Hindawi
Mobile Information Systems
Volume 2017, Article ID 2301970, 15 pages
https://doi.org/10.1155/2017/2301970

Research Article

Study of Immune-Based Intrusion Detection Technology in Virtual Machines for Cloud Computing Environment

Ruirui Zhang[1] and Xin Xiao[2]

[1]*School of Business, Sichuan Agricultural University, Chengdu 610000, China*
[2]*School of Computer Science, Southwest Minzu University, Chengdu 610000, China*

Correspondence should be addressed to Ruirui Zhang; zhangruiruisw@gmail.com

Received 5 May 2017; Revised 16 August 2017; Accepted 10 September 2017; Published 23 October 2017

Academic Editor: Laurence T. Yang

Copyright © 2017 Ruirui Zhang and Xin Xiao. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Cloud computing platforms are usually based on virtual machines as the underlying architecture; the security of virtual machine systems is the core of cloud computing security. This paper presents an immune-based intrusion detection model in virtual machines of cloud computing environment, denoted as IB-IDS, to ensure the safety of user-level applications in client virtual machines. In the model, system call sequences and their parameters of processes are used, and environment information in the client virtual machines is extracted. Then the model simulates immune responses to ensure the state of user-level programs, which can detect attacks on the dynamic runtime of applications and has high real-time performance. There are five modules in the model: antigen presenting module, signal acquisition module, immune response module, signal measurement module, and information monitoring module, which are distributed into different levels of virtual machine environment. Performance analysis and experimental results show that the model brings a small performance overhead for the virtual machine system and has a good detection performance. It is applicable to judge the state of user-level application in guest virtual machine, and it is feasible to use it to increase the user-level security in software services of cloud computing platform.

1. Introduction

Cloud computing has become the mainstream of the next generation of information technology; it provides a new and economic technology of allocating and using computing resources. Due to huge scale, complex software and hardware structure, third-party data storage, and unprecedented openness and complexity in cloud computing systems, it makes the security of cloud computing stricter than traditional information systems. If security issues cannot be well solved, it will seriously restrict the rapid development of cloud computing and the popularity of cloud computing applications.

Cloud computing platforms are usually based on virtual machines as the underlying architecture; the security of virtual machine systems is the core of cloud computing security. At present, there are few security researches on virtual machine system in cloud computing environment, and existing researches are briefly introduced.

Haeberlen et al. put forward the concept of accountable virtual machines (AVMs) [1], in which programs are executed and related information is recorded to determine whether programs are normal. This method belongs to static assessment and cannot detect the real-time safety of programs.

Payne et al. [2] presented the Lares system, inserting a hook function in the client virtual machine which can proactively monitor events of client virtual machine (VM). This hook function can trigger safety program of security virtual machine (privileged VM) which make decisions for events of client VM. The monitoring program is located within the secure VM and out of the client VM. Therefore, it belongs to the out-of-VM monitoring method. This method is of high security but requires frequent contexts switching between virtual machines, which brings greater performance cost and especially does not apply to fine-grained monitoring.

Sharif et al. [3] put forward a common in VM monitoring framework, in which monitoring and judging processes

run in untrusted guest VM. In order to achieve the same security with out-of-VM monitoring method, this framework uses hardware memory protection mechanism and hardware virtualization technology. In the guest VM, a memory space protected by the VM monitor is divided and used by the safety monitoring program under controlled conditions. This framework requires hardware virtualization support.

Wang et al. [4] put forward a lightweight system named HookSafe based on VM monitor, which is mainly used to monitor the rootkit attacks of kernel spaces. Rootkit attack modifies the control data or hook function address. Hooks are often dynamically allocated with other data and distributed in noncontiguous memory areas, which needs byte-level granularity protection, while current hardware protection mechanism only provides page-level granularity. To solve the problem, Hooksafe introduced a hook function jump layer, which maps hooks to a contiguous page-aligned memory space and then uses the hardware protection mechanism to control access to this block of memory area.

The work in [5, 6] is also used to detect kernel rootkits. The work in [5] monitors invariants in controlled flow transferring and constant relationships in data of uncontrolled flow. The work in [6] adopts the Daikon tool to deduce invariants from data structures which are extracted from memory pages and monitors these invariants to determine the state of kernels.

Bharadwaja et al. [7] analyzed the security issues raised by hypercalls in virtualized environments and proposed a Xen-based distributed intrusion detection system, which implemented filtering operations on hypercalls in the privileged domain to achieve security.

Srivastava et al. [8] studied the use of rootkit to fuzzed system calls for virtual machine monitor (VMM) attacks and proposed a Xen-based monitoring system named Sherlock. The system overlooks call flows by increasing observation points in the process of kernel implementation and automatically adjusts the sensitivity according to security needs.

Szefer et al. [9] proposed the NoHype system. The system does not require too much involvement of VMM, runs VM directly on the underlying hardware, and maintains multiple virtual machines, in order to reduce the possibility of attacks between virtual machines and security threats caused by vulnerabilities of VMM. The main ideas are as follows: preallocating processor and memory resources, use of virtualization I/O device, small modifications of the client OS to perform examinations in the system boot process, and preventing the client VM from indirect contact with the hardware.

Benzina and Goubault-Larrecq [10] pointed out that Domain 0 is an important loophole of virtualization system and proposed a role-based access control model. This model describes unnecessary activity streams by simple timing formulas, which reduces threats of Domain 0 attacks, such as Trojan horses.

Wang et al. [11] proposed a detection method of hidden processes which is based on VMM. This method runs the detection tool out of the VM to be monitored and has high security. It gets the underlying status information of VMs to be monitored through VM introspection mechanism and reconstructs process queues to determine malicious processes.

The above works studied security of user procedures in VM and vulnerabilities of VMM and proposed corresponding defensive methods. However, through careful analysis, current methods cannot accurately determine the real-time status of client VM applications or the security vulnerabilities of VMM. Most of proposed methods are for particular attacks and vulnerabilities and cannot effectively deal with threats of other attacks.

Inspired by the immune response mechanism and the danger theory of the biological immune system, this paper presents an immune-based intrusion detection model in virtual machines of the cloud computing environment, named IB-IDS. The main contributions of this model are as follows. (1) The model introduces the danger theory into VM intrusion detection and defines the implementation of danger signals; (2) the model can monitor the state of applications and detect attacks on the dynamic runtime of applications, which has high real-time performance; (3) the model monitors the whole intrusion detection process and makes sure that every module of the model is safely running; (4) immune evolution mechanism and performance analysis of the model are described, which shows that the model is effective theoretically. The remainder of this paper is organized as follows. The theories of the model including description of the architecture, definitions of the model, implementation mechanism of danger signals, implementation mechanism of information monitoring, and the immune evolution model are described in Section 2. Performance analysis of the model is showed in Section 3. The effectiveness of IB-IDS is verified in Section 4. Finally, the conclusion is given in the last section.

2. Theories of the Model

Virtualization technology is the foundation of cloud computing. With the popularity of cloud computing, it has received more and more attention. Virtualization technology is achieved when there are many virtual machines in one physical machine, and each virtual machine runs different operating systems and applications and has good isolation with other virtual machines. These are implemented by adding a layer of software called virtual machine monitor (VMM) to the hardware. There is usually a virtual machine with a relatively high authority, called privileged virtual machine (privileged VM), which can manage and control other client virtual machines (Guest VM) to a certain extent. Xen [12, 13] was developed by the University of Cambridge's computer laboratory. It is an open-source project, is therefore widely used in academic research, and is also based on a number of cloud computing platforms, such as Amazon EC2 Service and Eucalyptus. In Xen, VMM is called hypervisor, and VM is called domain: the first domain which starts together with the hypervisor is called dom0, and other domain is called domU, which is shown in Figure 1.

For a virtual machine system, the most common attacks are basically completed using some certain vulnerabilities of the system. And these attacks are performed by a program or

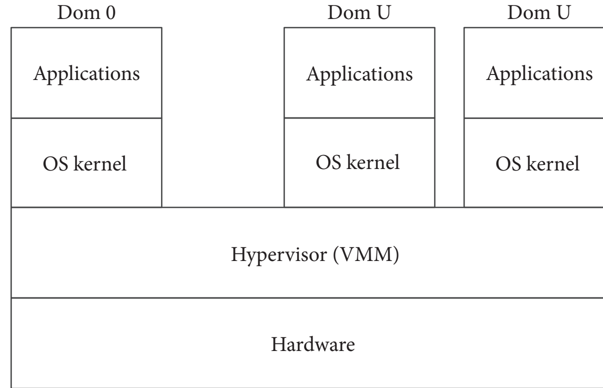


Figure 1: Xen virtual machine system.

software, which is called malware (malicious software). Common malwares are viruses, worms, Trojan horses, and rootkits. Some of them are user-state malicious processes which do not affect the operating system kernel; some are lurking in the kernel or process and modifying the memory space. When the system has no defense, it is vulnerable to be attacked. For example, when a program runs, we cannot be sure that the dynamic data structure changes in the inner core of the region are reasonable or because of the invasion. The proposed model can detect these kinds of malware.

2.1. Description of the Architecture. Due to the high privilege levels and relatively streamlined structure of the privileged VM and the hypervisor, it is assumed that these two are safe. The main intention of this model is to ensure the safety of user-level applications of guest VM. The architecture of IB-IDS is shown in Figure 2. This architecture is divided into four levels: the underlying hardware layer, the VMM layer, the privileged VM layer, and the guest VM layer. Modules of the model are distributed into these four levels. In order to reduce context switching between dom0 and domU and be able to do fine-grained monitoring, antigen presenting module and signal acquisition module are deployed in every guest VM. Immune response module and signal measurement module are deployed in the privileged VM. These two modules do not need communicating with domU and just get data on a regular basis during execution and are deployed separately in dom0, which can reduce the performance cost and improve the security of dom0. Information monitoring module is deployed in VMM. Because the guest VM is not credible, the model introduces the information monitoring module to supervise the running of antigen presenting module and signal acquisition module, to ensure the safety of the detection process.

The detection process is as follows. First, the antigen presenting module monitors executions of user-level applications in client VMs, extracts critical data as antigens, and delivers them to the immune response module in privileged VM through inter-VM communication mechanism. Meanwhile, the signal acquisition module collects environmental information when the program executes and transmits to the signal measurement module in privilege VM. These operations are performed on a regular basis. Then, the immune response module evaluates whether to trigger secondary response based on the set of memory antibodies. If it does, invasion occurs. If the secondary response is not triggered, the signal measurement module will evaluate the current environment's risk rating through cloud model, produce danger signals of different degrees, and then determine whether the invasion happens. If it does, the model will start a further initial response to eliminate alien antigens. Information monitoring module periodically runs after the system starts through accessing memory spaces of antigen presentation module and signal acquisition module, in order to ensure that these two modules are not attacked.

2.2. Model Definition. In the software system of virtual machines, all the information in the end can be reduced to a binary string and the virtual machine intrusion detection is classification of the binary string according to certain rules and a priori knowledge. Define that the problem state space $\Omega = \bigcup_{i=1}^{\infty} \{0,1\}^i$. Based on biological immune principles, we define the virtual system platform as organism, client virtual machines as immunologic tissues, and the user programs in virtual machines as antigens. Define that $AG \subset \Omega$ is the collection of antigens. The aim of the virtual machine intrusion detection is to differentiate patterns. Given an input pattern x, $x \in AG$, the system detects and makes sure whether this pattern belongs to a self or a nonself. There are two mistakes in the process of testing: false negative, which sorts nonselves for selves; false positive, which classifies selves as nonselves.

Forrest et al. [14] found out that the execution of critical programs can be described by the sequence of system calls, which is also called the execution trace. The situation of system calls can reflect behavioral characteristics of the program to some extent, and the execution trace has a local stability when the program is running. Taking system calls and their parameters into account, which are up to six in the Linux system regulation, we define the process ID, the short sequence of system calls, and their parameters as gene fragments of antigens.

Definition 1. The antigen is defined as a triple $\mathrm{ag} = \langle \mathrm{gid}, \mathrm{pid}, \langle x_1, x_2, \ldots, x_k \rangle \rangle$, which represents the feature vector in the solution space of the problem domain.

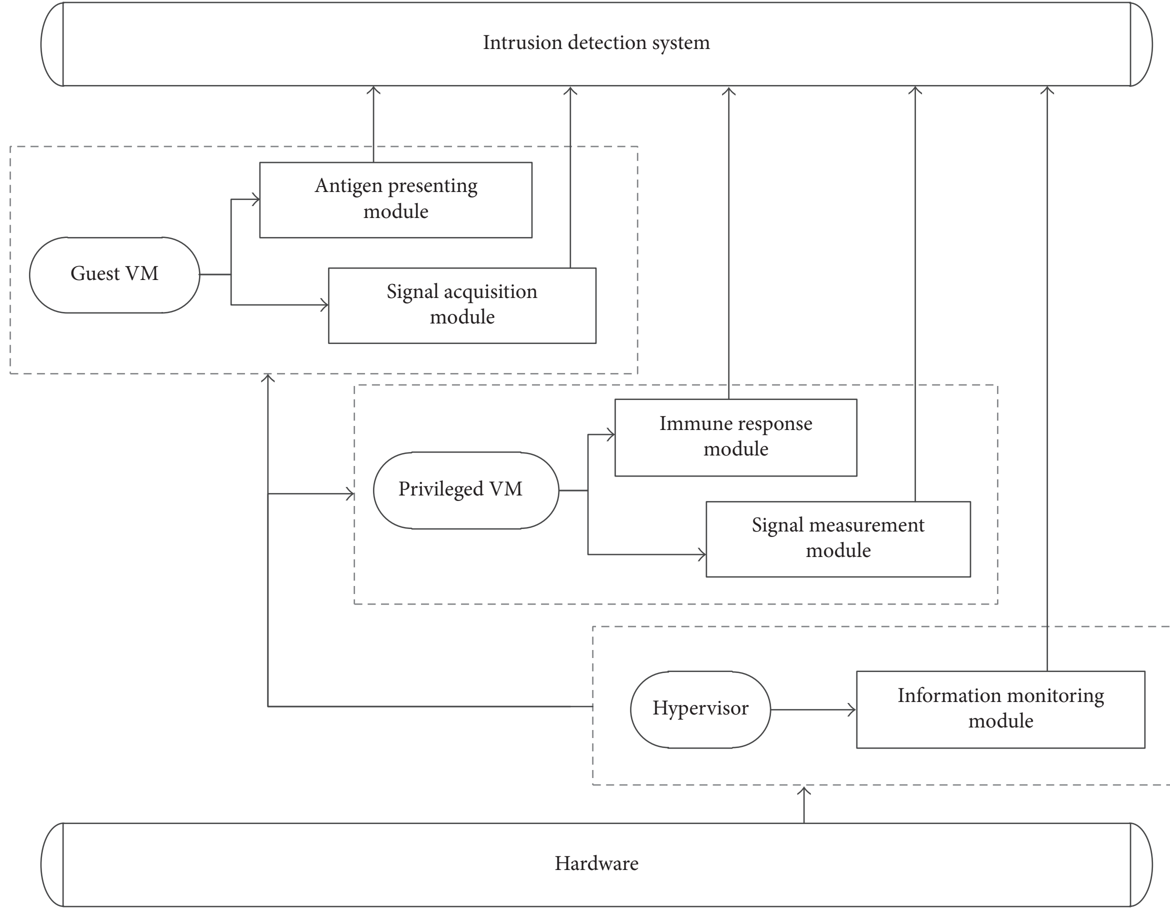


Figure 2: Structure of the intrusion detection model.

gid is the unique ID which identifies the client VM. pid is the process ID. $x_i = \langle \mathrm{sid}_i, p_{i1}, p_{i2}, \ldots, p_{il} \rangle$ $(i = 1, 2, \ldots, k)$ is the gene fragments of antigens. sid is the system call ID. k is the length of the short system call sequence, that is to say, the encoded length of immune cells, which reflects order relationships of system calls during the execution process. p_{ij} is the parameter of a system call, $i = 1, 2, \ldots, k$, $j = 1, 2, \ldots, l$. l is the number of parameters. All the antigens in the space compose a collection $\mathrm{AG} = \bigcup_{i=1}^{\infty} \{\mathrm{ag}_i\}$.

It is assumed that normal short sequences that can be recognized by the model are defined as self set S, all the unknown short sequences are defined as N, abnormal short sequences that produce danger signals are defined as D, and short sequences that are judged as invasions are defined as I.

Then, $S \cap N = \emptyset$, $S \cup N = \mathrm{AG}$. Danger theory does not distinguish between self and nonself, only recognizes intrusion set $I = D \cap N$ which triggers immune responses, and does not respond to harmless set $D \cap S$.

Definition 2. Antibodies can recognize antigens and trigger specific immune responses. Antibodies have the same structure as antigens, are used for detecting and matching antigens, and are expressed as $\mathrm{ab} = \langle \mathrm{gid}, \mathrm{pid}, \langle x_1, x_2, \ldots, x_k \rangle \rangle$. The set of antibodies are defined as $\mathrm{AB} = \bigcup_{i=1}^{\infty} \{\mathrm{ab}_i\}$.

Definition 3. The matching rule which is the affinity of antibody and antigen is indicated as the binding strength between antibody and antigen. In this paper, we propose an improved r-continuous bit matching method:

$$\mathrm{affinity}(\mathrm{ab}, \mathrm{ag}) = \begin{cases} 1, & \sum_{i=1}^{k} \dfrac{f(\mathrm{ab}.x_i, \mathrm{ag})}{k} \geq \beta, \ \mathrm{ag.gid} = \mathrm{ab.gid}, \ \mathrm{ag.pid} = \mathrm{ab.pid} \\ 0, & \text{others}, \end{cases} \quad (1)$$

where β is the value of matching threshold and $f(x, y)$ is r-continuous bit matching method between antibody gene fragment x_i and antigen:

$$f(x, y) = \begin{cases} 1, & \exists i, j, j - i \geq |x|, \ 0 < i \leq j \leq k \cdot (l + 1), \ x_i = y_j, x_{i+1} = y_{j+1}, \ldots, x_{|x|} = y_{j+|x|-1} \\ 0, & \text{others}. \end{cases} \quad (2)$$

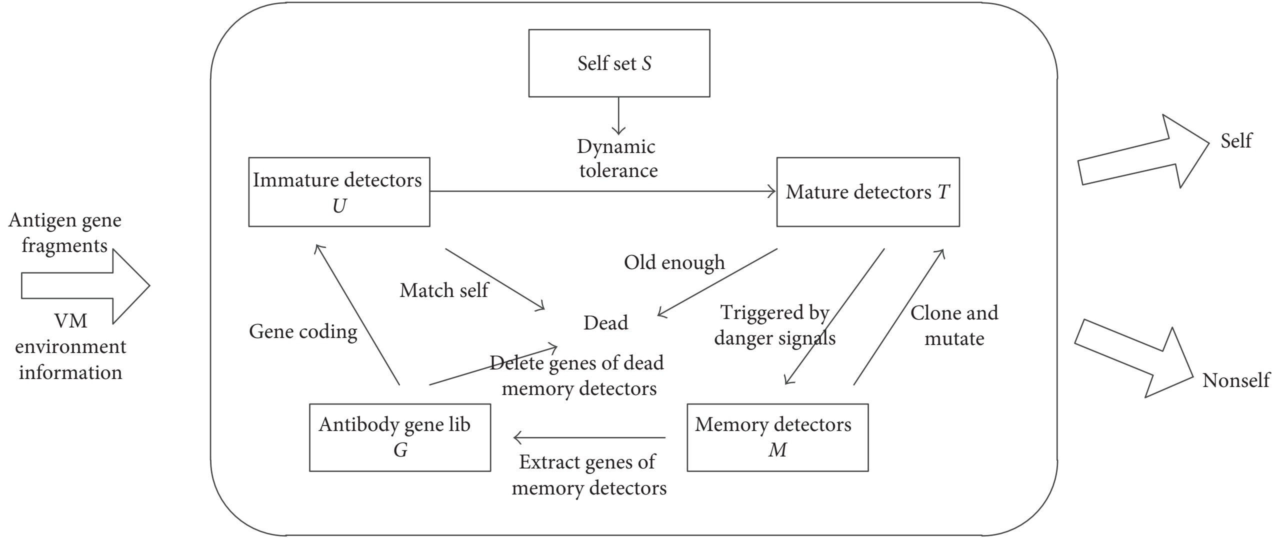


Figure 3: The immune mechanism of the model.

Definition 4. Detector set is defined as $B = \{\langle \text{ab}, \text{age} \rangle \mid \text{ab} \in \text{AB} \cap \text{age} \le \text{age}_{\max}\}$, where ab is antibody of the detector, age is the age of the detector, and $\text{age}_{\max}$ is the maximum age of the detector. The detector set consists of immature detectors, mature detectors, and memory detectors. The immature detector which is not subjected to self-tolerance will evolve into a mature one when it passes self-tolerance. The mature detector will become a memory one after it is activated.

The immature detector set is defined as $U = \{x \mid x \in B \cap x.\text{age} < \gamma\}$, where γ simulates tolerance period. The mature detector set is defined as $T = \{x \mid x \in B \cap \gamma \le x.\text{age} < \text{age}_{\max} \cap \forall \text{ag} \in S(\text{affinity}(x.\text{ab}, \text{ag}) = 0)\}$. The memory detector set is defined as $M = \{x \mid x \in B \cap x.\text{age} = \text{age}_{\max} \cap \forall \text{ag} \in S(\text{affinity}(x.\text{ab}, \text{ag}) = 0)\}$.

In the detector generation process, if $\text{Affinity}(x, \text{ag}) = 1$ $(\text{ag} \in S)$, the detector x can describe self and triggers immune self-reaction, which must be removed. In the end of the process, remaining detectors only can describe elements of the nonself set. In the detection process, if $\text{Affinity}(x, \text{ag}) = 1$ $(\text{ag} \in I)$, antigen ag can be described by detector x, triggering the immune response.

We use Figure 3 to represent the immune mechanism of the model. In the model, a new immature detector is generated by gene coding, and the immature detector evolves into a mature detector by negative selection (self-tolerance). If it matches selves, it dies. Mature detector has fixed length of the life cycle. If it is activated by danger signals in the life cycle, it evolves into the memory detector and generates first response; otherwise, it dies (deleting those detectors which are useless against antigens). The memory detector has a long life cycle, and once it is matched to an antigen, it will be activated immediately and produce second response.

2.3. Implementation Mechanism of Danger Signals. Danger theory emphasizes that danger signals which are generated from environmental changes result in various degrees of immune response, and the area around signals is called danger zone. The most important issue of introducing danger theory into intrusion detection systems is the definition of danger signals, which is how to determine the danger. In a virtual machine environment, we select the number of regular files of system variable N_{reg}, the memory ratio used by a process Rss, and the number of files reported by lsof command N_{files}, these three environmental values as assessments of danger signals, and normalize them to real value intervals between $[0, 100]$.

For antigen ag_i, define the function of danger signal $\text{DS}(\text{ag}_i)$ below. This function takes the three environmental values N_{reg}, Rss, and N_{files} as inputs and then generates signal values where the antigen is.

$$\text{DS}(\text{ag}_i) = \frac{(k_1 N_{\text{reg}} + k_2 Rss - k_3 N_{\text{files}})}{(k_1 + k_2 + k_3)}. \quad (3)$$

As can be seen, N_{reg} and Rss will have a negative influence on the environment, and the increase of N_{reg} and Rss shows that the environment is damaged or the possibility of being damaged is larger. N_{files} will have a positive influence on the environment, and the increase of N_{files} shows that the possibility of the environment being normal is larger.

The size of the danger zone limits the scope of the immune response, and immune cells in the region will be activated to participate in the immune response. For antigen ag_i, define the function of the danger zone $DA(ag_i)$ below. This function returns a collection of detectors whose distance from ag_i is less than r_danger:

$$DA(ag_i) = \left\{ x \mid \frac{1}{\left(\sum_{j=1}^{k} f\left(x.ab.x_j, ag_i\right)/k\right)} \leq r_danger \cap x \in T \right\}, \tag{4}$$

where r_danger is the radius of the danger zone.

How to determine whether the environment is damaged according to danger signals? We took advantage of the cloud model to evaluate. The cloud model [15] is a probabilistic reasoning tool and is a mathematical transformation model between the qualitative concept expressed by language values and quantitative data, which has three numerical characteristics: expectation Ex, entropy En, and hyperentropy He. Based on the danger signal modeling, we use cloud rule generator and reverse cloud generator to carry out qualitative analysis of environments of guest virtual machines. Rule generator can be divided into front cloud and rear cloud. IF part is the condition of the rule, which is achieved by the front cloud, while THEN part is a result of the rule, which is implemented by the rear cloud. The inputs of front cloud are values to be seized, and the output is the membership of some rule activated by samples, which is also input of rear cloud, and the output of rear cloud is the conclusion of the rule.

First, danger signals $DS(ag_i)$ were sampled m times in a safe state and an attacked state. Based on obtained cloud droplets, we got numerical characteristics of front cloud $\{Ex_{si}, En_{si}, He_{si}\}$ and $\{Ex_{di}, En_{di}, He_{di}\}$ through reverse cloud generator. If the secure state cloud and dangerous state cloud cover the entire state space, then we can use these two clouds to determine the status of the system. This is an ideal situation. If these two clouds cannot cover the whole state space, we need to divide the empty part, and it can be divided into weak secure state cloud and weak dangerous state cloud. In general, the closer it is to the center of discourse domain, the smaller the entropy and hyperentropy of clouds are; the more it is distant from the center, the larger the entropy and hyperentropy are. For two clouds which are next to each other, entropy and hyperentropy of the smaller one are 0.618 times of the greater one. That is the empirical value. So we can get En_{lsi}, En_{ldi}, He_{lsi}, He_{ldi}. According to the "3En rules" of the cloud model, we can estimate expectations of weak secure state cloud and weak dangerous state cloud. Formulas are as follows:

$$Ex_{lsi} = Ex_{si} + 3En_{lsi} = Ex_{si} + 3 * 0.618En_{si}, \tag{5}$$

$$Ex_{ldi} = Ex_{di} - 3En_{ldi} = Ex_{di} - 3 * 0.618En_{di}. \tag{6}$$

We design rules listing in the following to build the rule generator. Then we can get the environment and the level of membership according to actual value of danger signals.

Rule 1. IF danger signal indicator is low, THEN the system is safe and does not elicit the immune response, and the corresponding antibody can be deleted.

Rule 2. IF danger signal indicator is comparatively low, THEN the system is relatively safe and does not elicit an immune response.

Rule 3. IF danger signal indicator is comparatively high, THEN the system is relatively in danger and elicits an immune response.

Rule 4. IF danger signal indicator is high, THEN the system is in danger, elicits an immune response, and adds corresponding mature antibody into the memory antibody collection.

When the system triggers the secondary response or danger signals trigger the initial response, antibodies will mutate based on the immune response mechanism to generate new antibodies which have higher affinity with original antigens in order to more quickly identify danger and also generate antibodies which have lower affinity to add into immature antibody collection in order to ensure the diversity of the immune system.

2.4. Implementation Mechanism of Information Monitoring. Antigen presenting module and signal acquisition module are deployed in domU. Because Linux is an open-source operating system, we can add these two modules into domU's kernel. Information monitoring module is deployed in VMM. To ensure antigen presentation module and signal acquisition module's safety, the model accesses memory spaces which they belong to and performs hash computing of the memory data. The implementation mechanism needs to solve two important issues. The first one is how to find the memory space which antigen presenting module and signal acquisition module belong to and the second is how to use hashing to ensure that the two modules are not attacked.

VMM is responsible for managing and distributing various hardware resources and provides virtual hardware resources for the upper operating system kernel. domU accesses the physical memory through VMM. In Linux system, system.map file is a specific kernel symbol table and lists all the kernel symbolic names and their corresponding virtual addresses. A kernel symbol may be a variable name or a function name. Since antigen presenting module and signal acquisition module are in domU's kernel space, all the variables and functions which they contain can be found in system.map; that is to say, we can find virtual memory addresses of these variables and functions in domU. In Xen system, there are three memory structures which are virtual memory, pseudophysical memory, and machine memory. Virtual memory means that each process has a separate virtual memory address space. Pseudophysical memory locates

between virtual memory and machine memory, and each operating system of domUs believes that pseudophysical memory is "physical memory." In fact, machine memory is real physical memory. VMM maintains a M2P (Machine to Physical) global conversion table, and each domU maintains a P2M (Physical to Machine) partial conversion table. As can be seen, we can find the pseudophysical address corresponding to virtual memory address through domU's page table and find machine address corresponding to pseudophysical address through domU's P2M table.

Through the above method, we can find the memory space to which antigen presenting module and signal acquisition module belong. Information monitoring module reads contents of all initialized data, read-only data, and functions' memory which belong to the two modules in the order in accordance with the system.map file, as hash input. Hash computing can map binary value of arbitrary length to a shorter fixed-length binary value, and two different inputs cannot be mapped to the same value. Therefore, we use hash computing to ensure the integrity of memory spaces of antigen presenting module and signal acquisition module. In hypervisor, we define two variables hd_{ag} and hd_{sig}, which store cumulative hash values of antigen presenting module and signal acquisition module, and they are calculated as follows:

$$\begin{aligned} hd_{ag}(i+1) &= \text{hash}\left(hd_{ag}(i) \,\&\, r_{ag}(i+1)\right), \\ hd_{sig}(j+1) &= \text{hash}\left(hd_{sig}(j) \,\&\, r_{sig}(j+1)\right). \end{aligned} \quad (7)$$

In (5), hash(x) is the hash function, & is a binary string concatenation operator, $r_{ag}(i)$ is the content of the ith memory segment of antigen presenting module, and $hd_{ag}(i)$ is the accumulative value after i times hash computing for antigen presenting module. Meaning of (6) is by analogy. We mark the final cumulative hash values of antigen presenting module and signal acquisition module stored by hypervisor in a safe state as standard values hd'_{ag} and hd'_{sig}. Information monitoring module periodically is executed. Through comparing hash values hd_{ag} and hd_{sig} which are obtained when the program is running with standard values, we can determine the security of antigen presenting module and signal acquisition module.

2.5. The Immune Evolution Model

2.5.1. Self-Evolution Model

$$\begin{aligned} S(t) &= \begin{cases} S_{\text{first}}, & t = 0 \\ S(t-1), & t \bmod \delta \neq 0 \\ S(t-1) \cup S_{\text{new}}(t) - S_{\text{unload}}(t) - S_{\text{dead}}(t), & t > 0 \cap t \bmod \delta = 0, \end{cases} \\ S_{\text{dead}}(t) &= \begin{cases} \emptyset, & S(t-1) \cup S_{\text{new}}(t) - S_{\text{unload}}(t) < \text{size}_{\max} \\ \{\text{ag} \mid \text{ag} \in S(t-1) \cap \text{Eliminate } \left|S_{\text{new}}(t) - S_{\text{unload}}(t)\right| \text{ elements according to some principles}\}, & \text{others}, \end{cases} \end{aligned} \quad (8)$$

where $S(t), S(t-1) \subset S$, t, respectively, express the self set in the moment of t and $t-1$. S_{first} is the self set in the initial moment. δ is the evolutionary cycle of selves. In the δ cycle, the self set remains unchanged; in the end of δ period, new elements S_{new} will complement, such as loading new programs, those programs $S_{\text{unload}}(t)$ that have been uninstalled will be deleted, and part of selves $S_{\text{dead}}(t)$ will be eliminated, in order to avoid increases of self set without limit.

The computer software system is a huge collection. The self set of a complete software system is too large for the calculation ability at the present stage of computer, and it is very difficult to find an absolute reliable self set in the dynamic software system. The evolution of the self set can make the model only need to maintain a smaller set of selves, to ensure higher time efficiency according to the existing computing capacity. In addition, because of the continuous evolution of selves, nonself elements which mix into selves will eventually be removed, reducing the rate of false negative caused by incomplete self set.

2.5.2. Antibody Gene Lib Evolution Model

$$G(t) = \begin{cases} G_{\text{first}}, & t = 0 \\ G(t-1) - G_{\text{dead}}(t) \cup G_{\text{new}}(t), & t > 0, \end{cases} \quad (9)$$

where $G(t), G(t-1) \subset G$, respectively, express the set of antibody gene lib in the moment of t and $t-1$. G_{first} is the initial antibody gene collection, which are gene fragments of these typical kinds of malware. $G_{\text{dead}}(t) = \bigcup_{x \in M_{\text{dead}}(t)} \bigcup_{i=1}^{k} \{x.\text{ag}.x_i\}$ is set of mutated genes which should be removed in the time of t. $M_{\text{dead}}(t)$ is set of memory detectors with false positive. When mature detector is cloned, its gene $G_{\text{new}}(t) = \bigcup_{x \in T_{\text{cloned}}(t)} \bigcup_{i=1}^{k} \{x.\text{ag}.x_i\}$ will join the antibody gene library as the dominant gene. $T_{\text{cloned}}(t)$ is set of activated mature detectors.

Antibody gene lib is mainly used to improve the generation efficiency of immature detectors. In the generation process of new immature detectors, their antibodies are produced by gene encoding measures, so they have the ability

to detect known malware variants, reducing the tolerance time. The use of genetic coding produces "Baldwin effect": evolution and learning will enable new individuals to acquire some of the same characteristics, reducing the diversity of the system. In order to solve this problem, a certain proportion of randomly generated immature detectors are added to ensure the diversity of the system.

2.5.3. Immature Detectors Evolution Model

$$\begin{aligned} U(t) &= \begin{cases} \emptyset, & t=0 \\ f_{\text{age}}(U(t-1)) - \left(U_{\text{untolerance}}(t) \cup U_{\text{matured}}(t)\right) \cup U_{\text{new}}(t), & t>0, \end{cases} \\ U_{\text{untolerance}}(t) &= \left\{x \mid x \in f_{\text{age}}(U(t-1)) \cap \exists y \in S(t-1)\ \left(\text{affinity}(x.\text{ab}, y) = 1\right)\right\}, \\ U_{\text{matured}}(t) &= \left\{x \mid x \in f_{\text{age}}\left(U(t-1) - U_{\text{untolerance}}(t)\right) \cap x.\text{age} > \gamma\right\}, \end{aligned} \tag{10}$$

where $U(t), U(t-1) \subset U$, respectively, express set of immature detectors in the moment of t and $t-1$. $f_{\text{age}}(X)$ $(X \subset B)$ means adding 1 to the age of every detector in X. $U_{\text{untolerance}}(t)$ is set of immature detectors which does not pass self-tolerance, and $U_{\text{matured}}(t)$ is set of mature detectors which pass self-tolerance. $U_{\text{new}}(t)$ is newly created immature detectors in the time t and includes two parts: completely random-generated detectors (to ensure diversity) and detectors generated by genes encoding in the antibody gene lib (to ensure availability).

2.5.4. Mature Detectors Evolution Model

$$\begin{aligned} T(t) &= \begin{cases} \emptyset, & t=0 \\ \left(f_{\text{age}}(T(t-1)) - \left(T_{\text{dead}}(t) \cup T_{\text{cloned}}(t)\right)\right) \cup U_{\text{matured}}(t) \cup T_{\text{permutation}}(t), & t>0, \end{cases} \\ T_{\text{dead}}(t) &= \left\{x \mid x \in f_{\text{age}}(T(t-1)) \cap x.\text{age} = \text{age}_{\max} \cap \nexists y \in N(t-1)\ \left(x \in \text{DA}(y)\right)\right\}, \\ T_{\text{cloned}}(t) &= \left\{x \mid x \in \left(f_{\text{age}}(T(t-1)) - T_{\text{dead}}(t)\right) \cap \exists y \in N(t-1)\ \left(x \in \text{DA}(y)\right)\right\}, \\ T_{\text{permutation}}(t) &= f_{\text{clone_mutation}}\left(T_{\text{cloned}}(t) \cup M_{\text{cloned}}(t)\right), \end{aligned} \tag{11}$$

where $T(t), T(t-1) \subset T$, respectively, express the set of mature detectors in the moment of t and $t-1$. $T_{\text{dead}}(t)$ is set of mature detectors which are not activated at the end of the life cycle. $T_{\text{cloned}}(t)$ is set of mature detectors activated by danger signals. $U_{\text{matured}}(t)$ is set of new mature detectors. $T_{\text{permutation}}(t)$ is set of mature detectors which are produced by clonal mutation of activated ones. $f_{\text{clone_mutation}}(X)$ $(X \subset T)$ is clonal variation equation and executes clone and mutation operation for each element x in X.

2.5.5. Memory Detectors Evolution Model

$$\begin{aligned} M(t) &= \begin{cases} M_{\text{first}}, & t=0 \\ \left(M(t-1) - M_{\text{dead}}(t)\right) \cup f_{\text{age2}}\left(M_{\text{cloned}}(t)\right), & t>0, \end{cases} \\ M_{\text{dead}}(t) &= \{x \mid x \in M(t-1) \cap \exists y \in S(t-1)\ \left(\text{affinity}(x.\text{ab}, y) = 1\right)\}, \\ M_{\text{cloned}}(t) &= \{x \mid x \in M(t-1) \cap \exists y \in N(t-1)\ \left(x \in \text{DA}(y)\right)\}, \end{aligned} \tag{12}$$

where $M(t), M(t-1) \subset M$, respectively, express the set of memory detectors in the moment of t and $t-1$. M_{first} is set of initial memory detectors. These detectors can be obtained from common malwares. $M_{\text{dead}}(t)$ is set of memory detectors with false positive in the moment t. $f_{\text{age2}}(M_{\text{cloned}}(t))$ expresses set of newly created memory detectors. $f_{\text{age2}}(X)$ $(X \subset B)$ sets the age of each detector in X to $\text{age}_{\max}$. $M_{\text{cloned}}(t)$ is set of activated memory detectors in the time t.

2.5.6. Antigen Detection

$$\begin{aligned} \text{AG}(t) &= \begin{cases} \text{AG}_{\text{first}}, & t=0 \\ \left(\text{AG}(t-1) - \text{AG}_{\text{self}}(t) - \text{AG}_{\text{nonself}}(t)\right) \cup \text{AG}_{\text{new}}(t), & t>0, \end{cases} \\ \text{AG}_{\text{nonself}}(t) &= \{x \mid x \in \text{AG}_{\text{checked}}(t) \cap \exists y \in \left(T_{\text{cloned}}(t) \cup M_{\text{cloned}}(t)\right)\ \left(\text{affinity}(y.\text{ab}, x) = 1\right)\}, \\ \text{AG}_{\text{self}}(t) &= \{x \mid x \in \text{AG}_{\text{checked}}(t) \cap \forall y \in (T(t) \cup M(t))\ \left(\text{affinity}(y.\text{ab}, x) = 0\right)\}, \end{aligned} \tag{13}$$

where $AG(t), AG(t-1) \subset AG$, respectively, express the set of antigens in the moment of t and $t-1$. AG_{first} is set of initial antigens. $AG_{checked}(t) \subset AG(t)$ expresses antigens to be checked in the moment t.

3. Performance Analysis of the Model

Set the number of programs in a computer as N_p, and usually the proportion of nonselves is ρ. The size of the self set is $|S|$, the size of the mature detector set is $|T|$, and the size of the memory detector set is $|M|$. The matching probability between any given detector and any given antigen is P_m (which is related to the specific matching rule). $P(A)$ is the probability of occurrence of event A.

Theorem 5. *For any detector which passes the self-tolerance, the probability of this detector matching those selves which are not described is* $P_n = (1-P_m)^{|S|} \cdot (1-(1-P_m)^{N_p\cdot(1-\rho)-|S|})$.

Proof. Set that A is event "the given detector does not match any self in the self set," and B is event "the given detector matches at least one self in the un-described self set." It is clear that the detector from A is self-tolerated and the detector from B may be not self-tolerated. $P_n = P(A)P(B)$. In the event A, the number of times X that detectors match selves meets the binomial distribution, that is to say, $X \sim b(n,p)$, where, $n = |S|$, $p = P_m$. Then, $P(A) = P(X=0) = (P_m)^0(1-P_m)^{|S|} = (1-P_m)^{|S|}$. In a similar way, in the event B, the number of times Y that detectors match selves meets the binomial distribution, that is to say, $Y \sim b(n,p)$, where, $n = N_p \cdot (1-\rho) - |S|$, $p = P_m$. Then, $P(B) = 1 - P(Y=0) = 1-(1-P_m)^{N_p\cdot(1-\rho)-|S|}$. $P_n = P(A)P(B) = (1-P_m)^{|S|} \cdot (1-(1-P_m)^{N_p\cdot(1-\rho)-|S|})$. □

Theorem 6. *For any given nonself antigen ag, the probability of this antigen identified correctly is* $P_r = 1-(1-P_m)^{(|M|+|T|)(1-P_n)} \approx 1-e^{-P_m(|M|+|T|)(1-P_n)}$.

Proof. Set that A is event "ag matches some memory detector or some mature detector which is triggered by danger signals." $P_r = P(A)$. In the event A, the number of times X that antigens match detectors meets the binomial distribution, $X \sim b(n,p)$, where, $n = (|M|+|T|)(1-P_n)$, $p = P_m$. The memory detector and the mature detector which recognize selves cannot identify nonselves, which is not counting. Then, $P_r = P(A) = 1-P(X=0) = 1-(1-P_m)^{(|M|+|T|)(1-P_n)}$. According to Poisson theorem, when P_m is small and $(|M|+|T|)(1-P_n)$ is large, $P_r \approx 1-e^{-P_m(|M|+|T|)(1-P_n)}$. □

Theorem 7. *For any given nonself antigen ag, the probability of false negative with this antigen is* $P_{neg} = (1-P_m)^{(|M|+|T|)(1-P_n)} \approx e^{-P_m(|M|+|T|)(1-P_n)}$; *for any given self antigen ag, the probability of false positive with this antigen is* $P_{pos} = 1-(1-P_m)^{(|M|+|T|)P_n} \approx 1-e^{-P_m(|M|+|T|)P_n}$.

Proof. By Theorem 6, $P_{neg} = 1-P_r = (1-P_m)^{(|M|+|T|)(1-P_n)} \approx e^{-P_m(|M|+|T|)(1-P_n)}$. Set that A is event "the given self matches memory detector or mature detector." Then, $P_{pos} = P(A)$. In event A, the number of times X that selves match detectors meets the binomial distribution, $X \sim b(n,p)$, where $n = (|M|+|T|)P_n$, $p = P_m$. So, $P_{pos} = P(A) = 1-P(X=0) = (1-P_m)^{(|M|+|T|)P_n}$. According to Poisson theorem, when P_m is small and $(|M|+|T|)P_n$ is large, $P_{pos} \approx 1-e^{-P_m(|M|+|T|)P_n}$. □

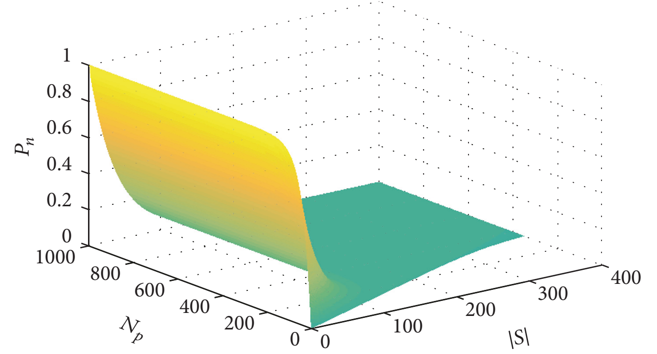


Figure 4: Effect of N_p and $|S|$ on P_n, $P_m = 0.025625$, $\rho = 0.01$.

Theorem 8. *Selves of the model are completely described at the macrolevel. The spatial complexity of the dynamic tolerance model producing a fixed number of mature detectors is constant, and the time complexity is linear with the number of detectors (excluding immature detectors).*

Proof. According to (8), the self set evolves with a fixed length of time slice. With the passage of time, $\bigcup_{t=0}^{\infty} S(t)$ will cover the entire self space, which is to say, description of selves at the macrolevel is complete. Moreover, the size of the self set is limited to $size_{max}$. Without loss of generality, considering the extreme case, the number of selves is $|S(t)| = size_{max}$. D'haeseleer et al. [16] pointed out that, for an arbitrary matching rule, the spatial complexity of producing a fixed number of mature detectors is $O(l \cdot size_{max})$, and the time complexity is $O(((-\ln(P_{neg}))/(P_m \cdot (1-P_m)^{size_{max}})) \cdot size_{max})$. For a specific matching algorithm, P_m is constant. By Theorem 7, $P_{neg} \approx e^{-P_m(|M|+|T|)(1-P_n)}$. By Theorem 5, $P_n = (1-P_m)^{size_{max}} \cdot (1-(1-P_m)^{N_p\cdot(1-\rho)-size_{max}})$. So, the time complexity of producing a fixed number of mature detectors is $O(((-\ln(P_{neg}))/(P_m \cdot (1-P_m)^{size_{max}})) \cdot size_{max}) = O(((|M|+|T|)(1-P_n)/(1-P_m)^{size_{max}}) \cdot size_{max}) = O((|M|+|T|)(((1-P_n)\cdot size_{max})/(1-P_m)^{size_{max}}))$. That is to say, the time complexity of producing a fixed number of mature detectors is linear with the number of memory detectors and mature detectors. □

For a specific matching rule, P_m is constant [17]. For r-continuous bit matching method, $P_m = 0.025625$. Figures 4 and 5 are the Matlab simulations of Theorem 5. As can be seen from the figures, when $|S|$ is large enough, effect of N_p and ρ on P_n is small. When $|S| = 200$, $N_p = 500$, $\rho = 0.01$, $P_n < 1\%$ reaches the ideal value.

Figure 6 is the Matlab simulation of Theorem 6. As can be seen from the figure, when $|M|$ and $|T|$ become large, P_r increases.

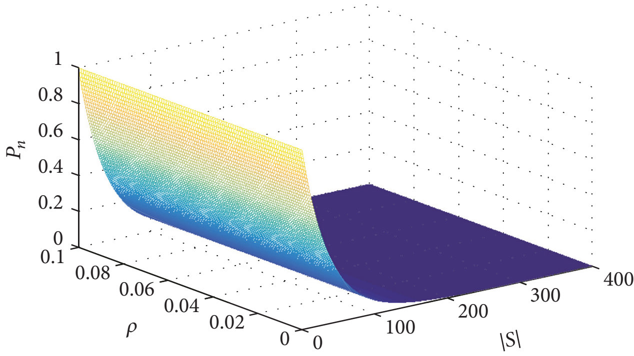


Figure 5: Effect of ρ and $|S|$ on P_n, $P_m = 0.025625$, $N_p = 400$.

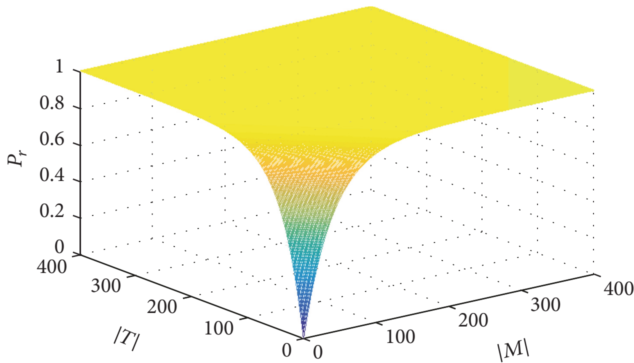


Figure 6: Effect of $|M|$ and $|T|$ on P_r, $P_m = 0.025625$, $P_n = 0.01$.

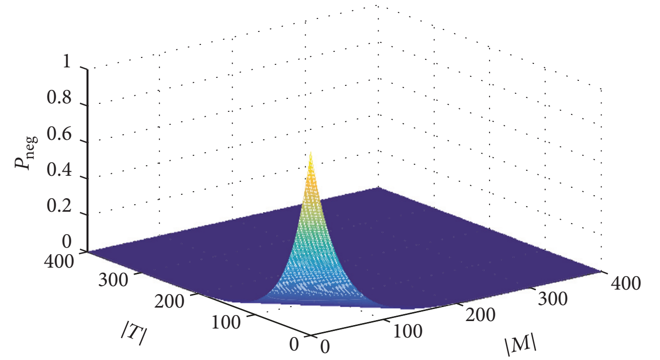


Figure 7: Effect of $|M|$ and $|T|$ on P_{neg}, $P_m = 0.025625$, $P_n = 0.01$.

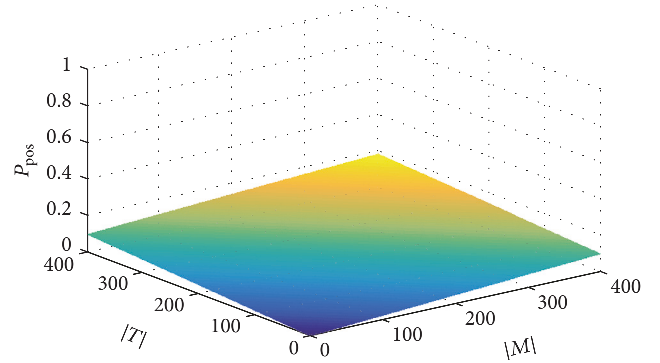


Figure 8: Effect of $|M|$ and $|T|$ on P_{pos}, $P_m = 0.025625$, $P_n = 0.01$.

Figures 7 and 8 are the Matlab simulations of Theorem 7. As can be seen from the figures, with the rise of $|M|$ and $|T|$, P_{neg} decreases and P_{pos} increases.

Considering simulations of Theorems 5, 6, and 7, when $|S| = 200$, $N_p = 500$, $\rho = 0.01$, $|M| = 100$ and $|T| = 100$, $P_n < 1\%$, $P_r > 95\%$, $P_{neg} < 1\%$, $P_{pos} < 5\%$ reach ideal values.

4. Experimental Results and Analysis

In this section, we verified the validity of IB-IDS through experiments, including security analysis, effects on the performance of programs after joining IB-IDS into the Xen virtual machine system, and intrusion detection efficiencies of IB-IDS. Experimental environment is as follows. All tests were performed on the ThinkPad T540p notebook. This type of hardware configuration is an Intel Core i5-4300M 2.60 GHz quad-core CPU and 8 G of physical memory. Xen version number is 4.4.1, which manages two domains, privileged VM dom0 and guest VM dom1. These two virtual machines run Ubuntu system with the version 14.04, and the kernel version of Linux is 3.13.0.19. Dom0 is allocated four VCPU and 4 G physical memory, and CPU scheduling weight is set to 256, while Dom1 is allocated four VCPU and 1 G physical memory, and CPU scheduling weight is set to 256.

In IB-IDS, parameters are set as follows. Danger signal parameters $k_1 = 1$, $k_2 = 0.5$, $k_3 = -1.5$, and the radius of danger zone $r_danger = 0.5$. Experiments run 10 times and averaged results were acquired.

4.1. Security Analysis. In the architecture description of the model, each module is distributed in different virtual machines. In domU, data is collected and then passes to dom0 through interdomain communication mechanism. The authorization list of Xen can make sure that a domain's memory space can only be accessed by its authorized domain. In the model, domU is the owner of a ring sharing buffer, and dom0 has the only granted permission; other domain cannot access. Therefore, data will not be leaked to other unauthorized domain, and the data transfer process is safe.

In paravirtualized Xen, domU accesses the hardware indirectly through dom0. To ensure the safety of the immune calculation, the model passes data to dom0 for computation. In this model, we assume that the privileged virtual machine is a trusted node.

Some traditional intrusion detection tools typically need to be deployed in a client virtual machine. Because the client virtual machine is not a trusted node, and it is exposed to various attacks, so the detection tools are also vulnerable. In this model, we assume that the virtual machine monitor is also a trusted node. The memory space of the two modules which are deployed in domU will be monitored by the virtual machine monitor.

Therefore, the monitoring process and results of the model are reliable.

4.2. Performance Evaluations of the Model. The introduction of IB-IDS to a virtual machine system will obviously bring

Table 1: Illustrations of tested parallel programs.

Program names	Meanings	Parameter settings
FFT	Computing a fast Fourier transform	$m = 22$, $p = 2$, $n = 65536$, $l = 4$
LU	Splitting a sparse matrix into a product of a lower triangular matrix and an upper triangular matrix	$p = 2$, $n = 2048$, $b = 16$
Ocean	Simulating movements of an entire ocean through the edge of the ocean currents (noncontiguous block allocation method)	$p = 4$, $n = 258$, $t = 380$, $e = 1e - 09$
Raytrace	Path simulation of lights	$p = 4$, envfile = ball4
Barnes	Simulating a three-dimensional multibody system (e.g., galaxies)	$p = 2$, fleaves =2

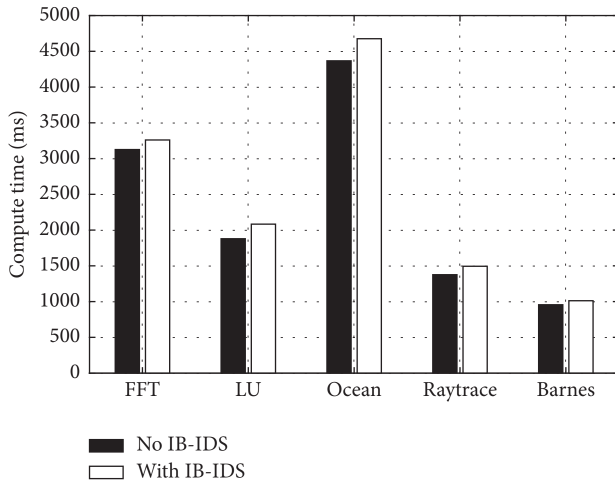


Figure 9: Testing of parallel programs.

some performance cost. In cloud computing, many applications are executed concurrently. Therefore, this section firstly uses the appropriate performance test to assess the impact of IB-IDS on parallel programs. In our tests, we used the classic SPLASH-2 program group [18, 19]. The programs are written in C, are composed of 12 benchmarks, and use PThread parallel mode. We randomly select five procedures for testing and Table 1 gives a brief introduction.

Figure 9 shows contrasts of the five benchmarks between loading IB-IDS and unloading IB-IDS. As can be seen from the figure, the calculation time of dom1 is longer than the original system, and the average increased time is 7.33%, up to 10.86% on LU program, which indicates that the additional cost of virtual machine system with integrated IB-IDS is very small and in the acceptable range. Applying IB-IDS to cloud computing platforms will not have significant impact on parallel applications.

In IB-IDS, the main performance overhead of domU is from antigen presenting module and signal acquisition module, as well as the operation of passing data to dom0 through intervirtual machine communication mechanism. These acts are performed regularly, and the cost is limited. For example, antigen presenting module is a proactive monitoring program on system call sequence and is not triggered by every system call. Signal acquisition module is the same. Through the event channel, domU puts antigens and environmental status into the ring buffer, and only if the ring buffer is empty, it will notify dom0, which will cause a context switch between domU and dom0. If there is data in the ring buffer, Dom0 would have been kept reading, and domU's notification is not required. So, the overhead of context switching is limited. In addition, implementations of immune response module, signal measurement module, and information monitoring module will increase performance overhead of dom0, and the impact on domU can be ignored.

Then, we test the impact of IB-IDS on computation intensive applications. In our tests, we used set of benchmark programs, SPEC (Standard Performance Evaluation Corporation) CPU2000 [20]. The programs include two parts. One is CINT2000 against integer computation intensive applications. The other is CFP2000 against float applications. We choose CINT2000 which has 12 applications. And we randomly select five procedures for testing and Table 2 gives a brief introduction.

Figure 10 shows contrasts of the five benchmarks when loading IB-IDS and unloading IB-IDS. As can be seen from the figure, the calculation time of dom1 is longer than the original system, and the average increased time is 9.12%, up to 11.48% on 254.gap program. Compared with parallel programs, the influence of IB-IDS on the virtual machine is larger, but it is still in the acceptable range. So, IB-IDS can be integrated in the computation intensive program scenario of cloud computing.

At last, we test the impact of IB-IDS on web server. In our tests, DomU runs the web server and is composed of apache http server and PHP. We use the httperf tool [21] to generate continuous network requests that can cause the server to be overloaded. Using autobench tool [22], we can run httperf for many times, increase the number of requests per second, and extract the output of httperf results. Figure 11 shows contrasts of server responses when loading IB-IDS and unloading IB-IDS. As can be seen, when the frequency of HTTP request increases, the response time of the server after the introduction of IB-IDS rises. When the HTTP request frequency is 100, the increased time is less than 0.5 s which is acceptable. Therefore, in the cloud computing platform with the deployment of a web server, IB-IDS system can also be applied.

Table 2: Illustrations of tested computation intensive programs.

Program names	Meanings
164.gzip	The compression and decompression operations of a set of files
175.vpr	According to specific algorithms, placement and routing operations for field-programmable gate array circuit
186.crafty	Chess programs, find the next move in view of the board layout
252.eon	Probability ray tracing used to create a 3d object image
254.gap	Solving the problem of correlation analysis and calculation of discrete mathematics

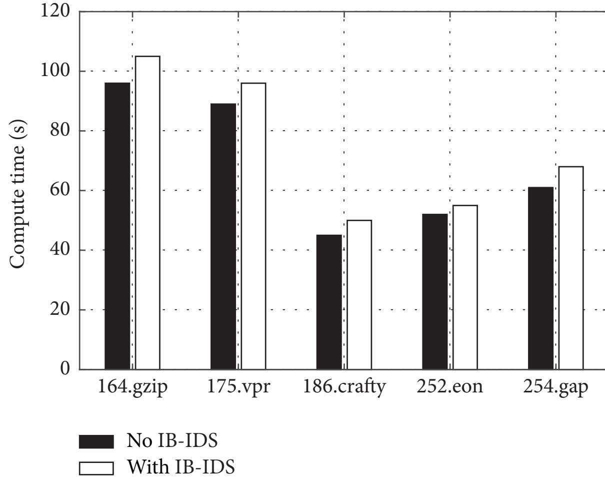


Figure 10: Testing of computation intensive programs.

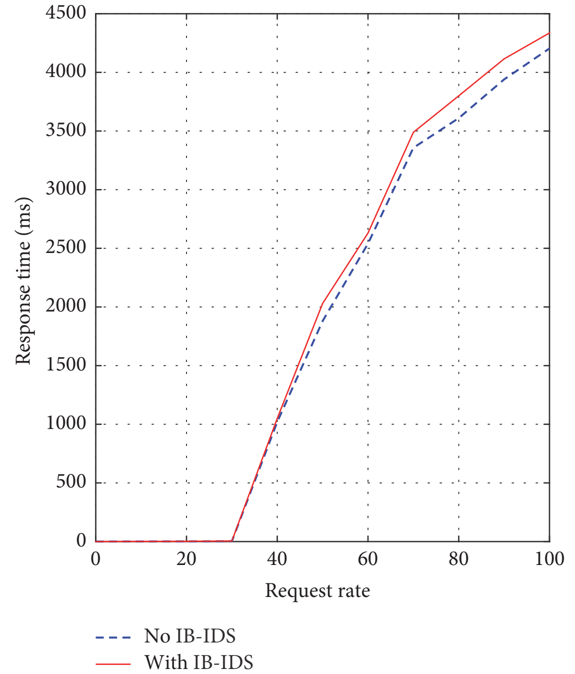


Figure 11: Testing of web server load.

4.3. Comparisons of Detection Rates and False Alarm Rates. This section will test the ability of IB-IDS for detecting attacks. Experiments adopt detection rate (DR) and false alarm rate (FAR) to measure the effectiveness of the system and to compare with ARTIS model proposed by Glickman et al. [17]. As a general computer immune system, the model has characteristics of diversity, distribution, dynamic learning, adaptability, and self-monitoring. It consists of a series of lymph nodes, and each node independently completes the immune function. Each node contains multiple detectors (a detector is a blend of the nature of B cells, T cells, and antibodies). ARTIS model draws on a variety of biological immune mechanisms, and coordinated stimulus and the dynamic evolution of detectors (immature ones, mature ones, and memory ones) make it continuously learning. The model has been successfully applied in intrusion detection, virus identification, pattern recognition, and so forth [17, 23]. Figure 12 shows the life cycle of detectors.

Figures 13 and 14 show comparisons of DR and FAR for IB-IDS and ARTIS in the simulation environment. In Figure 13, experiments adopt data with 60 nonselves in every 100 antigens, where 30 nonselves are just confirmed. This means that previously this type of antigen is considered to be self (normal procedure) and is now thought of as nonself (abnormal procedure). For example, unload some attack process instantly and stop providing related services. In Figure 14, experiments adopt data with 40 selves in every 100 antigens, where 20 nonselves are just defined. For example, load some new processes to provide new services. Experimental results show that IB-IDS has higher DR and lower FAR.

Then, we adopt wu-ftpd2.6.0 program, sendmail8.12.0 program, and some typical rootkit in Linux which are widely deployed as anomaly detection applications. Attacks against wu-ftpd are the scripting attack of file name matching vulnerability, the attack of getting around access restrictions, the scripting attack of site exec vulnerability, and so on. Attacks against sendmail are the sccp attack, decode attack, remote buffer overflow attack, and so on. Some of the representative rootkits include simple hook rootkit, inline hook rootkit, inline hook complex rootkit, and so on. Simple hook rootkit: a rootkit of this type modifies the system call function's entry address to a malicious function. When the corresponding system call is called, the malicious function is executed instead of the original system call function. Inline hook rootkit: a rootkit of this type does not modify the system call table entry address but will replace a few bytes of beginning system call function with a jump statement. Compared with the simple hook rootkit, the rootkit is more subtle. Inline hook complex rootkit: a rootkit of this type does not

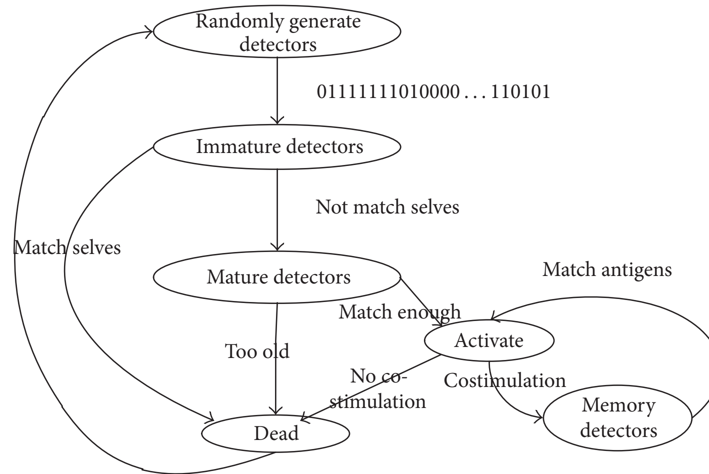


Figure 12: The life cycle of detectors in ARTIS.

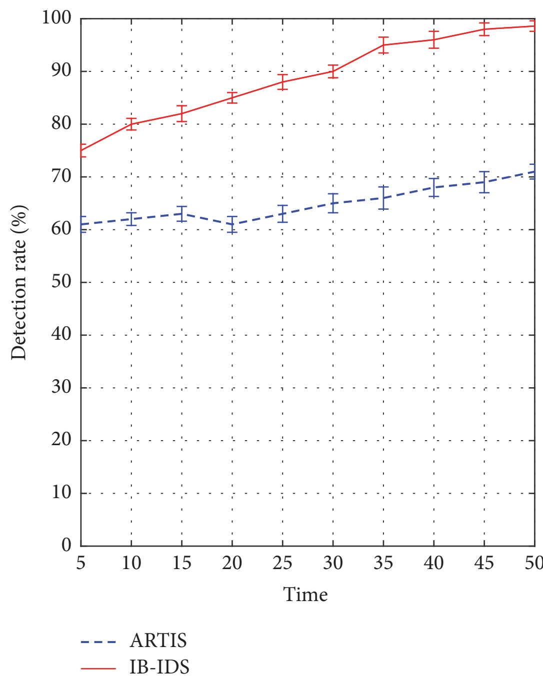


Figure 13: Comparisons of DR for IB-IDS and ARTIS.

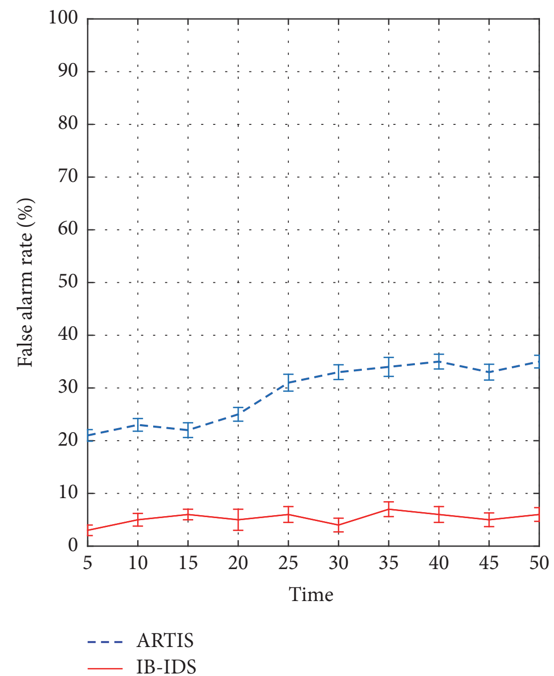


Figure 14: Comparisons of FAR for IB-IDS and ARTIS.

replace the first bytes of the system call function with jump statements, except the other few bytes, for example, bytes in the middle. Table 3 lists DRs and FARs of IB-IDS and ARTIS, and variances are in parentheses. As can be seen from the table, IB-IDS has high detection rates and low false alarm rates under various attacks and is feasible for judging applications in client virtual machines.

5. Conclusions

Cloud computing platforms are usually based on virtual machines as the underlying architecture; the security of virtual machine systems is the core of cloud computing security. Current study on security of user programs and vulnerabilities of virtual monitors cannot accurately judge the real state of the client application in the virtual machine. At the same time, the proposed defense methods are only for specific attacks and vulnerabilities and cannot effectively deal with threats under other attacks. This paper presents an immune-based intrusion detection model in virtual machines of the cloud computing environment, to ensure safety of user-level applications in client virtual machines. The model extracts system call sequences and their parameters of programs, abstracts them into antigens, and fuses environmental information of guest virtual machines into danger signals in client VMs. Then, immune responses will be performed

Table 3: Detection results.

Processes	ARTIS		IB-IDS	
	DR%	FAR%	DR%	FAR%
wu-ftpd				
file name matching vulnerability	76.12 (5.11)	10.28 (4.17)	96.55 (1.14)	7.22 (1.22)
site exec vulnerability	79.87 (2.45)	9.87 (5.32)	97.31 (1.23)	6.65 (2.01)
attack of getting around access restrictions	77.54 (4.77)	12.75 (3.74)	97.02 (1.08)	7.43 (1.67)
sendmail				
sccp attack	74.52 (3.56)	14.62 (3.41)	98.11 (1.25)	5.15 (1.63)
decode attack	81.21 (4.84)	15.72 (3.87)	98.35 (1.01)	5.42 (1.69)
remote buffer overflow attack	82.45 (5.46)	12.84 (5.63)	98.78 (1.14)	5.80 (1.28)
rootkit				
simple hook rootkit	85.15 (5.16)	9.41 (4.12)	99.99 (0)	0 (0)
inline hook rootkit	82.45 (6.82)	10.75 (8.20)	99.99 (0)	0 (0)
inline hook complex rootkit	75.14 (5.23)	9.56 (6.77)	95.84 (2.42)	3.78 (2.89)

in the privileged VM. During the detection process, information monitoring mechanism will be executed in VMM. Experimental results show that the model brings a small performance overhead for the virtual machine system and has a good detection performance. It is applicable to judge the state of user-level application in guest virtual machine, and it is feasible to use it to increase the user-level security in software services of cloud computing platform.

Conflicts of Interest

The authors declare that there are no conflicts of interest.

Acknowledgments

The authors would like to acknowledge Sichuan Agricultural University Double Support Project for providing financial aid.

References

[1] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable Virtual Machines," in *Proceedings of the In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[2] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP*, pp. 233–247, Oakland, Calif, USA, May 2008.

[3] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM monitoring using hardware virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS'09*, pp. 477–487, Chicago, Illi, USA, November 2009.

[4] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS'09*, pp. 545–554, Chicago, Ill, USA, November 2009.

[5] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011*, pp. 279–290, Newport Beach, Calif, USA, March 2011.

[6] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 670–684, 2011.

[7] S. Bharadwaja, W. Sun, M. Niamat, and F. Shen, "Collabra: A xen hypervisor based collaborative intrusion detection system," in *Proceedings of the 2011 8th International Conference on Information Technology: New Generations, ITNG 2011*, pp. 695–700, Las Vegas, NV, USA, April 2011.

[8] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti, "Operating system interface obfuscation and the revealing of hidden operations," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6739, pp. 214–233, 2011.

[9] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS'11*, pp. 401–412, Chicago, Ill, USA, October 2011.

[10] H. Benzina and J. Goubault-Larrecq, "Some Ideas on Virtualized System Security, and Monitors," in *Data Privacy Management and Autonomous Spontaneous Security*, vol. 6514 of *Lecture Notes in Computer Science*, pp. 244–258, Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2011.

[11] L. Wang, H. Gao, W. Liu, and Y. Peng, "Detecting and managing hidden process via hypervisor," *Jisuanji Yanjiu yu Fazhan/Computer Research and Development*, vol. 48, no. 8, pp. 1534–1541, 2011.

[12] P. Barham, B. Dragovic, K. Fraser et al., "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 164–177, New York, NY, USA, October 2003.

[13] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*, Prentice Hall Press Upper Saddle River, NJ, USA, 2007.

[14] S. Forrest, A. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 202–212, Oakland, Calif, USA.

[15] L. I. De-Yi, C. Y. Liu, D. U. Yi, and X. Han, "Artificial intelligence with uncertainty," *Journal of Software*, vol. 15, no. 11, article 2, 2004.

[16] P. D'haeseleer, S. Forrest, and P. Helman, "An immunological approach to change detection: algorithms, analysis and implications," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 110–119, Oakland, Calif, USA.

[17] M. Glickman, J. Balthrop, and S. Forrest, "A machine learning evaluation of an artificial immune system," *Evolutionary Computation*, vol. 13, no. 2, pp. 179–212, 2005.

[18] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24–36, Santa Margherita Ligure, Italy.

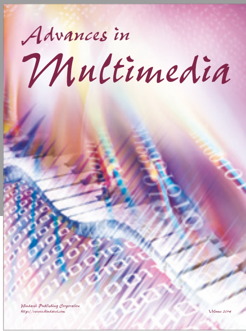
[19] J. P. Singh, W. Weber, and A. Gupta, "SPLASH," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 1, pp. 5–44, 1992.

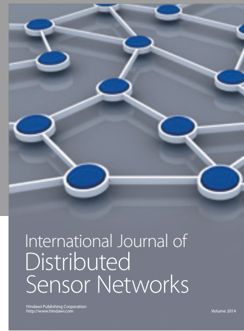
[20] Standard Performance Evaluation Corporation. http://www.spec.org/.

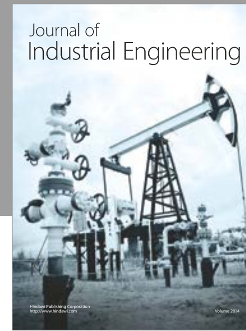
[21] httperf. http://www.hpl.hp.com/research/linux/httperf/.

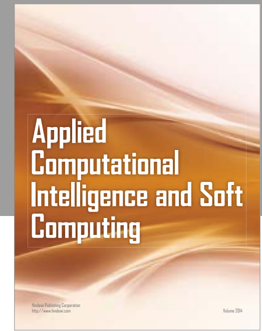
[22] autobench. http://www.xenoclast.org/autobench/.

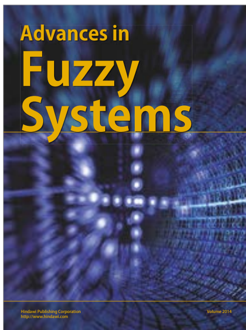
[23] J. Balthrop, S. Forrest, M. E. J. Newman, and M. M. Williamson, "Technological networks and the spread of computer viruses," *Computer Science*, vol. 304, no. 5670, pp. 527–529, 2004.

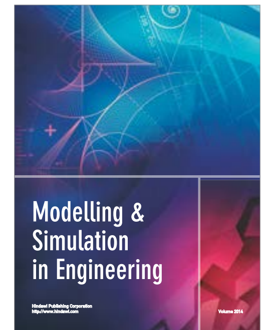












Hindawi

Submit your manuscripts at
https://www.hindawi.com