Computing and Informatics, Vol. 31, 2012, 271–297

AUTOMATED APPROACH TO INTRUSION DETECTION IN VM-BASED DYNAMIC EXECUTION ENVIRONMENT

Feng Zhao, Hai Jin

School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074 China
e-mail: `zhaof@hust.edu.cn`

Communicated by N. Jaisankar

Abstract. Because virtual computing platforms are dynamically changing, it is difficult to build high-quality intrusion detection system. In this paper, we present an automated approach to intrusions detection in order to maintain sufficient performance and reduce dependence on execution environment. We discuss a hidden Markov model strategy for abnormality detection using frequent system call sequences, letting us identify attacks and intrusions automatically and efficiently. We also propose an automated mining algorithm, named *AGAS*, to generate frequent system call sequences. In our approach, the detection performance is adaptively tuned according to the execution state every period. To improve performance, the period value is also under self-adjustment.

Keywords: Intrusion detection, virtual machine, hidden Markov model (HMM), sequential data mining, dynamic graph

1 INTRODUCTION

As computer systems play increasingly vital roles in modern society, network security has become more and more remarkable. Intrusion detection system (IDS) is one of the most critical technologies to help protect these systems. With respect to the origin of analyzed data, there are two main approaches for intrusion detection: Network-based IDS (NIDS) based on watching the network traffic flowing through the systems to monitor, and Host-based IDS (HIDS) based on watching local activity on a host, like processes, network connections, system calls, logs, etc. Comparing these two dominant IDS architectures, NIDS offers high attack resistance at the cost of visibility, and HIDS offers high visibility but sacrifices attack resistance. To avoid the disadvantages of the two types of IDS, many researchers have proposed and implemented virtual machine (VM) technology to build intrusion detection systems which provide good visibility into the state of the monitored host, while still providing strong isolation for the IDS.

While virtual machines provide significant flexibility for users and administrators to create, destroy, migrate and modify guest machines with unprecedented ease, it forces IDS to work in a dynamic computing environment, which gives rise to radically different solutions than are found in traditional computing environments [1]. This can present some unique challenges to detect intrusion. First, the semantic gap between guest OS and underlying virtual machine monitor (VMM) brings forth handicap to detect intrusion in high level; second, the virtual machines that run various services could potentially hurt system security of computing environment themselves; finally, traditional IDS relies on the execution environment, in particular, on the operation system running. In VM-based virtual computing platform, the execution environment is dynamically changing, and it is impossible for IDS to detect malicious attacks with sufficient performance.

In VM-based virtual computing environments, VMM is a layer of software running on the hardware platform and VM runs as a user process on the host. Both the guest OS and guest applications are inside this single host process. Thus, it is obvious that privileged processes should be a good level to focus on because the exploitation of vulnerabilities in privileged processes can give confident evidence to indentify intruders in virtual computing environments. A natural observable view on processes in VM-based virtual computing environment would be based on system calls, because guest OS processes access system resources and peripheral device through the use of system calls. To narrow the semantic gap and reduce the dependence on execution environment, in particular on the operation system running, various data mining and machine learning techniques have been used in intrusion detection to discover abnormalities in the behavior of privileged processes. Such research projects include Mining Audit Data for Automated Models for Intrusion Detection (MAMAD ID) [2], Intrusion Detection System using System Calls [3], Minnesota Intrusion Detection System (MINDS) [4], HMM-based Intrusion Detection System [5] etc.

However, the performance of data mining-based intrusion detection greatly relies on the quality of training data and the detection models. As virtual computing environment keeps on changing, it is too difficult to collect training data covering all kinds of security vulnerabilities and attacks in real time. Moreover, system parameters of data mining-based IDS should be adjusted continually in order to maintain the performance since fixed detection model is not suitable for the dynamic environment. Zhenwei Yu et al design an Automatically Tuning Intrusion Detection System (ATIDS) and point out that tuning intrusion detection model on-the-fly with the verified data can achieve performance improvement [6], but the tuning procedure is a complex process which will radically aggravate the burden on the system operators.

Our work is inspired by intrusion tolerance (IT), an immune approach in system security emerged and gained impressive momentum recently, which is instead of trying to prevent every single intrusion but tolerated: the system has the means to prevent the intrusion from generating a system failure. Our work is also concerned with IT to construct security VM-based dynamic computing environment. In this paper, we present an automated approach to intrusions detection in order to maintain sufficient performance and reduce dependence on execution environment. We discuss a hidden Markov model strategy for abnormality detection using frequent system call sequences, letting us identify attacks and intrusions automatically and efficiently. We also propose an automated mining algorithm, named *AGAS*, to generate frequent system call sequences. In our approach, the detection performance is adaptively tuned according to the execution state every period. To improve performance, the period value is also under self-adjustment.

This article is structured as follows: Section 2 surveys related work; Section 3 introduces how to automatically discover abnormalities in system call sequences; Section 4 describes the dynamic techniques of intrusion detection in VM-based virtual computing environments, Section 5 presents the implementation results, and Section 6 details the proposal.

2 RELATED WORK

With the development of virtualization technology, virtual machines are widely used to improve the security of a computing system against attacks to its services. X. X. Jiang et al. present a new approach to apply intrusion detection techniques to virtual machine based systems, and keep the intrusion detection system out of reach from intruders [7]. Tal Garfinkel at al. have implemented a HIDS prototype in virtualization environment, named *Livewire*, which can get the state information of other VMs from VMM and interpret the hardware-level information with OS semantic by using the OS interface library [8]. *Revirt* and *Subvirt* propose an intermediate layer between the monitor and the host system to analyze intrusion actions by capturing the data through the syslog process of the virtual machine and sending them back to the host system for recording and later analysis [9, 10]. The virtualization overhead of *Revirt* is up to 58 % for kernel-build, the time overhead of *Revirt* is at most 8 %; but its vulnerability is no-deterministic because it depends on a time-of-check to time-of-use race condition. *Lares* addresses the problem of secure active monitoring in virtualized systems and presents a hybrid model that gives security tools the ability to do active monitoring while still benefits from the increased security of an isolated virtual machine [11]. M. Sharif et al. propose a general-purpose framework, secure In-VM Monitoring (SIM), to enable security monitoring applications to be placed back in the untrusted guest VM for efficiency without sacrificing the security guarantees provided by running them outside of the VM [12]. What's more, VMM also becomes a popular platform for building Honeypot and Honeyfarm to capture and detect intrusion [13, 14, 15]. For example, [13] introduces *Collapsar*, which has a virtualization-incurred overhead of about 50 % and performance overhead incurred by the traffic redirection and dispatching mechanisms of about 275 % in TCP throughput.

In contrast to pursuing the nearly impossibility of a perfect barrier, many researchers are working on intrusion tolerance instead of intrusion prevention [16, 17]. For example, Matthews et al. utilized data protection and recovery to tolerant intrusion, they also presented an architecture in which personal data is protected in a file server virtual machine and in which trusted checkpoints of virtual machine appliances house system data and enable rapid recovery from attack [18]. In [19], a novel intrusion architecture which applies VM-based and OOB network to support reliable control even though the primary network is under severe attack is described and implemented. *XenFIT* and *XenRIM* are designed to monitor fire alarms on intrusion in real-time manner in *Xen*-based virtual computing environment, which are not required to create and update the database like in the legacy methods [20, 21]. In [22], authors present an architecture for intrusion tolerance using virtual machines that benefits from a shared memory to simplify the consensus protocol, named SMIT (Shared Memory based Intrusion Tolerance), which demonstrates that the safe component only needs to provide a simple shared memory abstraction to reach the computational reduction.

To improve and balance detection rate and false rate, Sabhnani et al. build an intrusion system using pattern recognizing and machine learning algorithms on KDDCup99 dataset [23], but their system is based on the performance of different subclassifier on the dataset. Wenke Lee et al. propose a data mining method as high-efficient intrusion detection which can detect new attacks as soon as possible [2, 24], but their system is tightly related with the high-quality training dataset. Zhenwei Yu et al. present an automatically tuning intrusion detection system [6]; their system achieves about 30 % performance but has about 10 % false predictions. In [25], a control-theoretic HMM model for intrusion detection using distributed multiple nodes is presented, which reduces the frequency of false positives, but this model lacks automated response and has high computation complexity. Jiankun Hu et al. also propose a simple data preprocessing approach to speed up a HMM training for system-call-based anomaly intrusion detection, which reduces training time by up to 50 % with unnoticeable intrusion detection performance degradation, compared to a conventional batch HMM training scheme [26].

3 DISCOVERING ABNORMITY IN SYSTEM CALL SEQUENCES

3.1 Overview

The technique of virtual memory introspection was introduced by Bryan D. Payne et al. They design the *XenAccess* architecture and present the *XenAccess* monitoring library to provide virtual memory introspection and virtual disk monitoring capabilities based on six high-level requirements [27]. Monitoring virtual memory with *XenAccess* requires no changes to the VMM, VM and OS. Using introspection, *XenAccess* can view the memory of another VM access with the target to infer OS data at an abstract level.

Attacks, especially those that attempt to compromise a computer system using the system call interface, are an increasingly important threat to virtual computing environment. Using virtual memory introspection provided by *XenAccess*, monitoring system call at the abstract level becomes feasible and more convenient, which can detect and control guest applications by checking them at runtime. Monitoring system calls of guest VM and specifying the program's normal behavior is an effective approach for stopping a large class of malicious attacks [28]. Essentially, it is helpful to convert a potentially successful attack into a fail-stop failure of the compromised process.

3.2 HMM for Profiling System Calls

The Hidden Markov Model (HMM) is a powerful statistical tool for modeling generative sequences that can be characterized by an underlying process generating an observable sequence [29]. Because system call track is a group of time-varying discrete time sequence data, we use the HMM to describe the statistical rules among local system calls in the process of system operation. In early studies some researchers reported preliminary evidence that short sequences of system calls can be used to discriminate several types of intrusions. The method presented here prolongs those studies to long sequences of system call in order to improve the accuracy of intrusion detection.

HMM is a special type of Bayesian Network. The formal definition of a HMM is as follows:

$$\lambda = (S, V, A, B, \pi., \tag{1}$$

where S is the state set, and V is the observation set. Suppose n is the total number of states, and m is the maximum number of observed sequence:

$$S = (s_1, s_2, \ldots, s_n., \tag{2}$$

$$V = (v_1, v_2, \ldots, v_m). \tag{3}$$

A is the state transition probability matrix, storing the probability of state j following state i.

$$A = [a_{ij}]_{n\times n}, a_{ij} = p(\text{step } t \text{ at } s_j|\text{step } (t-1) \text{ at } s_i). \tag{4}$$

B is the observation probability array, storing the probability of observation k from state i.

$$B = \{b_i(k)\}, b_i(k) = p(v_k|s_i). \tag{5}$$

π is the initial probability array, storing the probability of state i at first step.

$$\pi = \{\pi_i\}, \pi_i = p(s_i \text{ at initial step}). \tag{6}$$

For HMM model $\lambda = (S, V, A, B, \pi)$, the system call sequences are compared to the observation V, the observed sequences will be either normal or attack. Our way is to profile system calls by means of establishment of HMM model for them. Comparing to traditional HMM model, the modification in HMM of our profiling is as follows: Firstly, the state space is limited in 3 states in order to improve performance; secondly, the length of observation sequence is changeable instead of fixed length in traditional HMM; thirdly, computation on π_i is predigested for efficiency.

1. State space, $S = (Normal, \text{Attack}, \text{Intrusion})$ or $S = (0, 1, 2)$, represents that the system call sequence has the following three states:

   - *Normal* (N) state indicates a legal activity.
   - *Attack* (A) state indicates an attack activity that is setting itself up. An attack includes attempts to get privileged resources, enhance system vulnerabilities, accelerate memory usage, and gain remote login trust relation and so on. *Intrusion* (I) state indicates an attack has achieved his goal and has become a successful intrusion. A successful intrusion is always accompanied by unusual resource usage, service failure and data leak etc.

2. V is the observed system call sequence, such as (*open*, *read*, *mmap*, *mmap*, *open*, *read*, *mmap*). The maximum number of observations m is dependent on the number of system calls, for example, in *Xen3.1* there are 325 system calls, that is $m = 325$.

   Even if the raw system call sequence is widely used as evidence to detect intrusion, there are two inherent vulnerabilities for high-required intrusion detection. At first, numerous sequences observed are not distinct enough, which has great influence on the efficiency of intrusion detection; Secondly, it is difficult to search for suitable length of system call sequence for intrusion detection. Short sequence is always selected, but it is at the cost of detection rate. In order to improve the accuracy and efficiency of intrusion detection, here we only take into account the maximum frequent sequences of system call as V.

3. Given that n=3, matrix A is defined as: $A = [a_{i,j}]_{3\times 3} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix}$.

   Here $a_{i,j}$is the transition probability of maximum frequent patterns amid different states.

4. $\pi = \{\pi_0, \pi_1, \pi_2\}$ is the initial probability array for every state, and $\sum_0^2 \pi_i = 1$. For raw system call sequences, under ideal circumstance the system calls are all legal, that is $\pi = \{1, 0, 0\}$; but when V is matched with frequent sequences of

system call, the initial probability depends on the training dataset. The π_i is computed as $\pi_i = p(i) = \frac{\text{initial number of patterns in state } i}{\text{total number of patterns}}(1 \leq i \leq 2)$, and $\pi_0 = 1 - \sum_1^2 \pi_i$.

5. $B = \{b_i(j)\}$, which is the probability distribution of frequent sequential patterns in state i.

3.3 Computing $p(O|\lambda)$

An HMM-based approach correlates the system call observations and state transitions to predict the most probable intrusion state sequence, which can be viewed as that evaluating how well a HMM model predicts a given observation sequence. The probability of the observation O for a specific state sequence Q is $p(O|\lambda)$. In this paper, we calculate $p(O|\lambda)$ and analyse it to find out abnormity from activity sequences. That is, after establishing accurate HMMs on normal behavior exactly, if probability of observed behavior is smaller than a given threshold probability, we should believe that this activity is a suspicious behavior.

An efficient computational algorithm called *forward algorithm* is reported to compute $p(O|\lambda)$, which has a relatively lower computational complexity. The *forward algorithm* has three principal equations which are the initialization of what is called forward variable α, an induction step, and the termination step. The forward variable is defined as $\alpha_t(i) = p(o_1, o_2, \ldots, o_t, q_t = s_i|\lambda)$, and α is the probability of the partial observation sequence $o_1, o_2, \ldots, o_t(1 \leq t \leq T)$and state $s_i$at time t.

The algorithm for this process is called the *forward algorithm* and is as follows [30]:

1. Initialization:
$$\alpha_1(i) = \pi_i b_i(o_1., 1 \leq i \leq n, \tag{7}$$

2. Induction:
$$\alpha_{t+1}(j) = [\sum_{i=1}^{n} \alpha_t(i) a_{ij}] b_j(o_{t+1}., 1 \leq t \leq T-1, 1 \leq j \leq n, \tag{8}$$

3. Termination:
$$p(O|\lambda. = \sum_{i=1}^{n} \alpha_T(i). \tag{9}$$

According to the state space $S = (\text{Normal}, \text{Attack}, \text{Intrusion})$ and the *forward algorithm* defined above, the trellis of $p(O|\lambda)$ computing process for our HMM is as shown in Figure 1. It is apparent that by caching α values the *forward algorithm* reduces computational complexity involved to $O(n^2T)$. In our Hidden Markov Model, $n = 3$, thus computing $p(O|\lambda)$ has a linearity complexity of calculations involved to $O(T)$.

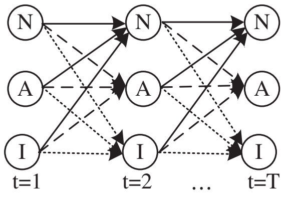

Fig. 1. Computing process for our HMM

3.4 Mining Frequent Sequences of System Calls

In normal circumstances with the increase of length of the observation sequences, the emergence probability of observation sequences, $p(O|\lambda)$, is getting smaller and smaller. So it is very difficult to determine whether or not an observation sequence is a normal sequence only by measuring the probability of $p(O|\lambda)$. That is, the decision process is greatly related to the length of observation sequences. To reduce this dependence, we introduce sequence data mining technique as an assistant artifice to discover abnormality from normal activities. Here we present an improved mining algorithm, named $AGAS$ (algorithm of Automatically Generating Attack Sequence), to find the attack frequent sequences of system call. Rather than focusing on building a highly effective initial detection model, $AGAS$ is proposed to describe a tuning mechanism when it is exposed to the dynamically changing environment.

Suppose O is the whole track of an inner process or a running program in VM-based virtual computing systems; this sequence of system calls observed can be described as $O = \langle sc_1, sc_2, \ldots, sc_i, \ldots \rangle$, which conforms to the given order constraints, and encodes an interesting fact that system call sc_i occurs after sc_{i-1} and before sc_{i+1}. Any subsequence compressed, $SC = \langle sc_1, sc_2, \ldots, sc_L \rangle$ and L is the length of SC, is the candidate sequence pattern which can be utilized to discover abnormality. If a subsequence is superior to a certain requirement, it can be specified as a rule to discriminate suspicious act from normal activity; this kind of sequence is consequently called frequent sequence.

In conventional way, frequent sequence is distinguished by a probability named support, that is, a sequence is frequent only if its appearance frequency is superior to the threshold probability. Here, a novel threshold probability to frequent sequence is defined as follows:

$$\overset{(L)}{\sup} = \frac{\text{length}(SC) \times p(SC)}{\text{length}(O)}, \tag{10}$$

and $p(SC) = \prod_{i=1}^{L} p(sc_i|sc_{i-1} \ldots sc_1.$, $p(sc_i)$ is the objective probability.

In order to find the potential attacks and the relationships among sequences of system calls, we associate graph with frequent sequences to depict the ways in which an attack can force the computing environment into an unsafe state. The formal definition of system call sequence graph is as follows:

$$G = (V, E, P). \tag{11}$$

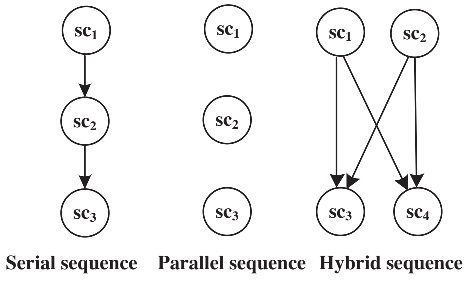

Fig. 2. Graph of sequences of system calls

V is the nodes set of graph representing the observation set of system calls, $E = \langle sc_i, sc_j \rangle$ is the edges set of graph. Every system call sc_i is a node in the graph, each edge $\langle sc_i, sc_j \rangle$ describes the relationship between two nodes, and an execution is defined as a finite sequence of system calls $\langle sc_1, sc_2, \ldots, sc_L \rangle$. P is the set of probabilities, transition probability p_{ij} denotes the probability of transition from system call s_j to s_i which is also defined as the reciprocal of the number of successors of the system call s_j. There are three types of system call sequences including serial sequence, parallel sequence and hybrid sequence which can be structured into $G = (V, E, P)$ as shown in Figure 2.

According to the graphic definition, *AGAS* algorithm is specified as follows:

Input:

- $V = \langle sc_1, sc_2, \ldots, sc_i, \ldots sc_m \rangle$ – Set of system calls
- $P \subseteq V \times V \to [0, 1]$ – Transition matrix of system calls
- $V_0 \subseteq V$ – Set of initial system calls from *XenAccess*

Output: Attack execution $\langle sc_1, sc_2, \ldots, sc_L \rangle$

Algorithm AGAS: Step 1: Construct initial vector

Let $v = (v_1, v_2, \ldots, v_i, \ldots, v_m)$ be an m-dimensional vector which represents the initial state of observation sequence of system call.

$$v_i (1 \leq i \leq m) = \begin{cases} 1 & v_i \in V_0 \\ 0 & v_i \notin V_0 \end{cases} . \tag{12}$$

Step 2: Compute reachable probability

Vaibhav Mehta et al. give a method to rank states in an attack graph. They define a transition probability from one state to another state [31]. We use that in our algorithm. Let $r^{(m)}$ be an m-dimensional vector which represents the reachable probabilities for all system calls in a random simulation run of length up to m.

$$r^{(m)} = \sum_{k=0}^{m} P^k v \tag{13}$$

The reachable probability is computed as follows:

```
for (k = 1; k ≤ m; k++)
{
compute $r^{(k)}$ according to formula (12)(13);
compute $\sup^{(k)}$ according to formula (10);
if ($r^{(k)} \geq \sup^{(k)}$ and $r^{(k+1)} < \sup^{(k+1)}$)
return (k);
}
```

Step 3: Output attack sequence

$$SC = \emptyset$$

```
for (l = 1; l ≤ k; l++)
{
$SC = SC \cup sc_l$;
}
output (SC);
```

There are numerous reasons for our preference for *AGAS* algorithm, and I shall explore only a few of the most important ones here. One chief reason is that this algorithm is extremely simple and fast which greatly reduces the computing complexity. This algorithm is also very suitable to generate frequent sequence in virtual machine with a storage complexity of $O(n^2)$, since there is not enough space to store temp candidate sequence. Next, traditional mining algorithm using multiple scans is not appropriate because system call sequence is a one-time operation. Besides, bottlenecks of traditional candidate-based algorithms get great influence on efficiency. But attack recognition is a real time processing, traditional algorithms are strongly advised to be discarded because of latency. All in all, taking into account the metrics mentioned above, we chose this probability computing-based algorithm.

3.5 Intrusion Detecting Algorithm

Paulo Veríssimo et al. represent a well-defined relationship between attack, vulnerability, and intrusion which is called *AVI* composite fault model [32]. This model represents a fundamental sequence: *attack →vulnerability→intrusion→failure*. The *AVI* sequence can recursively occur in a coherent chain of events generated by the intruders. In *AVI* mode, vulnerabilities are defined as the primordial faults existing inside the components, essentially requirements, specification, design or configuration faults, and attacks are defined as interaction faults that maliciously attempt to activate one or more of those vulnerabilities. The event of a successful attack activating a specified vulnerability is called an intrusion.

According to *AVI* model, the state space $S = (\text{Normal}, \text{Attack}, \ldots, Intrusion)$ is proposed as mentioned above. Thus, we can discover abnormalities from normal activities that these abnormalities are also considered as attacks, and then distinguish intrusions from abnormalities. We briefly describe the method for discovering attacks and intrusions. The first step is to determine the set of frequent sequences $SC$and their maximal lengths k that are from the observed system call sequences. Next, the *forward algorithm* computes the emergence probability that SC is input as observation set. In this step, the *forward algorithm* would circularly compute $p(O|\lambda)$ until the length of observation sequence is up to k. The algorithm of detecting attacks and intrusions is given as follows which reports "attack" or "intrusion" for a given system call sequence.

Input:

- $V = \langle sc_1, sc_2, \ldots, sc_i, \ldots, sc_m\rangle$ – Set of system calls
- $P \subseteq V \times V \to [0,1]$ – Transition matrix of system calls
- $V_0 \subseteq V$ – Set of initial system calls from *XenAccess*

Output: Running execution $\langle sc_1, sc_2, \ldots, sc_L\rangle$

Detection algorithm $\det ect\_A\_I()$**:**

```
$AGAS$ $(V, P, V_0)$; // Generate frequent sequence
attack($SC$); // flag $SC$ as attack sequence
$for$ $(l = 1;\ l \le length(SC) = k;\ l{+}{+})$
{
compute $p(O|\lambda)$ using $forward\ algorithm$;
if $(p(O|\lambda. \le \xi)$ // $\xi$ is the user-specified threshold
intrusion($SC$); // flag $SC$ as intrusion sequence
exit;
}
```

4 DYNAMIC ALGORITHM FOR INTRUSION DETECTION

In virtual computing environment, creating, migrating, updating and destroying a VM is as easy as processing a file. The fast and unpredictable changes occurring with VMs exacerbate intrusion detection tasks and significantly multiply the impact of malicious attacks. Rarely do traditional detection algorithms dynamically and automatically deal with these rapid changes. In order to ensure the security of dynamical computing environment, we design a novel detection algorithm based on backtracking to discover intrusions in virtual computing platform.

4.1 Tracing Changes in Virtual Machines

While the set of VMs is dynamically changing, intrusion detection engineer would like to know at any time which a VM has been altered or whether VM has been created etc. The changes occurring with VMM and VM running on it can be viewed as a dynamic graph that is undergoing a sequence of updates. Let the dynamical graph be described as $DG = (U', E')$ with $|U'| = n$ and $|E'| = e$. U' is a running unit in virtual computing environment, such as VMM, VMs or applications, etc.

E' representing the edges of graph is the set of relationship among running units. Especially, the vertices representing VMM are called *root* in that every running unit is directly or indirectly related to the VMM in virtual computing system.

In this paper, we take VM as a dependent unit. Consequently, creating VM is an operation that inserts vertices of the graph, destroying VM is an operation that deletes vertices of the graph, migrating VM is an operation that changes vertices and edges of the graph, updating VM is an operation that updates attributes associated with vertices or edges of the graph.

Henzinger et al. introduce a data structure called *ET trees* to work on dynamic forests whose vertices are associated with weighted or unweighted keys [33, 34]. They also design the first fully dynamic algorithms to combine graph decomposition with randomization which supports in $O(\log n)$ time to update and query. It provides three query functions:

- *Connected*(u_1, u_2): returns whether vertices u_1 and u_2 are in the same tree.
- $Size(u)$: returns the number of vertices in the tree that contains u.
- $Minkey(u)$: returns a key of minimum weight in the tree that contains u; if keys are unweighted, an arbitrary key is returned.

We adapt *ET trees* to construct our dynamic graph. An *Euler tour* of a tree is a maximal closed work over the graph obtained by replacing each edge by two directed edges with opposite direction. The work traverses each edge only once, that is, if the tree has n vertices, the *Euler tour* has length of $2(n-1)$. An *Euler tree* is a balanced dynamic binary tree over some *Euler tour* around the tree and leaves of the balanced binary tree are the nodes of the *Euler tour*.

Suppose $ET = (U, E)$ is the *Euler tree* architecture for VMM and VM running on it. U representing the nodes of tree is the unit set of VMM and guest VMs, each element of U is a running unit in virtual computing environment. E representing the edges of tree is the set of communication relationship between VMM and VM, or between VM and VM. There are two types of edges in ET: the first is the edge between VMM and guest VM, the other is the edge between guest VM and guest VM that there is an inherent relation between them in a specified malicious activity; for example, a VM may act as a puppet machine controlled by another VM to carry out *DDos* attack.

In order to minimize the amount of recomputation required by intrusion detection after each change, the dynamical ET should support several queries.

The basic query is ET membership: While the graph is dynamically changing, intrusion detection engineer would like to know which ET tree contains a given node. The goal of this query is to check whether or not a VM is destroyed or a new VM is created. We use *Member*(u) to represent this basic query.

The second query is ET connectivity: while the graph is dynamically changing, intrusion detection engineer would like to know whether two given nodes u_1 and u_2 are in the same tree. The goal of this query is to check whether or not a VM is migrated. We use *Connect*(u_1, u_2) to represent this connectivity query.

The third query is ET update: If the node is accompanied by additional values, intrusion detection engineer also would like to know whether or not this information is changed when the graph is dynamically changing. The goal of this query is to check whether or not the state of a VM is updated. We use *Updatekey*(u) to represent this update query.

4.2 Constructing ET Trees

In this subsection, we present the process that we use to construct the ET trees from a fully dynamic graph. The whole process includes four stages.

Initially, cluster the dynamic graph into dynamic forests. In this stage, the architecture graph is partitioned into a collection of associated trees, such that each change involves only a small number of such tress. The goal of partition is to minimize the cost of maintaining the dynamic graph. For a given $DG = (U', E')$, clustering is used to partition the edge set E' into different collections, so that each subtree has the same vertex set U' and maintains a subset information about the edges. Next, walk over the substree and replace all edges of subtree with directed edges by *Euler* tour. Then, build up a dynamic balanced binary tree over the *Euler* tour given above around the subtree. Thus, the leaves of the balanced binary tree are the nodes of the *Euler* tour of the second stage. Finally, repeat the operation of the second and third stage until all subtrees produced in the first stage are exhausted.

4.3 Query Algorithms

After constructing ET trees, the edges of DG are partitioned into h levels and $\cup_h E_i = E'$, and for $i \neq j$, $E_i \cup E_j = \emptyset$. For each i, we keep a *Euler* tree $ET_i = (U_i, E_i)$ as a spanning tree which is on level i.

Query *Connect*(u_1, u_2):

In connectivity query, the deletions-only connectivity algorithm is efficient for all operations used in our algorithm. Thus, we choose *Connected*(u_1, u_2) mentioned above, whose main ideal is using a function *Replace*(e, h) to argue that if a replacement edge exists, it will be found. The algorithm is described as follows.

```
// $u_1$ and $u_2$ are in the same $ET$ tree
while ($ET_i$ is not exhausted)
{
compute $Connected(u_1, u_2)$ in $ET_i$;
if ($Connected(u_1, u_2) = true$)
return $true$ and exit from the algorithm;
}
// $u_1$ and $u_2$ are in the same graph
while ($ET_i$ is not exhausted)
```

```
{
compute $Connected(root, u_1)$ in $ET_i$;
while ($ET_j \neq ET_i$)
{
compute $Connected(root, u_2)$ in $ET_j$;
if ($\prod_{x=1}^{2} Connected(root, u_x) = true$)
return true and exit from the two while loop;
}
}
```

Query *Member*(u):

We will consider two operations: vertices insertion and deletion. When a VM is created, we maintain the dynamic graph by inserting vertices and edges into ET trees. The insertion is adding a vertex u to U_l and adding edge $\langle root, u\rangle$ into DG and ET_h. When a VM is destroyed, we maintain the dynamic graph by deleting the corresponded node and the edges.

We can use $Size(u)$ to discover whether or not a node is in the graph. The algorithm is extremely simple, and it is described as follows.

```
while ($ET_i$ is not visited)
{
compute $Size(u)$ in $ET_i$;
if ($Size(u) > 0$)
return true and exit from the algorithm;
}
return false;
```

Query *Updatekey*(u):

To discover whether or not the state of a node is changed, the dynamic graph maintains additional values. The additional details are specified by security related attributes of the attacker and the VM-based computer system, such as trust relation between VMs, model of the intruder, application actions, computing system topology etc. The algorithm *Updatekey*(u) is described as follows.

```
while ($Connect(root, u) = true$)
{
compute $Minkey(u)$ in $ET_i$;
while ($ET_j \neq ET_i$)
{
compute $Minkey(u)$ in $ET_j$;
}
if (the two $Minkey(u)$ is equal)
return false and exit from the algorithm;
}
return true;
```

4.4 Dynamic Intrusion Detection Algorithm

The basic idea of dynamically detecting intrusion is to define a period δ and monitor the dynamic changes of virtual computing environment periodically using the three query algorithms mentioned above. During this period δ, $\det ect\_A\_I()$ is used to detect attacks and intrusions. If there are changes, reconstruct the dynamic graph and ET trees. Then, backtrack to the *root* vertex of changed ET trees and detect intrusions by $\det ect\_A\_I()$ corresponding to various changes. The dynamic intrusion detection algorithm is specified as follows.

Step 1: Initialization

Firstly, construct the dynamic graph DG for a given virtual computing environment and compute the ET trees for this DG described above. Initially, the set of ET trees has the same root vertices since there is only one.

Next, define the period δ. The period δ should be larger than minimal limitation since the $AGAS$ and $p(O|\lambda)$ computation are atomic operations which cannot be interrupted. Moreover, the time gap between query process and detection process is also an important factor affecting the period time. Thus, $\delta \geq t_{AGAS} + t_{p(O|\lambda)} + |t_{query} - t_{\det ect\_A\_I}|$.

Step 2: Monitor changes of virtual computing environment and adjust value of δ according to the dynamic state.

If a period is not over, detect attacks and intrusions using $\det ect\_A\_I($. algorithm during this period δ. If a period is over and no changes occur, increase δ. If a period is over and changes occur, decrease δ and go to step 3.

Step 3: Reconstruct ET trees changed and update the dynamic graph DG.

In the case of insertion, add a vertices u to U_l and adding edge $\langle root, u\rangle$ into ET_h and update DG.

In the case of deletion, find the ET_i containing the specified vertices u, then delete vertices u and delete edges which connected with u in ET_i. Finally, update DG.

In the case of migration, find the ET_i containing the emigrated vertices u_1 and ET_j containing the immigrated vertices u_2. Then delete u_1 and the related edges from ET_i, and insert vertex u_2 and an edge $\langle root, u_2\rangle$ into ET_j as a leaf node.

Step 4: If δ is higher than the minimal time gap, repeat to execute from step 2 every period δ.

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

As an experiment, we set up a prototype implementation of our automated approach to dynamic intrusion detection and give the experimental results. We first give an overview of the implementation presenting system architecture and core system components of our prototype system, then describe performance evaluation on detecting intrusion. For comparison, we also describe the experimental results with respect to how to achieve performance improvement. Finally, we quantify the virtualization overhead and performance impact of our approach compared to static methods.

5.1 Implementation Overview

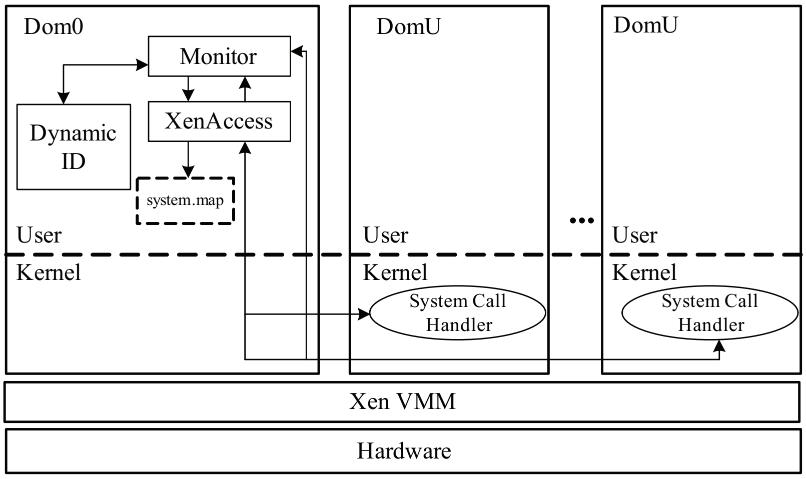

Fig. 3. Prototype system architecture

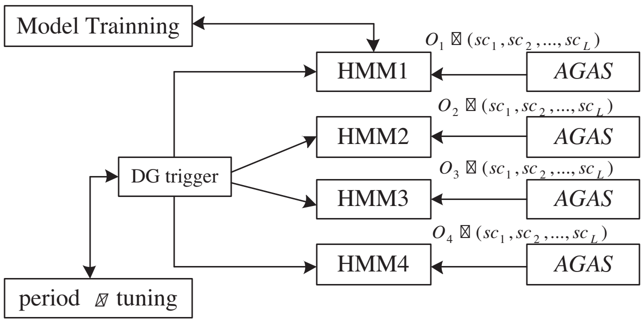
Model Trainning
O_1 ⊠ $(sc_1, sc_2, ..., sc_L)$
HMM1
AGAS
O_2 ⊠ $(sc_1, sc_2, ..., sc_L)$
HMM2
AGAS
DG trigger
O_3 ⊠ $(sc_1, sc_2, ..., sc_L)$
HMM3
AGAS
O_4 ⊠ $(sc_1, sc_2, ..., sc_L)$
HMM4
AGAS
period ⊠ tuning

Fig. 4. Dynamic intrusion detection

IDS is a monitoring system which reports alarms to the system operator when there are abnormalities. Our prototype relies on the fact that an application deviates from its normal behavior during execution when intruders achieve their goals successfully. Figure 3 depicts the system architecture of our prototype, which is running on *Fedora* 8, programmed by C/C++, storing data in database MySql, constructing model using HMM *Toolbox* of UBC. The *Monitor* requests to view a kernel symbol and *XenAccess* introspects the semantic model of an application's behavior in terms of sequences of system calls. *XenAccess* finds the virtual address for the kernel symbol in *system.map* and returns a pointer and offset for the data page to monitor with read/write privileges. The *System Call Handler* is developed to capture and monitor system calls. We include the automated approach to intrusion detection in dynamic environment in Figure 4, which is the component *Dynamic ID* in our prototype.

When implementing the experiment, *Monitor* in Dom0 executes monitoring tasks though two approaches. One approach monitors guest VM though *XenAccess*, which introspects *system.map*, more implementation detail can be seen in [27]. The other approach monitors guest VM by hunting system calls of guest OS. For detailed programming, system calls are handled by *sysenter* including *sysenter_CS*, *sysenter_ESP* and *sysenter_EIP*. Firstly, *sysenter_EIP* is caught, and the value of *sysenter_EIP* is replaced by another false value. Then, guest OS triggers a page fault, which is hunted as an event by *Xen*. Next, *Xen* compares the virtual address of page fault event and the false value mentioned before. If these two values are the same, the operation of guest OS is certified as a system call. Finally, we get some key values in Dom0, such as *process id*, *Eax* etc. According to this approach, *Monitor* can handle every system call of guest OS.

Dynamically detecting and responding process works as follows: Firstly, Computer Immune Systems (CIS) Data sets of University of New Mexico (UNM) are used as our training data to construct HMM statistic models though HMM *Toolbox*, since it is difficult to collect real-life datasets due to various limitations [35]. In CIS Data sets, some of the normal data are synthetic and some are live. Synthetic traces are collected in production environments by running a prepared script, live normal data are traces of programs collected during normal usage of a production computer system [36]. For example, the *sendmail* program data includes a total of 85 tracks, in which the normal tracks are 79 and the attack tracks are 6 containing *sccp* attacks and *decode* attack. Secondly, dynamic graph is constructed and *ET* tree is computed according to initial state of the computing system in *DG trigger*. For every vertex of *ET* trees build an *AGAS* engine and a HMM check engine to classify attacks an intrusions using algorithm $\det ect\_A\_I($.. The maximal frequent sequences of system calls generated by *AGAS* engine are the input observations of HMM check engine. When programming, dynamic graph is stored as adjacency matrix and *ET* trees are stored in heap for performance. Thirdly, the period listening process executes queries to track changes of virtual computing environment. While the execution environment is dynamically changing, the dynamic graph is tuned periodically. Consequently, the structures of *AGAS* engines and HMM check engines are also adjusted. In this stage, we only adjust the changed vertices and the edges in *ET* trees it related for minimal overhead. Two implemented problems are focused: Signal is used to trigger periodical tuning and features of HMM are extracted and transformed as relational data for quick searching.

In our experiments, the experimental hardware platform of prototype includes two computers: one is running 3 guest VMs; the other is a backup machine for migration. *Xen* 3.2 and *Fedora Core* 8 have been preinstalled on each machine. During the execution, the guest VMs can migrate from one computer to another. Based on this hardware platform, the proposed algorithm is quite critical for the efficiency of intrusion detection since we wish to maximize both speed and detection capabilities. Therefore, we require that HMM patterns match exactly. To achieve this goal, we construct the hidden Markov model in various sequence lengths of system call off-line from training data sets. We implement this model based on system call sequences whose length is six according to Forrest's research. The HMM size generated is presented in Table 1.

A careful look at the HMM of system call sequences generated shows that there is redundancy between short sequences and long sequences. Therefore, it is necessary to eliminate this redundancy for improving the matching efficiency. It is difficult to find which method is the best way since the elimination process is accompanied by loss of accuracy; but it seems that the long sequences are better to represent the program action. Thus, we prune HMM of short sequences if they are subsequence of long sequences. An example of redundancy rate is presented in Figure 5 when the length of system call sequences is eight.

Program	Process number	System call number	HMM size ($L = 6$)
MIT *lpr*	2 703	2 926 304	170
UNM *lpr*	4 298	2 027 468	178
named	27	9 230 572	17
xlock	72	16 937 816	20
login	12	8 894	6
ps	24	6 144	9
inetd	3	541	2
stide	13 726	15 618 237	613
sendmail	71 760	44 500 219	861

Table 1. HMM size generated

5.2 Algorithm Efficiency

In the first experiment, we evaluate efficiency of algorithm $det\,ect\_A\_I()$ with *detection rate* (*DR*) and *false positive rate* (*FR*) on several typical intrusion behaviors. *DR* = Number of intrusions identified/Total number of anomalous system events, which is the correct rate of intrusions identified. *FR* = Number of normal events misidentified as intrusions/Total number of intrusions identified, which is the error rate of normal data identified as anomalous.

We compare the proposed algorithm $det\,ect\_A\_I()$ with traditional intrusion detection algorithm using HMM [5], HMM combined with sliding window [37], automated intrusion detection method [6] and intrusion detection approach though

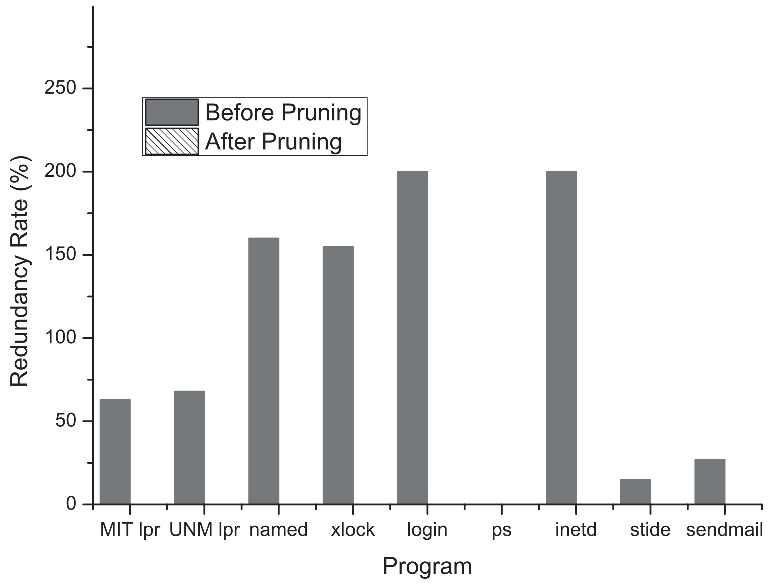

Fig. 5. HMM redundancy rate when sequence length is 8

Livewire [8]. According to research in [2, 3, 35], we set sequence length of system call as 6 and also adopt 6 as sliding window size. As a rough estimate, on condition of HMM probability threshold $\xi = 0.001$, the average detection rate of algorithm $\det ect\_A\_I()$ on these 9 data sets is about 96.3 %, the average false positive rate is below 0.15 %. The maximal detection rate is up to 97 % on *sendmail* program and the minimal detection rate is a little more than 94 % on *xlock* program. Compared with other related approaches mentioned above, algorithm $\det ect\_A\_I()$ has higher detection rate and almost the similar false positive rate on *lpr* and *ps* program. It has lower false positive rate and similar detection rate on *named*, *xlock*, *login* and *inetd* program. Especially, it has not only higher detection rate but also lower false positive rate on *sendmail* program. The *stide* program is not shown here because its detection rate is higher but its false positive rate is also higher and it is difficult to define whether it is good or not. Figures 6 and 7 give the relationships between probability threshold and efficiency. In Figure 6, the y-axis presents the detection rate on *sendmail* program, the x-axis describes the HMM probability threshold of ξ ranging from 1e−10 to 0.01. In Figure 7, the y-axis presents the false positive rate on *sendmail* program, the x-axis describes the HMM probability threshold of ξ ranging from 1e−10 to 0.01. Note that the false positive rate of $\det ect\_A\_I()$ is much smaller in Figure 7 as $\xi \geq 0.001$. In many cases, it is benefit for users to verify malicious system call sequences from huge amount of system processes.

In the next experiment, we test dynamic capability of our approach by creating, destroying a guest VM and migrating a guest application. We set up a HTTP server

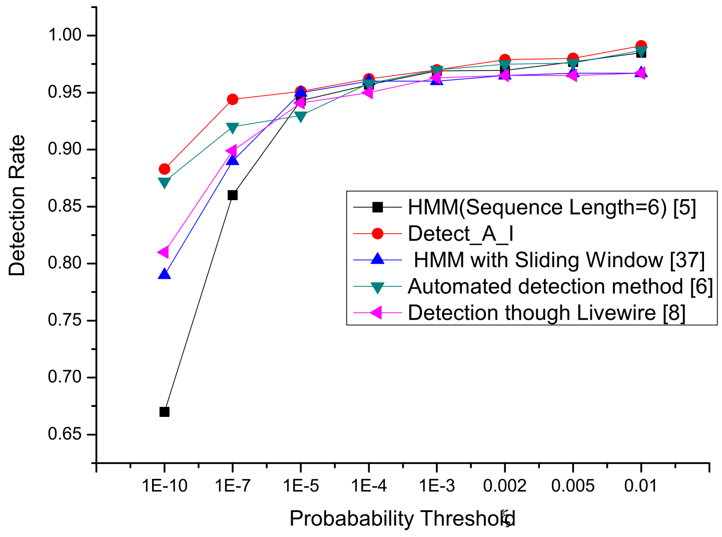

Fig. 6. Comparison of detection rate on *Sendmail*

(Apache Http Server) in a guest VM and a client VM to request a specified file to test our algorithm considering parameter on file size. This specified file is continuously written to increase the size. To make the observation convenient, we send http request 10 000 times. Figure 8 describes the comparison on dynamic det$ect_A_I()$ and HMM detection method when migrating a guest application of a VM. With increasing document length, the request times per second decrease. The file size is from 1 KB to 1 024 KB, the migrating process is executed when file size is larger than 128 K. If a file is too small, it is too difficult to observe the changing state. Figure 9 describes the creation and destruct response times per second. In Figure 9, we cannot compare dynamic det$ect_A_I()$ and HMM detection approach in that the response times is zero. For a created VM or a destroyed VM, HMM method cannot run automatically until operated by security administrator.

5.3 Performance

The additional overhead, which is the ratio of time consumption using assigned approach and time consumption in normal computing environment, of our approach includes two aspects: the monitor overhead using virtual memory introspection and periodical refresh overhead of dynamic structure. As an example, t_1 is the time of constructing a graph, t_2 is the total time of constructing a dynamic graph with periodically tuning structure; then the refresh overhead of dynamic structure is $t_2/t_1 \times 100\,\%$.

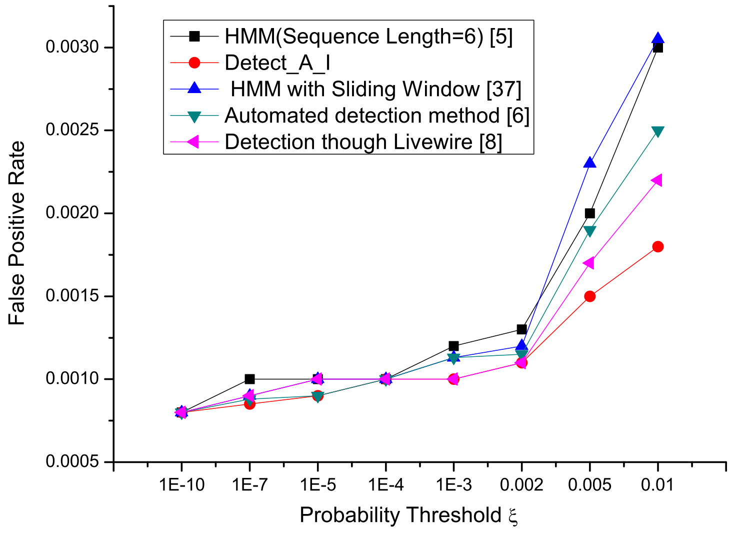

Fig. 7. Comparison of false positive rate on *Sendmail*

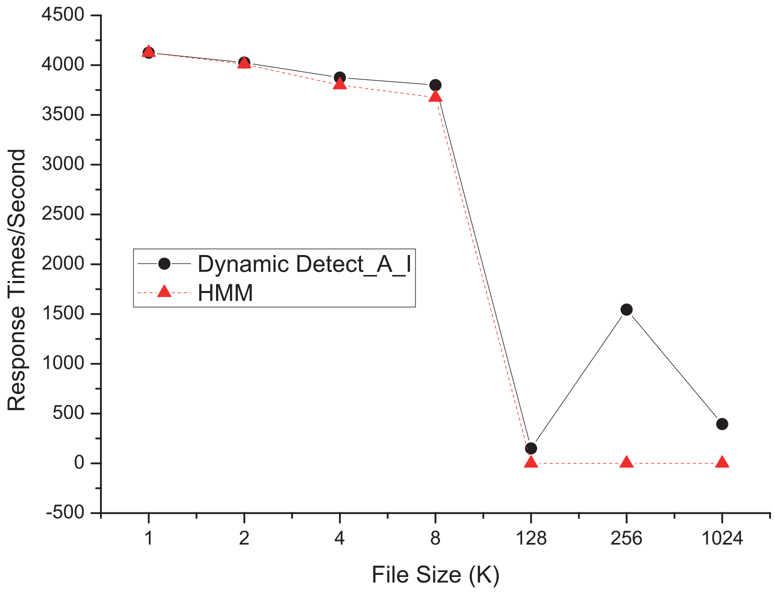

Fig. 8. Comparison of response times when migrating

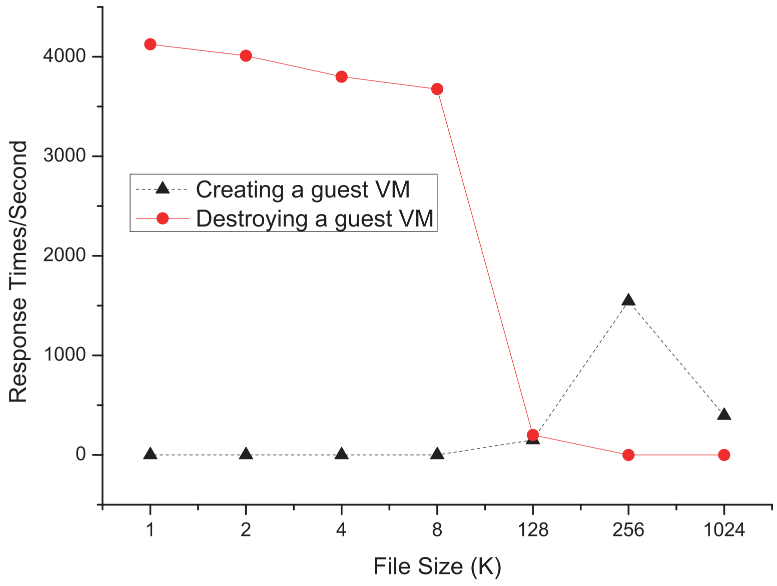

Fig. 9. Response times when creating and destroying

Using *XenAccess*, the average time to complete the specified function in a target domain is about 52 microseconds by virtual address memory access, the average time for monitor to read memory is about 2.4 microseconds when data size is 2 000 bytes. The time of refreshing dynamic structure is tightly related to the changing size. If only one vertex in an ET tree is changed, the time on refreshing ET tree and dynamic graph is about 0.8 microseconds; but the total time is not a simple plus, it greatly depends on the structure of dynamic graph. As a rough test, the additional writing time is about more than 35 %. If the dynamic graph is drastically changing, that is if the period δ is much larger than the gap between two changes, the overhead is very high, up to 440 %. Fortunately, the proposed tuning mechanism of period δ can reduce this kind of overhead to some degree. With this tuning mechanism, the maximal overhead is about 240 %.

6 CONCLUSION

Because virtual machine-based computing environments are continuously changing, it is difficult to build high-quality intrusion detection systems. In this paper, we propose an automated approach to intrusions detection in order to maintain sufficient performance and reduce dependence on execution environment. We present a dynamic graph structure to monitor the dynamic changing of computing environment. Based on this structure, we discuss a hidden Markov model strategy for abnormality detection using frequent system call sequences, letting us identify attacks and intrusions automatically and efficiently. We also propose an automated mining algorithm, named *AGAS*, to generate frequent system call sequences. Rather than setting a user-defined threshold on mining frequent sequences, *AGAS* algorithm utilizes related probabilities to identify frequent sequences. In our approach, the detection performance is adaptively tuned according to the execution state every period. To improve performance, the period value is also under self-adjustment. Our experimental results show that the proposed algorithm and method are effective compared to traditional HMM-based intrusion detection approach.

Compared to traditional intrusion detection method based on HMM, dynamic det *ect_A_I*() is more adaptive and effective. The advantages of our approach include:

1. High automatic and safety: *AGAS* algorithm supports self-generation on frequent sequences without human interference. The output of *AGAS* algorithm is the input of HMM checker, which avoids interference from attackers.
2. High performance: The experimental results on different data sets show that our approach has higher detection rate or lower false positive rate, especially on *sendmail* dataset.
3. Real-time: The system monitor adjusts the system architecture every period using dynamic graph, which assures the detection engineer achieves the real-time information.

The self-tuning of period value is also a benefit to enhance the real-time ability.

We plan to extend this approach to build a real-time intrusion detection system on *Xen*-based virtual computing environment. We have further noticed that if system behavior changes drastically, the overhead of dynamic graph might be increased and the benefit of our approach will be diminished. We will focus on this topic. Another direction is to combine our approach with the flexibility of virtual machine to build intrusion tolerance system (ITS) to tolerate rather than prevent intrusion.

Acknowledgement

This work is partially supported by National Natural Science Foundation of China under Grant 60803114, National 973 Basic Research Program of China under Grant 2007CB310900.

REFERENCES

[1] Garfinkel, T.—Rosenblum, M.: When Virtual Is Harder Than Real: Security Challenges In Virtual Machine Based Computing Environments. In: Proceedings of the 10$^{\text{th}}$ Conference on Hot Topics in Operating Systems – HOTOS05, Santa Fe, June 2005, pp. 20–26.

[2] Lee, W.—Stolfo, S. J.: A Framework for Constructing Features and Models for Intrusion detection systems. ACM Transactions on Information and System Security, Vol. 3, 2000, No. 4, pp. 227–261.

[3] Maggi, F.—Matteucci, M.—Zanero, S.: Detecting Intrusions Through System Call Sequence and Argument Analysis. IEEE Transactions on Dependable and Secure Computing, Vol. 7, 2010, No. 4, pp. 381–395.

[4] Ertoz, L.—Eilertson, E.—Lazarevic, A.—Tan, P.—Srivastava, J.—Kumar, V.—Dokas, P.: The MINDS – Minnesota Intrusion Detection System. Next Generation Data Mining, MIT Press 2004.

[5] Joshi, S. S.—Phoha, V. V.: Investigating Hidden Markov Models Capabilities in Anomaly Detection. In: Proceedings of 43rd Annual Southeast Regional Conference, Kennesaw, GA, March 2005, pp. 98–103.

[6] Yu, Z. W.—Tsai, J. J. P.—Weigert, T.: An Automatically Tuning Intrusion Detection System. IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics, Vol. 37, 2007, No. 2, pp. 373–384.

[7] Jiang, X. X.—Wang, X. Y.—Xu, D. Y.: Stealthy Malware Detection and Monitoring Through VMM-Based Out-of-the-Box Semantic View Reconstruction. ACM Transactions on Information and System Security, Vol. 13, 2010, No. 2, Article 12.

[8] Garfinkel, T.—Rosenblum, M.: A Virtual Machine Introspenction Based Architecture for Intrusion Detection. In: Proceedings of Network and Distributed Systems Security Symposium, California, February 2003, pp. 126–133.

[9] Dunlap, G. W.—King, S. T.—Cinar, S.—Basrai, M.—Chen, P. M.: Revirt: Enabling Intrusion Analysis Through Virtual Machine Logging and Replay. In: Proceedings of 2002 Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, December 2002, pp. 211–224.

[10] King, S.—Chen, P.: Subvirt: Implementing Malware With Virtual Machines. in: Proceedings of 2006 IEEE Symposium on Security and Privacy, Oakland, California, May 2006, pp. 314–327.

[11] Payne, B. D.—Carbone, M.—Sharif, M.—Lee, W.: Lares: An Architecture for Secure Active Monitoring Using Virtualization. In: Proceedings of 2008 IEEE Symposium on Security and Privacy, California, USA, May 2008, pp. 233–247.

[12] Sharif, M.—Lee, W.—Cui, W. D.: Secure In-Vm Monitoring Using Hardware Virtualization. In: Proceedings of 16th ACM Conference on Computer and Communications Security, Illinois, USA, November 2009, pp. 477–487.

[13] Jiang, X. X.—Xu, D. Y.—Wang, Y. M.: Collapsar: A VM-Based Honeyfarm and Reverse Honeyfarm Architecture for Network Attack Capture and Detention. Journal of Parallel and Distributed Computing, Vol. 66, 2006, No. 9, pp. 1165–1180.

[14] Asrigo, K.—Litty, L.—Lie, D.: Using VMM-Based Sensors to Monitor Honeypots. In: Proceedings of the 2nd International Conference on Virtual Execution Environments – VEE 2006, Ottawa, Canada, June 2006, pp. 13–23.

[15] Wang, P.—Wu, L.—Cunningham, R.—Zou, C. C.: Honeypot Detection in Advanced Botnet attacks. International Journal of Information and Computer Security, Vol. 4, 2010, No. 1, pp. 30–51.

[16] Reiser, H. P.—Kapitza, R.: Fault and Intrusion Tolerance on the Basis of Virtual Machines. Tagungsband des 1. Fachgespräch Virtualisierung. Paderborn, Germany.

[17] Scales, D. J.—Nelson, M.—Venkitachalam, G.: The Design and Evaluation of a Practical System for Fault-Tolerant Virtual Machines. Technical Report VMware-TR-2010-001, May 11, 2010.

[18] Matthews, J. N.—Herne, J. J.—Deshane, T. M.: Data Protection and Rapid Recovery from attack With a Virtual Private File Server and Virtual Machine Appliances. In: Proceedings of the IASTED International Conference on Communication, Network and Information Security, Phoenix, USA, November 2005, pp. 170–181.

[19] Sun, W. C.—Chen, Y. M.: Vmitn: A Novel Intrusion Tolerance Architecture for Treating The Rapid Propagation of Malicious Programs. In: Proceedings of the 20th ACM International Conference on Supercomputing (ICS 2006), Queensland, Australia, June 2006.

[20] Nguyen, A. Q.—Takefuji, Y.: A Novel Approach for a File-System Integrity Monitor Tool of Xen Virtual Machine. In: Proceedings of 2nd ACM Symposium on Information, Computer and Communications Security (AsiaCCS 2007), Singapore, March 2007, pp. 194–202.

[21] Nguyen, A. Q.—Takefuji, Y.: A Real-Time Integrity Monitor for Xen Virtual Machine. in: Proceedings of The IEEE International Conference on Networking and Services, ICNS06, Silicon Valley, July 2006, pp. 90–98.

[22] Junior, V. S.—Lung, L. C.—Correia, M.—Fraga, J. S.—Lau, J.: Intrusion Tolerant Services Through Virtualization: A Shared Memory Approach. In: Proceedings of 2010 24th IEEE International Conference on Advanced Information Networking and Applications, Perth (Australia), April 2010, pp. 768–774.

[23] Sabhnani, M.—Serpen, G.: Application of Machine Learning Algorithms to Kdd Intrusion Detection Dataset Within Misuse Detection Context. In: Proceedings of International Conference of Machine Learning: Models, Technology and Applications, MLMTA2003, Las Vegas, June 2003, pp. 209–215.

[24] Perdisci, R.—Lanzi, A.—Lee, W.: Classification of Packed Executables for Accurate Computer Virus Detection. Pattern Recognition Letters, Vol. 29, 2008, No. 14, pp. 1941–1946.

[25] Khanna, R.—Liu, H.: Control Theoretic Approach to Intrusion Detection Using a Distributed Hidden Markov Model. IEEE Wireless Communications, Vol. 15, 2008, No. 4, pp. 24–33.

[26] Hu, J. K.—Yu, X. H.–Qiu, D.—Chen, H.: A Simple and Efficient Hidden Markov Model Scheme for Host-Based Anomaly Intrusion Detection. IEEE Network, January/February 2009, pp. 42–47.

[27] Payne, B. D.—Carbone, M. D. P. A.—Lee, W.: Secure and Flexible Monitoring of Virtual Machines. In: Proceedings of 23rd Annual Computer Security Applications Conference, ACSAC 2007, Miami Beach (Florida), USA, December 10–14 2007, pp. 385–397.

[28] Rajagopalan, M.—Hiltunen, M. A—Jim, T.—Schlichting, R. D.: System Call Monitoring Using Authenticated System Calls. IEEE Transactions on Dependable and Secure Computing, Vol. 3, 2006, No. 3, pp. 216–229.

[29] Khanna, R.—Liu, H.: Control Theoretic Approach to Intrusion Detection Using a Distributed Hidden Markov Model. IEEE Wireless Communications, Vol. 15, 2008, No. 8, pp. 24–33.

[30] Abdel-Galil, T. K.—El-Saadany, E. F.—Youssef, A. M.—Salama, M. M. A.: Disturbance Classification using Hidden Markov Models and Vector Quantization. IEEE Transactions on Power Delivery, Vol. 20, 2005, No. 1, pp. 2129–2135.

[31] Mehta, V.—Bartzis, C.—Zhu, H.—Clarke, E.—Wing, J.: Ranking Attack Graph. In: Proceedings of 9th International Symposium on the Recent Advances in Intrusion Detection, RAID 2006, Hamburg, Germany, September 2006, pp. 127–144.

[32] Bessani, A.—Sousa, P.—Correia, M.—Neves, N. F.—Verissimo, P.: Cheap Intrusion-Tolerant Protection for Crutial Things. Technical Report, DI-FCUL, TR-2009-14, June 2009.

[33] Demetrescu, C.—Italiano, G. F.: A New Approach to Dynamic All Pairs Shortest Paths. Journal of the Association for Computing Machinery, Vol. 51, 2004, No. 6, pp. 968–992.

[34] Tarjan, R. E.—Werneck, R. F.: Dynamic Trees in Practice. Journal of Experimental Algorithmics, Vol. 14, December 2009, pp. 4.5–4.23.

[35] Parampalli, C.—Sekar, R.—Johnson, R.: A Practical Mimicry Attack Against Powerful System-call monitors. In: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, Tokyo (Japan), March 2008, pp. 156–167.

[36] Computer Immune Systems Data sets. Available on: `http://www.cs.unm.edu/~immsec/data-sets.htm`.

[37] Raman, C. V.—Negi, A.: A Hybrid Method to Intrusion Detection Systems Using HMM. In: Proceedings of Second International Conference on Distributed Computing and Internet Technology, Bhubaneswar, India, December 2005, pp. 389–396.

Feng Zhao is currently an Associate Professor of computer science, Huazhong University of Science and Technology, China. He received his PhD degree in computer science from Huazhong University of Science and Technology in 2006. His research interests include data mining, security and distributed computing.

Hai Jin is currently a Professor of computer science at Huazhong University of Science and Technology, China. He is the Dean of College of Computer Science and Technology of the same university. His current research interests include cloud computing, virtual computing, computer architecture etc.