HODetector: The Hidden Objects Detection Based on Static Semantic Information Library Outside Virtual Machine

YongGang Li[1, 2], ChaoYuan Cui[1], BingYu Sun[1], WenBo Li[3]

[1] Institute of Intelligent Machine, Chinese Academy of Sciences, China
[2] School of Information Science and Technology, University of Science and Technology of China, China
[3] Institute of Technology Innovation, Chinese Academy of Sciences, China
lygzr@mail.ustc.edu.cn, {cycui, bysun, wbli}@iim.ac.cn

Abstract

With the spread of malwares, the security of virtual machine (VM) is suffering severe challenges recent years. Rootkits and their variants can hide themselves and other kernel objects such as processes, files, and modules making malicious activity hard to be detected. The existed solutions are either coarse-grained, monitoring at virtual machine level, or non-universal, only supporting specific operating system with specific modification. In this paper, we propose a fine-grained approach called HODetector based on static semantic information library (SSIL) to detect the hidden objects outside VM. We have deployed HODetector prototype on Xen virtualization platform and used it to detect the processes, files, and modules hidden by rootkits. The experiment results show that HODetector is effective for different rootkits and general for Linux operating system with various kernels.

Keywords: Virtualization, Fine-Grained detection, Semantic gap, Rootkit, Hidden objects detection

1 Introduction

Virtualization technology [1, 19] is the most important technical support for cloud computing. It can significantly increase computing resource utilization. Unfortunately, some new security challenges have been proposed with the application of the virtualization technology. A threat report from Rising [2] shows that: the total number of new viruses is still upward trend, and more than 43.27 million samples of new viruses were intercepted in 2016. According to the report, the security problem of virtualization has become increasingly serious. Malware, especially rootkit, can use various technologies to hide their and others presence. As a result, it can bypass the detection of anti-virus software leading to a significant threat to the existed operating system (OS) such as Linux. Therefore, it is very important to detect the hidden objects for virtual machine security.

For virtualization security, the traditional secure tools detecting malwares are placed into the guest VM (GVM) that may be injected by computer viruses. So, it's possible that the secure tool will be bypassed or cheated. For example, a rootkit named *f00lkit* can bypass *Chkrootkit* and *Rkhunter*, the most popular anti-malwares tools in Linux. Compared with the traditional method, one mechanism called out-of-box [3] detecting malwares out of VM is a better way, because the secure tool is outside any span of malwares. Then another problem appears: semantic gap [4]. The secure tool needs to get high level semantic information of GVM to judge if there exists intrusion. While the secure tool is out of GVM, and what it gets all are original contents of memory (0 and 1). So the key technology of out-of-box is how to resolve the problem of semantic gap.

Virtual machine introspection (VMI) [5] can translate original memory contents of GVM into high-level semantic information. It mainly solves the problem of semantic gap. VMI has been adopted by a great number of security system tools based on virtualization technology and widely used in malware detection [6], process detection [7], intrusion detection [8], etc. The approaches of VMI are various and the SSIL we proposed is also one kind of VMI tools that can solve the semantic gap problem.

In this paper, we design HODetector to detect the hidden objects including processes, files, and modules based on SSIL. It can monitor different Linux OSes outside VM without any specific modification to kernel, which improves the generality. Through testing on different rootkits, we found that the hidden objects can be detected by HODetector and the time overhead is in accptable range.

2 Related Works

Varieties of approaches have been proposed to monitor and detect GVM recent years. The most

*Corresponding author: ChaoYuan Cui; E-mail: cycui@iim.ac.cn
DOI: 10.3966/160792642018091905011

popular method is placing secure tool outside VM. Researches on how to bridge semantic gap and get the running state of GVM effectively have been done recent years.

DARKVFU [9] is a dynamic malware analysis system detecting attacks in windows. It makes use of Rekall [18] to parse the debug data provided by Microsoft to establish a map of internal kernel functions. Then, DARKVFU locates the kernel base address in memory by reading registers FS and GS and parsing *_KPCR* structure. After getting kernel base address it can trap all kernel functions via break point injection. Through monitoring the execution of specific funtions (eg. allocating kernal heap and accessing files, al.) and analysing kernal objects DARKVFU can find malwares. Unfortunately, it incurs a larger performance overhead for cpu and memory.

Xie and Wang [10] proposed a rootkit detection mechanism based on deep information extraction and cross-vertification at hypervisor level. To locate the target memory they extract kernel symbol table "*System.map*" of GVM and find out the address of process "*init_task*" in it. Starting with the address, the process list of GVM can be acquired by traversing doubly linked list. Files and network connections can be reconstructed according to the association of kernel objects. Cross-vertification between hypervisor and GVM is the last step to find rootkit. Both information extraction and cross-vertification depend on "System.map" that may be changed after restarting even during running by GVM. So the accuracy of detection results cannot be guaranteed.

VMDriver [11] is a novel approach for virtualization monitoring. It separates event sensor and semantic construction in VMM and management domain. VMDriver can list the information of all processes in GVM. But VMDriver needs a third-party tool to distinguish guest OS type and kernel version, which limits its generality.

Libvmi [12] is modified based on Xenaccess [13] and employs VMI technology to monitor VM. It can monitor the guest OS out of VM, and overcome semantic gap problem. However, the deployment needs to configure the file *Libvmi.conf* manually for different OSes, which may result in unexpected errors.

Compared with libvmi and [10], HODetector can be deployed more easily, because there is no specific modification and manual operation for OS or kernel to be done. Different with VMDriver, HODetector develops a naming protocol to distinguish guest OS type and kernel version. Besides, HODetector has built SSIL that can improve system generality and reduce performance overhead. Another contribution of HODetector is adopting message-passing mechanism to reduce the impact of VM state inconsistency [14] on results accuracy.

3 Designs

Figure 1 shows the overview of HODetector. It mainly contains two parts: one resides in secure VM (SVM) and another in guest VM (GVM). First, Message Sender generates a message and a signal simultaneously. The signal is used to wake up Memory Locating Module in SVM. The message will be transmitted to Message Receiver in GVM. After receiving the message, Message Receiver also generates a signal activating Back-end Module. As a result, Memory Locating Module and Back-end Module will take actions synchronously. Then Back-end Module captures the current semantic view of GVM (guest_view) by excuting instructions such as *ls* and *ps*. At the same time Memory Locating Module locates the virtual addresses of target objects of GVM. Next, the targeted virtual addresses will be transfered to Memory Mapping Module for memory reading and translating. Both Memory Locating Module and Memory Mapping Module need retrieve the semantic information indexed by SSIL to get the semantic view of SVM (secure_view). At last, secure_view and guest_view are sended to Judge Module judging if there exist hidden objects.

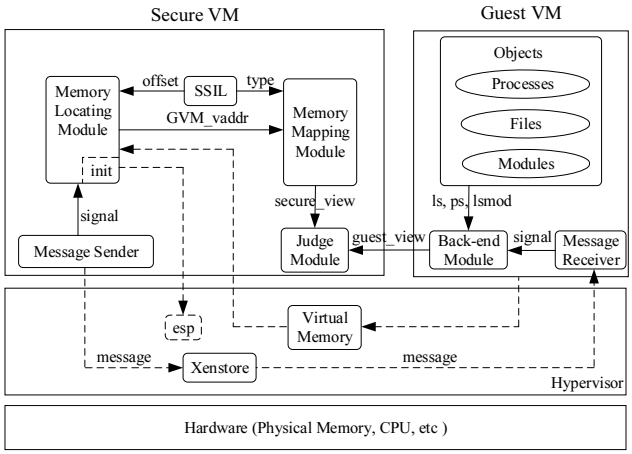


Figure 1. The overview of HODetector

3.1 The Message-Passing Mechanism

The problem of the inconsistency of the observed VM state is an unavoidable problem in VMI technolgy. Inconsistency appears during VMI tools introspecting a live system while its state is changing in the observed data structures. According to [14], inconsistencies can be classified into two categories: intrinsic inconsistencies and extrinsic inconsistencies. Any inconsistency may lead to a misjudgement. For example, a new process generated after having gotten guest_view while the secure_view is not to be captured will be treated as a hidden object. The message-passing mechanism is designed to reduce the inconsistency influence. It transmits message between SVM and GVM through Xenstore. The mechanism can guarantee secure_view and guest_view start to be generated at

the same time point. As a result, the two semantic views are almost consistent in space and time.

3.2 SSIL

SSIL is designed to improve HODetector's generality and resolve the problem of semantic gap. To build SSIL, all prevalent kernel versions are collected and analyzed. Every kernel version represents a subset named as the corresponding kernel version in SSIL. It contains crucial kernel information including some data types and offsets of certain entries in specific data structures such as *task_struct* and *dentry*. The offsets are used to locate the virtual address of GVM, and the data types are used to determin how many bytes to read in physical memory and what high-level semantic information should be reconstructed outside GVM. Building SSIL is offline, so it would not affect the running speed of VM.

In HODetector a protocol is defined: every VM must be named as (Release Version@user):(Kernel Version). For example, the linux OS ubuntu12.04-64 with kernel 3.2.0-23 may be named as ubuntu12.04-64@××:3.2.0-23. As a result the kernel version can be extracted from the VM name according to the name protocol. The extracted kernel version will be used to retrieve specific semantic information in SSIL for VMI.

3.3 Memory Locating Module

Memory Locating Module is designed to locate the virtual addresses of the target objects of GVM. There is a subset module called init module uesd for initialization. Init module gets the context of VCPU of GVM and extracts the esp register firstly. To get starting address of kernel stack of current process, init module excutes the code esp&~ (THREAD_SIZE - 1). The executing result is the starting address of *thread_info* whose first entry points to *task_struct* of the current process. Memory Locating Module can get the starting addresses of certain dada structures such as *dentry* through reading memory content pointed by corresponding pointers. Then it locates the viratual address of targeted entry by adding starting address and entry offset retrieved in SSIL according to entry name.

3.4 Memory Mapping Module

Due to the isolation mechanism between VMs, every operation about reading GVM memory in SVM should be handled by Memory Mapping Module. It utilizes the function *map_page()* containing *xc_map_foreign _range()* and *xc_translate_foreign_address()* in the library *libxc* for memory mapping. After mapping, Memory Mapping Module reads specific amount memory content according to data type retrieved in SSIL. What it gets all are raw bits that is difficult to describe the VM state. So Memory Mapping Module translates them into high-level semantic information according to corresponding data types.

3.5 Capture Process List

A process is an instance of a computer program that is being executed. Every process in Linux is described by a process control block defined as a data structure called *task_struct* which contains process name, pid, etc. All the running processes are linked by a doubly linked list implemented as members *next* and *prev* in *task_struct*. HODetector can capture all processes of GVM by traversing the doubly linked list. Figure 2 shows the procedure.

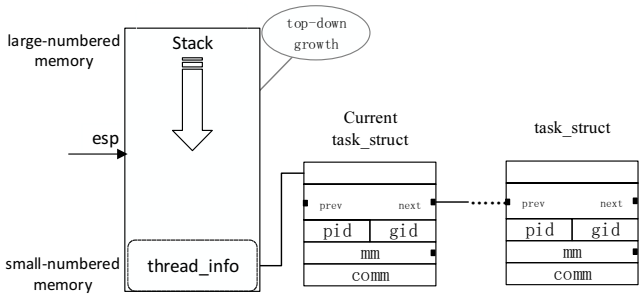


Figure 2. Process capture

3.6 Capture File Tree

In general, a complete Linux OS is composed of a large number of files organized in the form of tree. In addition to leaf nodes, all nodes of the tree represent directory names. The virtual file system uses a data structure defined as *dentry* to describe directory and file. All files in the same directory are linked by a doubly linked list named *d_child*. In *dentry*, the subdirectory is pointed by *d_subdirs* and the parent directory is pointed by *d_parent*. To get *dentry* several data structures should be traversed. The procedure shows in Figure 3. Along the pointer *d_parent* HODetector searches up according to the principle of depth first until getting the root directory. Next it searches down level by level according to the principle of breath first until getting the whole subtree specified before searching. That is to say HODetector can get the whole file tree via recursion method if it can get one *dentry*.

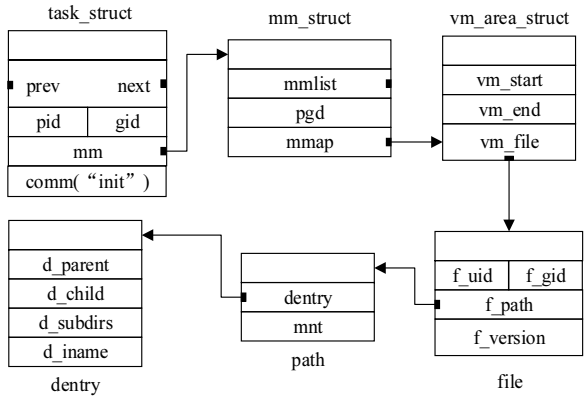


Figure 3. The procedure of getting *den try*

3.7 Capture Module List

Linux OS allocates a data structure defined as *module* to describe kernel module. All kernel modules are also stored in a doubly linked list shown as Figure 4. After getting a node address, HODetector can detect all the kernel modules by traversing doubly linked list. It gets a module starting address in a file named *System.map* that is a kernel symbol table containing kernel module address in the directory */boot* in GVM.

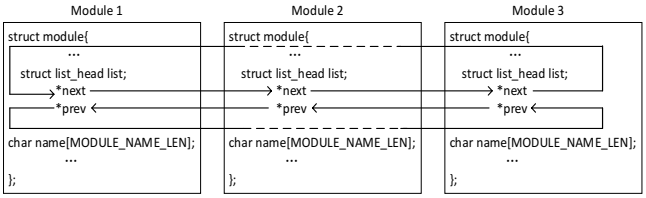


Figure 4. Module List

3.8 Definition and Algorithm

Definition 1. Define the process list as *P*, the file list as *F*, and the module list as *M*.
P = {(process1_pid, process1_name), (process2_pid, process2_name) …}
F = {(file1_name, file1_path), (file2_name, file2_path) …}
M= {(module1_name, module1_size), (module2_name, module2_size) …}
Definition 2. Define the guest view as $gV = \{P, F, M \mid$ acquired in guest VM$\}$.
Definition 3. Define the secure view as $SV = \{P, F, M \mid$ acquired in secure VM$\}$.
Definition 4. Define the key entry in certain data structure as φ_k , define the offset of φ_k in data structure as ζ_k , and define the type of φ_k as θ_k .
Definition 5. Define the certain data structure that contains some entries as $\Gamma_m = \{(\varphi_1,(\zeta_1,\theta_1)),(\varphi_2,(\zeta_2,\theta_2)), ...,(\varphi_k,(\zeta_k,\theta_k)),...\}$.
Definition 6. Define the subset of SSIL as Φ_n whose name is kernel version. $\Phi_n = \{\Gamma_1,\Gamma_2,...,\Gamma_n,...\}$.
Definition 7. Define the SSIL as $SL = \{\Phi_1,\Phi_2,...,\Phi_n,...\}$.

Algorithm 1 shows how HODetector detects the hidden objects. First, it selects the matched subset of SSIL with the index of VM name (lines 2-3). The function *get_content_of_entry()* (line 4) is used to get the high-level semantic of entry φ_k . Second, HODetector gets the starting address of *task_struct* through VCPU context (lines 5-8). The function *read_pointer()* (line 8) can read the address that is pointed by pointer entry in data structure through address mapping. HODetector gets the starting module address by executing specific command in GVM (line 9). To get *dentry* six data structures are traversed, which means calling *read_pointer()* five times (line 10). Then HODetector builds the secure view *SV* in SVM (lines 11-15). Functions *search_up()* and *search_down()* adopt recursive algorithm following the principle depth-first and breath-first respectively. Line 16 gets the guest view *gV* in GVM. At last, Jude Module judges if there exist hidden objects through cross-view contrast (lines 17-19).

Algorithm 1. Detect the Hidden Objects

(1) **Require:** $(\varphi_k,(\zeta_k,\theta_k)) \in \Gamma_m \in \Phi_n \subset SL$
(2) VM_name = get_dom_name_from_domID (int domID)
(3) Φ_n =get version from name (char * VM name)
(4) get_content_of_entry(base_addr, φ_k){
  (a) ζ_k = search_entry_offset(φ_k)
  (b) θ_k = search_entry_type(φ_k)
  (c) entry_addr=base_addr+ ζ_k
  (d) entry_map_addr=map_page(entry_addr, domID)
  (e) data=(θ_k *) read_mem (entry_map_addr, sizeof(θ_k))
  (f) return data}
(5) vcpu_context=get_vcpu_context(int domID, int vcpu)
(6) esp=vcpu_context→esp
(7) thread_info_addr= esp &~(THREAD_SIZE - 1)
(8) task_struct_addr=read_pointer(thread_info_addr)
(9) module_addr={$sed - n '/d modules $/p' System.map | command executed in GVM}
(10) dentry_addr= task_struct→mm→mmap →vm_file→f_path→dentry
(11) get_secure_view(task_struct_addr,module_addr, dentry_addr){
(12) **Get P:**
  (a) **do**
  (b) {task_struct_next=read_pointer(task_struct _addr+procrss_next_offset)
  (c) process_name= get_content_of_entry(task _struct_next,comm)
  (d) process_pid= get_content_of_entry(task _struct_addr,pid)
  (e) } while(task_struct_next!=task_struct_addr)
(13) **Get F:**
  (a) file_name= get_content_of_entry(dentry _addr,d_iname)
  (b) root_dir=search_up(dentry_addr)
  (c) mached_file=search_down(root_dir)
(14) **Get M:**
  (a) **do**{module_next=read_pointer(module _addr+module_next_offset)
  (b) module_name=get_content_of_entry (module_next,name)
  (c) module_size=get_content_of_entry (module_next,core_text_size)
  (e) } while (module_next!=module_addr)
(15) $SV = \{P, F, M \mid$ acquired from above procedures$\}$

(16) $gV = \{P, F, M \mid$ acquired from commands executed in GVM$\}$
(17) **if** (object $\in SV$ && object $\notin gV$)
(18) **Judge:** the object is hidden
(19) **end if**

4 Experiments and Result

Xen [15] is an open-source virtual platform that allows multiple guest operating systems to execute on the same computer hardware with high performance and strong resource isolation. The software architecture of Xen consists of hypervisor and domains including the only privileged domain named dom0 (SVM) and unprivileged domains named domU (GVM). We implement HODetector on Xen 4.1.2. Table 1. shows the specific configuration of experimental environment.

Table 1. Experimental environment

VM	Dom0	Dom1	Dom2	Dom3	Dom4	Dom5
OS	Ubuntu	CentOS	Ubuntu	Ubuntu	Ubuntu	Ubuntu
Edition	Desktop	Server	Server	Desktop	Server	Server
Version	12.04-64	6.4-64	10.04-32	5.04-32	12.04-64	10.04-64
Kernel	3.2.16	2.6.32	2.6.32	2.6.10	3.2.0-23	2.6.32
CPU	4cores, 3.2GHz	1core, 2.2GHz	1core, 2.2GHz	1core, 2.2GHz	1core, 2.2GHz	1core, 2.2GHz
Memory	4G	1G	1G	1G	1G	1G

4.1 Rootkit

Rootkit is a set of program or code that stealthily bypasses anti-virus and takes control of the computer. Most existed rootkits can be classified into user-level and kernel-level rootkit [16]. The former runs in user space of OS and replaces critical system utilities with modified versions to hide process, file, module, etc. The latter runs in kernel space and modifies kernel for hiding leading to a greater threat to OS. For example, adore-ng [17], a kernel-level rootkit, infects the kernel as a loadable kernel module (LKM). A user can use the control program *ava* in adore-ng to hide process, file, network connection, etc. Apart from adore-ng there are kinds of rootkits such as modulerootkit and suterusu, etc, and these rootkits will be tested in our experiment.

4.2 Detection

The rootkit detection results of HODetector are shown in Table 2. The symbol "√" represents HODetector can detect the hidden objects in experiment; "×" represents it does not work; and "– " represents the rootkit has no this functionality.

Table 2. Detection results

Rootkit	Adore-ng	Modulekit	Suterusu	Wnps	Kbeast	F00lkit	maK_it	brootus	Diamorphine
DomU	Dom1	Dom1	Dom2	Dom3	Dom2	Dom4	Dom5	Dom2	Dom4
Process Detection	√	–	√	√	√	√	–	√	√
File Detection	√	–	√	√	√	√	–	√	√
Module Detection	–	√	–	–	×	–	×	√	×

No matter the VM is 32bit or 64bit, desktop version or server version, even with different kernels, HODetector can work well without any specific modification. Because SSIL contains the semantic information of all kinds of kernels shielding implementation details in OS. However, through experiments we also find HODetector cannot detect the module that is hidden by removing from doubly linked list (eg. Kbeast). HODetector traverses doubly linked list to find all moules loaded in GVM. As a result, it cannot find the hidden one removed from doubly linked list. A possible solution is staring at the instruction "insmod". When "insmod" is executed, the allocated memory will be recorded. The solution feasibility will be verified in our future work.

Chekrootkit and *rkhunter* are the most popular detection tools detecting rootkits in Linux. They are selected for comparision with HODetector. Table 3 shows the detection results. The results show that HODetector is more effective than *Chekrootkit-0.49* and *rkhunter-1.4.0.* Different with *Chekrootkit* and *rkhunter,* HODetector is implemented out of GVM leading to stricter isolation, which makes it hard to be bypassed or cheated.

Table 3. Detection result comparison

Rootkit	adore-ng	modulekit	suterusu	wnps	kbeast	f00lkit	maK_it	brootus	Diamorphine
Chkrootkit-0.49	√	×	×	√	×	×	√	×	×
Rkhunter-1.4.0	×	×	√	√	√	×	×	√	×
HODetector	√	√	√	√	√	√	√	√	√

4.3 Time Analysis

4.3.1 Time Percentage

HODetector mainly gets three kinds of objects ouside GVM: process list, file tree, and module list. The taken time of capturing 300 objects (100 processes, 100 files, and 100 modules) is recorded, and the time percentage is shown in Figure 5.

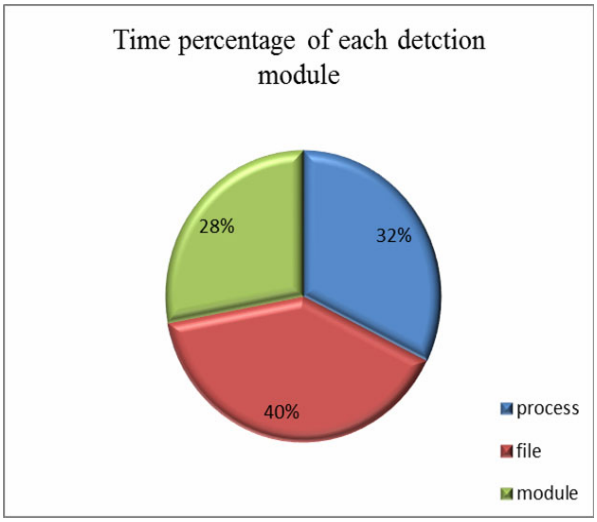


Figure 5. Time percentage

In Figure 5, we can find file capture is the most time-consuming and module capture is the least one. The cost time is mainly determined by the scanning path. File capture must traverse six data structures to get the file name while module capture traverses only two. As a result, their cost time is different.

4.3.2 Time Stability

To observe the time stability of HODetector, we test the taken time getting each kind of objects whose counts are linearly for increases. Figure 6 shows the test results. The results show that the taken time is also linear growth, which turn out HODetector has good stability in time. HODetector takes most time in getting memory and parsing it. The taken time is proportional to the amount of memory that is linear to the numbers of processes. As a result, the taken time of HODetector is linear to the numbers of processes.

4.3.3 Time Overhead of GVM

In this section, we evaluate the impacts of HODetector on the performance of GVM. We choose the application *tar* in GVM as a benchmark and measure its execution time when the target objects increase. The *tar* benchmark compresses a folder whose size is 35.6 MB containing 2186 files into a package. When the counts of objects are zero, it means HODetector does not work and we set the time

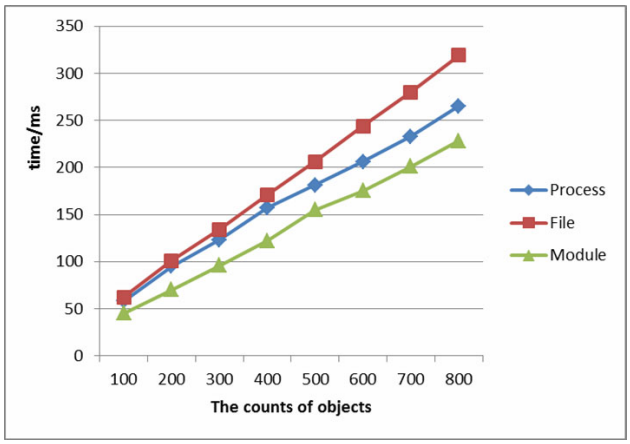


Figure 6. Time stability

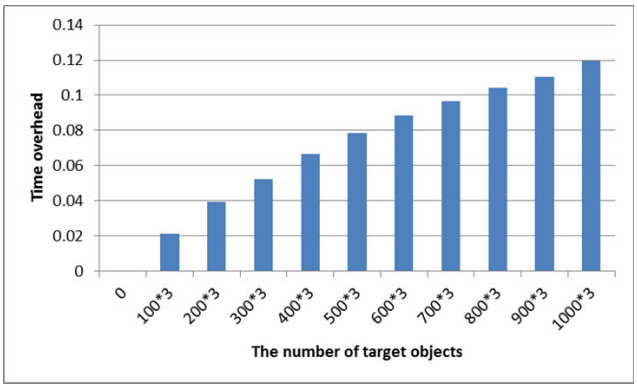


Figure 7. Time overhead of GVM

overhead is zero. Figure 7 shows the experimental results. From the figure we find when we increase the counts of objects, the overhead is increasing. However, when each kind of objects are 1000, that is to say all the three (processes, files, and modules) are 3000, the overhead is less than 12%. In general, there are less than 100*3 objects to be detected in GVM, which means the overhead is only 2%. The time overhead of GVM mainly comes from the impact on physical hardware of HODetector. HODetector needn't capture kernel semantic information online reducing time overhead, and SSIL can provide complete information set whenever HODetector works. So it introduces very low overhead in GVM.

4.3.4 Time Overhead of SVM

To evaluate the impact of HODetector on SVM, we adopt *nbench* to test the performance of SVM when HODetector detects different numbers of objects. The test results of *nbench* mainly include three facts: MEM, INT, and FP. MEM reflects the performance of processor bus, CACHE and memory; INT represents integer processing performance; FP reflects double-precision floating-point performance. The test results are represented as scores that are proportional to system performence. Figure 8, Figure 9, and Figure 10 show the test results.

The test results show that the performance

degradation of SVM will increase with the number of target objects. When the target objects are 100*3, MEM performance degradation is about 0.9%; INT performance degradation is about 0.5%; FP performance degradation is about 0.6%. The three kinds of performance degradation will increase to 4.5%, 2.8%, and 2.9% respectively when the target objects increase to 1000*3. The results show that HODetector has little more impact on SVM storage performance. Any operation about reading GVM memory shuould be handled by Memory Mapping Module when HODetector works. To read the memory of a process in x86-64 architecture, a time of reading memory and four times of memory mapping (corresponding four-level page conversion in x86-64 architecture) will be executed. Every memory mapping operation need set permission to target memory and copy raw bits through hypercall in SVM. All the above operations incure more overhead to memory.

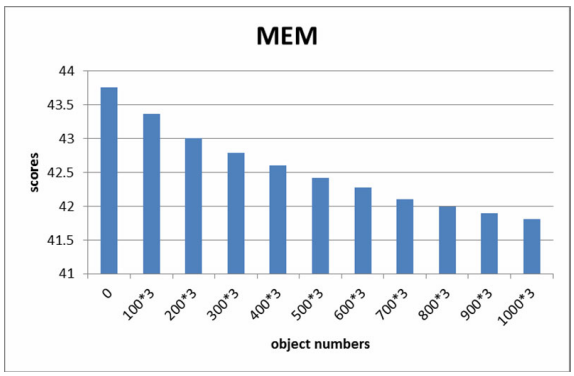


Figure 8. MEM test result

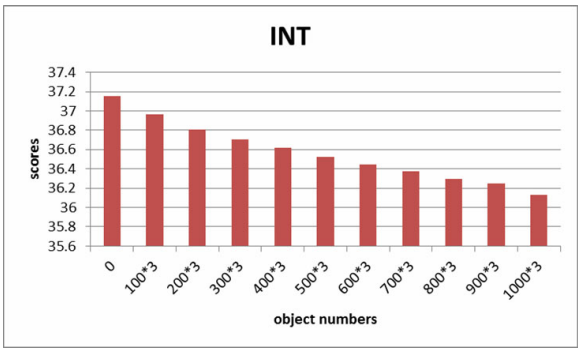


Figure 9. INT test result

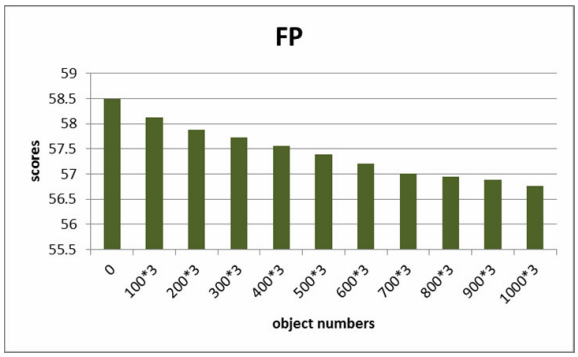


Figure 10. FP test result

5 Conclusions

In this paper, we propose an approach to detect the hidden objects in VM called HODetector. It detects guest OS at process, file, and module level out of GVM. HODetector can solve the semantic gap problem and improve the generality through SSIL. The detection results show that HODetector can detect the hidden objects manipulated by rootkit except the module hidden by removing itself from doubly linked list. Through the analysis of cost time, we find HODetector has good time stability, which is important in large-scale cluster system. The overhead result shows that HODetector is efficient to detect the hidden object.

Acknowledgments

This project is supported by the National Nature Science Foundation of China (No. 31371340), the National Key Technology R&D Program (No. 2014BAD10B08) and the National Key Technologies Research and Development Program of China under Grant (No. 2016YFB0502604).

References

[1] R. Kumar, S. Charu, An Importance of Using Virtualization Technology in Cloud Computing, *Global Journal of Computers & Technology*, Vol. 1, No. 2, pp. 56-60, February, 2015.

[2] Rising, *Information Security Report of China in 2016*, http://it.rising.com.cn/dongtai/18659.html

[3] X. Jiang, X. Wang, D. Xu, Stealthy Malware Detection and Monitoring through VMM-based "out-of-the-box" Semantic View Reconstruction, *ACM Transactions on Information & System Security*, Vol. 13, No. 2, Article No. 12, February, 2010.

[4] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, R. Sion, SoK: Introspections on Trust and the Semantic Gap, *2014 IEEE Symposium on Security and Privacy*, San Jose, CA, 2014, pp. 605-620.

[5] H. W. Baek, A. Srivastava, J. V. D. Merwe, CloudVMI: Virtual Machine Introspection as a Cloud Service, *IEEE International Conference on Cloud Engineering*, Boston, MA, 2014, pp. 153-158.

[6] M. Vlad, H. P. Reiser, Towards a Flexible Virtualization-Based Architecture for Malware Detection and Analysis, *2014 25th International Workshop on Database and Expert Systems Applications, Munich, Germany*, 2014, pp. 303-307.

[7] M. A. A. Kumara, C. D. Jaidhar, Virtual Machine Introspection based Spurious Process Detection in Virtualized Cloud Computing Environment, *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, Noida, India, 2015, pp. 309-315.

[8] J. Saxon, B. Bordbar, K. Harrison, Introspecting for RSA Key

Material to Assist Intrusion Detection, *IEEE Cloud Computing*, Vol. 2, No. 5, pp. 30-38, September-October, 2015.

[9] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, A. Kiayias, Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System, *Proceedings of the 30th Annual Computer Security Applications Conference*, New Orleans, LA, 2014, pp. 386-395.

[10] X. Xie, W. Wang, Rootkit Detection on Virtual Machines through Deep Information Extraction at Hypervisor-level, *IEEE Conference on Communications and Network Security*, National Harbor, MD, 2013, pp. 498-503.

[11] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, F. Zhao, VMDriver: A Driver-Based Monitoring Mechanism for Virtualization, *IEEE Symposium on Reliable Distributed Systems*, New Delhi, India, 2010, pp. 72-81.

[12] H. Xiong, Z. Liu, W. Xu, S. Jiao, Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM, *IEEE 12th International Conference on Computer and Information Technology*, Chengdu, China, 2012, pp. 549-556.

[13] A. TaheriMonfared, M. G. Jaatun, Handling Compromised Components in an IaaS Cloud Installation, *Journal of Cloud Computing*, Vol. 1, No. 1, pp. 1-21, August, 2012.

[14] S. Suneja, C. Isci, E. De Lara, V. Bala, Exploring VM Introspection: Techniques and Trade-offs, *ACM Sigplan Notices*, Vol. 50, No. 7, pp. 133-146, July, 2015.

[15] Xen Project, *Linux Foundation Collaborative Project*, https://www.xenproject.org/

[16] S. A. Musavi, M. Kharrazi, Back to Static Analysis for Kernel-Level Rootkit Detection, *IEEE Transactions on Information Forensics & Security*, Vol. 9, No. 9, pp. 1465-1476, September, 2014.

[17] Y. Ding, H.-Y. Fu, Y.-Z. Li, Research on VFS Layer Rootkit Technique in Linux, *Computer Engineering*, Vol. 36, No. 8, pp. 161-162, April, 2010.

[18] Rekall Forensics, Rekall Memory Forensic Framework, http://www.rekall-forensic.com/

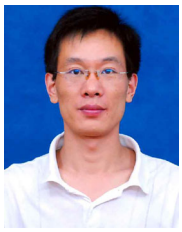
[19] M. Sato, T. Yamauchi, VMM-Based Log-Tampering and Loss Detection Scheme, *Journal of Internet Technology*, Vol. 13, No. 4, pp. 655-666, July, 2012.

Biographies

Yonggang Li, born in 1988, Ph.D. Candidate. His current research interests include operating system security, virtualization and cloud computing.

Chaoyuan Cui, born in 1972. Ph.D., associate professor. His main research direction includes system virtualization, architecture of cloud computing, information and communication security.

Bingyu Sun, born in 1974. Ph.D., professor. His main research direction includes, data mining and machine learning.



Wenbo Li, born in 1979. Ph.D., associate professor. His current research interests include information technology, water resource, remote sensing, and intelligent decision.