systems. They then, use the system to detect virus behavior. The rules are generated by collecting and analyzing the computer audit data. In contrast, the virus ID system, proposed in this paper, is built upon the principle of SCT in the framework of state transition analysis to detect new malicious executables. We use an MS-DOS trace tool (Diomidis, 1994) to trace the normal and malicious executables, and build state transition model based on the trace results.

Preliminary results of simulation experiments show that the ID system achieves a 75% virus detection frequency. Higher values of frequency are anticipated with further improvement of the event definition. Novelties and significant contributions of the paper are summarized below:

- A novel approach to detection and prevention of the spread of new malicious executables.
- Real-time detection of new malicious executables.
- Modeling of system calls as events in the formal language setting.
- Reliable and consistent protection of software system and application processes.

This paper is organized in four sections including the present section. Section 2 presents the underlying theory of the proposed approach and formulates the model based on this theory. Section 3 presents and analyzes the simulation. Section 4 summarizes the paper and discusses future work in detection of malicious executables.

2. BASIC THEORY AND MODEL FORMULATION

This section introduces the basic theory of discrete-event supervisory control and describes how to formulate a model of malicious executables based on this theory. Before proceeding to the core issue of modeling and control, we briefly present a taxonomy of computer viruses. There are three broad categories of viruses (Alan and Tim, 1994; Richard, 1990) as listed below:

- *File viruses* that infect *.exe and *.com files.
- *Macro viruses* that infect data files.
- *Boot viruses* that infect boot sectors of hard disk or floppy disk.

The paper focuses on *file virus* but the underlying concepts presented here are applicable to any class of viruses.

2.1 Analysis of malicious executables

Generally, malicious executables contain a *replicator* that controls the spread of the virus, a *concealer* that keeps the virus from being detected, and a *trigger* that activate the virus (Alan and Tim, 1994; Richard, 1990). The malicious functions of the virus may vary, and their activation conditions may be different, but the system calls made in the replication stage and the infection stage are very similar. As an extension of general virus behavior observed in virus (Alan and Tim, 1994; Richard, 1990), we derive a general outline of replication and infection of a file virus as follows.

```
Procedure: Replication
while files available
{
  find a file;
  if file not infected
    {Infection (filename);
     infected file number ++;}
  else do nothing;
}
```

```
Procedure: Infection (file)
{
  Get the file attribute;
  Change the file attribute;
  Save the file data/time stamps;
  Copy virus code to the file;
  Restore the file attribute, date/time stamps;
}
```

This paper has used an MS-DOS trace tool (Diomidis, 1994) to trace 25 non-infected programs, and 30 malicious programs created by two virus generation tools:

- BW (http://vx.netlux.org/dat/tb00.shtml)
- G2 (http://vx.netlux.org/dat/tg00.shtml).

The trace results verify the general outline given above in procedures *Replication* and *Infection.* We define a sensitive-sequence as a sequence of system calls common to most virus programs. An analysis of 30 file virus programs and 25 non-infected programs shows the following sensitive-sequence for file virus: *findfirst*, *getmod*, *chmod*, *open*, *get_time*, *read*, [*write*, *lseek*], *set_time*, *chmod*, *close.* Note that occurrence of *write* depends on whether the file is infected or not; it may not appear if the file is already infected. The order of the system calls may be slightly different for each individual virus. For example, *open* may happen anywhere between *findfirst* and *write.* Since file viruses spread when the virus code executes, all system calls of the program are monitored in real time. If the system calls generated by the program follows the sensitive-sequence, it can be predicted with high confidence that the program is either a virus or is infected by a virus. An outline of the proposed approach follows.

We first model the process interactions as a Deterministic Finite State Automaton (DFSA) and enumerate its states and events that cause state transitions. We map system calls into events. In order to detect the presence of a file virus and prevent its spread. The process interactions are modeled as the *plant* DFSA G and the control specifications (that constrain the plant) are represented by another DFSA S that has the same event alphabet Σ as G. The parallel combination of S and G gives rise to a DFSA S/G, which is the closed loop system model under the supervisory control. This paper formulates the plant DFSA G and a control DFSA S based on the information generated from a set of 30 file viruses.

2.2 Model Formulation

A discrete-event system is driven by instantaneous occurrences of events. Using the Ramadge and Wonham framework (P.J.Ramadge, W.M.Wonhanm, 1987), a discrete event system to be controlled called a plant, is modeled by a trim deterministic finite state automaton (DFSA) $G = (Q, \Sigma, \delta, q_0, Q_m)$. Since the sequence of system calls made by a running process can be represented as discrete events, a computer executable process is considered to be a discrete-event system that can be modeled as a DFSA, where the set Q of states represents the status of an executable while executing; Σ is the (finite) alphabet of events which would cause the state transitions; $\delta : Q \times \Sigma \to Q$ is the state transition function; q_0 is the initial state of the model; and Q_m is the set of marked states that represents certain important stages in process execution.

Upon formulation of a DFSA model for operations of an executable, we derive a regular language $L(G)$ that is acceptable by the model.

$$L(G) = \{s \in \Sigma^* \mid \delta(q_0, s) \in Q\} \subseteq \Sigma^*$$

The marked language of G would then be represented as $L_m(G)$, where

$$L_m(G) = \{s \in \Sigma^* \mid \delta(q_0, s) \in Q_m\} \subseteq L(G)$$

The current state of the model corresponds to the status of a computer program in execution. The system calls are the basic events that cause the state transition in the plant DFSA model. The event alphabet is divided into two disjoint subsets Σ_c and Σ_{uc}, controllable and uncontrollable event sets respectively. A supervisor controls the plant by disabling or re-enabling the controllable events depending on the control policy. Uncontrollable events cannot be disabled by the supervisor.

Figure 1 shows the plant DFSA model representing the process interactions. In most cases, the system calls are events that trigger state transitions. The events belonging to the alphabet Σ are listed in Table 1. We map the system calls into events and some of the events consist of more than one system call. The states are listed in Table 2. State 3 is the good marked state where the system wants to go. States 8, 9, 10, 11 and 12 are the bad marked states where the system does not want to terminate. State 1 is the idle state, i.e., the state when the process is not running. Once the process is started, it transits to state 2. At state 2, if the event happens to trigger the state transition to state 4, and then state 5', 7' and 8', then there is a probability that the process contains

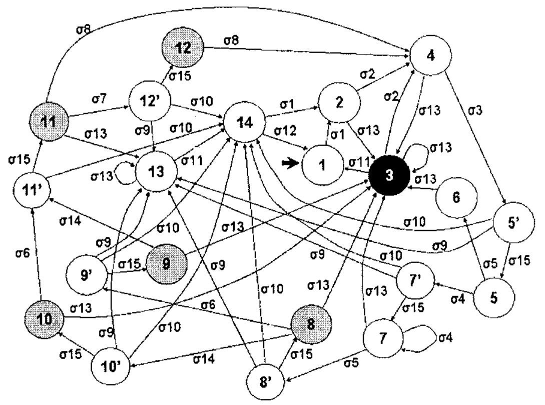


Figure 1: The unsupervised process.

Table 1: List of events in the DFSA

Event No	Event Name	Type	Description
σ_1	start	c	start executing a process
σ_2	findfirst getmod chmod open get_time	uc	Find and open the first file. Get time stamp and the mode of the file, and change the mode to write-able.
σ_3	read	uc	Read the file
σ_4	lseek/ write	uc	Specify the offset of a file from a certain position
σ_5	set_time	uc	Set the time stamp
σ_6	close chmod	uc	Change the file mode and close the file
σ_7	chdir	uc	Change directory
σ_8	Findnext getmod chmod open get_time	uc	Find next file
σ_9	jump	c	Jump to the normal part of the executable
σ_{10}	terminate	c	Terminate running process
σ_{11}	process-end	uc	A finish executing signal
σ_{12}	delete/recover	c	Recover the infected file, if impossible, delete it
σ_{13}	common system call	uc	At a certain state, it represents all the other system calls except the one(s) we specified at that state.
σ_{14}	chmod	un	Change the file mode
σ_{15}	continue	c	Continue the process without interference

c: controllable uc: uncontrollable

Table 2: List of States in the DFSA

State Number	Description
1	Idle and no process running
2	Start execution
3	Normal running condition
4	File Opened
5	File Read
6	Time stamp is set before the file is written
7'	File is written (decision pending)
7	File is written (continued)
8'	Time stamp is set after the file is written (decision pending)
8	Time stamp is set after the file is written (continued)
9'	File closed without changing the mode (decision pending)
9	File closed without changing the mode (continued)
10'	Mode changed without closing the file (decision pending)
10	Mode changed without closing the file (continued)
11'	File closed and mode changed (Virus is highly possible) (decision pending)
11	File closed and mode changed (Virus is highly possible) (continued)
12'	Directory is changed (Virus is detected) (decision pending)
12	Directory is changed (Virus is detected) (continued)
13	Jumped to normal part of the execution
14	Execution ended

malicious function; otherwise, it will come to state 3 and finally end at state 1 by event σ_{11}. If it then transits to state 11, there is a high probability that the running process contains a malicious code. If it transits to state 12, it is believed that the process contains malicious function. At the bad marked states, the supervisory actions are determined based on the control specifications. It may disable some of the controllable events and re-enable some other controllable events to avoid further damage. For example, the control specifications of the present supervised plant in Figure 2 disable event σ_{15} (i.e., continue the process without interference) when it triggers the transition from state 9' to state 9 and from state 10' to state10.

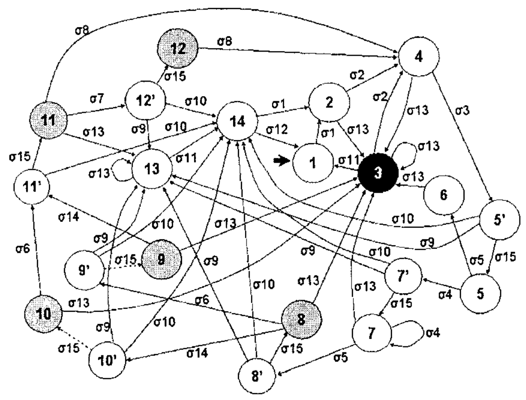
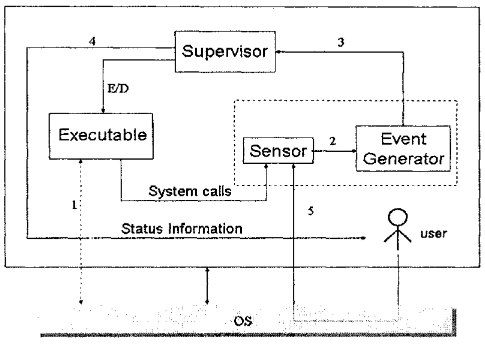


Figure 2: DFSA for Supervised Process.

3. SIMULATION RESULTS AND DISCUSSION

Figure 3 presents the structure of the implementation of the supervisory control system. There are three main modules: (1) *Sensor*, (2) *Event Generator*, and (3) *Supervisor*. The *Sensor* captures the system calls and user actions when a process is running and *Event Generator* receives a system call as an input, it maps the system call into an event and then feeds the event to the *Supervisor* to trigger the state transition of the DFSA. When the user receives the status information, he/she terminates the process, or let it execute without interference. The action of the user is also captured by the *Sensor* and is mapped as an event (it could be σ_9, σ_{10}, σ_{12} or σ_{15}) by *Event Generator*. We have created a data set of 20 benign executables and 60 malicious executables. The malicious executables are generated by three tools, VCL, G2, and BW.



E: Enable D: Disable

Steps:

1. Program executing.
2. *Sensor* sends the system calls and user actions to *Event Generator*.
3. *Event generator* feeds an event to the *Supervisor* to trigger the state transition.
4. When virus-like behavior is found, *Supervisor* informs user.
5. User action.

Figure 3: The structure of the simulation system.

These tools are available at website: *http://vx.netlux.org/dat/vct.shtml.* A source file of system calls is generated for each of the executable in its execution phase by use of the trace tool. To conduct simulation experiments, *Sensor* captures the system calls from the source file one by one, and then follows the steps shown in Figure 3.

When the simulation begins, a transition automatically takes place from state 1 to state 2. Similarly, when the simulation ends, the process terminates on state 1 or state 14, depending on which internal states it passes through. The top part of Figure 3 shows how *Event Generator* maps the system calls when it receives them from the *Sensor.* Once an event is generated, *Event Generator* feeds this information to the *Supervisor* to trigger the state transition. These phenomena are explained by four examples described below.

We illustrate the work of our system by three representative examples out of all experiments we did. In Example 1, all system calls are mapped to event σ_{13} so that the process stays at state 3, and finally ends at state 1. In Example 2, the DFSA model fails to detect a virus because *Event Generator* also maps all the system calls to σ_{13} according to the event definition in Table 1. The model works quite well on Example 3 and Example 4 for simulation on viruses generated by BW and G2, respectively. As the system transits to a user-pending state, and the user action is mapped as σ_{15}, the process continues running without interference. However, if σ_{15} is disabled, then it triggers the state transition from state 9' to state 9 and from state 10' to state 10. Consequently, further possible damage is avoided.

Simulation results are summarized in Table 3 to show that the frequency of false negative is zero, and that the frequency of false positive is dependent on the virus creation tool being used. It could detect classes of viruses generated by G2 and BW very well and can not detect with viruses generated by VCL tools.

Table 3: Simulation Results

	Benign program	Virus or virus infected programs		
		VCL	BW	G2
No. of programs	20	20	20	20
Detected as virus	0	0	20	20

When all the sample programs are equally distributed (i.e., the number of malicious programs generated by each tool and the number of benign programs are the same as Table 3 shows), the overall frequency of correct detection is about 75%. The rationale for this somewhat reduced frequency of correct detection is that the sensitive-system calls do not fully cover all categories of viruses. The results are very likely to improve through more appropriate definition of events.

4. SUMMARY AND CONCLUSIONS

This paper presents a novel approach based on discrete-event supervisory control to detect and prevent the spread of malicious executables. This approach looks for virus-like behavior rather than specific viruses. The simulation results show that this is an effective method.

This work is a novel extension of the SCT theory presented by Ramadge and Wonham (1987) for the control of malicious executables. The supervisor makes use of a feedback mechanism based on a control policy and recognizes a (constrained legal) language as a result of this feedback mechanism. This paper models the execution of a process as a DFSA, and models the system calls, made in executing the process, as events that trigger state transitions. A supervisor algorithm is then formulated to monitor and control the process execution so that it could prevent the spread of a malicious code after its detection. Future work includes detailed event definition so that our system can detect VCL and other categories of viruses.

REFERENCES

Alan Solomon, Tim Kay (1994). *Dr.Solomon's PC Anti-virus Book*, published by Newtech, Oxford.

Bandouin Le Charlier, Morton Swimmer (1995). Dynamic detection and classification of computer viruses using general behavior patterns. In *Proceedings of Fifth International Virus Bulletin Conference*, p75 Boston, September 20-22.

C.C. Michael, Anup Ghosh (2000). Using finite automata to mine execution data for intrusion detection: a preliminary report. *Proc. RAID 2000* (Springer LNCS 1907), pp 66-79. Oct.

C. Wallace, P. Jensen, N. Soparkar (1996). Supervisory Control of Workflow Scheduling. In *Proceedings of International Workshop on Advanced Transaction Models and Architectures*, 1996, Goa, August – September.

Christina Warrender, Stephanie Forrest, Brak Pearlmutter (1999). Detecting intrusions using system calls: alternative data Models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133-145, IEEE Computer Society.

Diomidis Spinellis (1994). Trace: A tool for logging operating system call transaction. *Operating System Review*, **28(4)**:56-63, October. *http://dmst.aueb.gr/dds/pubs/jrnl/1994-SIGOS-Trace/html/article.html* (Accessed March 22, 2002)

Eleazar Eskin, Wenke Lee (2001). Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of DISCEX II*, June, 2001.

G. Karsai, A. Ledeczi (2001). An approach to self adaptive software based on supervisory control. *IWSAS* 2001, Balatonfured, Hungary.

Koral Ilgun, Richard A. Kemmerer, Phillop A.Porras (1995). State transition analysis: a rule-based intrusion detection approach. *IEEE Transactions on software engineering*, Volume 21, Number 3, March.

Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo (2001). Data mining methods for detection of new malicious executables. In *proceedings of IEEE Symposium on Security and Privacy*. Oakland, CA: May.

Nittida Nuansri, Samar Singh, Tharam S. Dillon (1999). A process state-transition analysis and its application to intrusion detection. *ACSAC 1999*, p378-388.

P.J. Ramadge and W.M. Wonham(1987). Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, Vol.25, No. 1, January, 206-230.

R. Kumar, M. Fabian (1997). Supervisory control of partial specification arising in protocol conversion, *35th Allerton Conference on Communication, Control and computing*, 543-552, Urbana-Champaign, Illinois.

Richard B. Levin (1990). *The Computer Virus Handbook*, published by Osborne McGraw-Hill, ISBN 0-07-881047-5

Sandeep Kumar, Eugene H.Spafford (1992). A generic virus scanner in C++. *The Proceedings of the 8th Computer Security Applications Conference*, IEEE Press.

Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji (1998). Intrusion Detection using sequences of system calls. *Journal of Computer Security* **6(3)**: 151-180.

Steven T. Eckmann, Giovanni Vigna, Richard A. Kemmerer (2000). STATL: An attack language for state-based intrusion detection. In *Proceedings of the ACM Workshop on Intrusion Detection*, Athens, Greece, November,2000 ACM.

Wenke Lee, Salvatore J.Stolfo (1998). Data mining approaches for intrusion detection. In *Proceedings of the Seventh USENIX Security Symposium (SECURITY '98)*, San Antonio, TX, January,1998.

Xi. Wang, A. Ray, S. Phoha and J. Liu. (2002). J-DES: A Graphical Interactive Package for the Synthesis and Analysis of Discrete Event Systems. In *Proceedings of American Control Conference*, Denver, Colorado, June.