RootkitDet: Practical End-to-End Defense against Kernel Rootkits in a Cloud Environment

Lingchen Zhang[1,2,4], Sachin Shetty[2], Peng Liu[3], and Jiwu Jing[1]

[1] State Key Laboratory of Information Security,
Institute of Information Engineering, Chinese Academy of Sciences
[2] College of Engineering, Tennessee State University
[3] College of IST, Penn State University
[4] University of Chinese Academy of Sciences

Abstract. In cloud environments, kernel-level rootkits still pose serious security threats to guest OSes. Existing defenses against kernel-level rootkit have limitations when applied to cloud environments. In this paper, we propose RootkitDet, an end-to-end defense system capable of detecting and diagnosing rootkits in guest OSes with the intent to recover the system modifications caused by the rootkits in cloud environments. RootkitDet detects rootkits by identifying suspicious code region in the kernel space of guest OSes through the underneath hypervisor, performs diagnosis on the code of the detected rootkit to categorize it and identify modifications, and reverses the modifications if possible to eliminate the effect of rootkits. Our evaluation results show that the RootkitDet is effective on detection of kernel-level rootkits and recovery modifications with less than 1% performance overhead to the guest OSes and the computation and network overhead is linear with the quantity of the VM instances being monitored.

Keywords: Hypervisor, VM, Kernel-level rootkit, Defense, Cloud.

1 Introduction

A kernel rootkit is a form of malware that may subvert the kernel to achieve various goals, especially hiding certain malicious processes from security monitoring, anti-virus software, intrusion detection, and VMI (virtual machine introspection). A typical way for a kernel rootkit to achieve its goals is to modify certain kernel data structures. During the past 10 years, kernel-level rootkits have been emerging as a major security threat. For example, MacAfee Avert Labs [1] reported that during the three-year period from 2004-2006, the number of rootkits had increased 600 percent. Rootkits have been leveraged by criminals to conduct bank fraud [2].

In this paper, we focus on defending against kernel rootkits in a cloud environment. In such cloud environments as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), kernel rootkits should be as useful to criminals/attackers as in non-cloud environments. Taking IaaS as an example, kernel rootkits may enable the criminal to keep a backdoor in a VM (virtual machine) for the attacker to gain whole control of the guest

M. Kutyłowski and J. Vaidya (Eds.): ESORICS 2014, Part II, LNCS 8713, pp. 475–493, 2014.

operating system. They may also hide some other malware which may inflict serious damage or launch stealthy attacks. Due to the hiding, this malware can become difficult to detect or eliminate by the administrator.

In a cloud environment, cloud providers are responsible for countering kernel rootkits in tenant VMs as they can fully leverage the security features of the underneath hypervisors. We focus on cloud environments since besides standard requirements such as effectiveness and efficiency, cloud environments have several unique requirements regarding how kernel rootkits should be countered. (R1) End-to-end defense is highly desired. Besides detecting a rootkit, cloud administrators also need to quickly reverse the malicious modifications made by the rootkit to its target VM. If the admin has to manually diagnose and reverse the malicious modifications, the availability and business continuity loss could be too much to be accepted by the tenant. (R2) Scalable defense. The total defense cost should be linear (if not sublinear) in the number of VMs being simultaneously protected. The defense should also facilitate dynamic addition and deletion of VMs. (R3) Adoptable defense. The defense should be compatible with existing commercial (and open source) cloud platforms.

Although many research works have been done to tackle kernel rootkits, existing defenses are limited in meeting the requirements cloud environments have. To see how existing defenses are limited, let us break down the existing kernel rootkit defenses into 4 classes which we will review shortly in Section 6: (1A) Detecting modified control or non-control data or violations of invariants [3] [4] [5] [6]. (1B) Preventing installation of kernel rootkits by performing analysis on the code being loaded into the kernel space [7] [8] [9]. (1C) Defending kernel rootkits by cooperating with anti-malware software [10] [11]. (1D) Protecting the kernel by restoring infected kernels to healthy state [12]. We may briefly summarize the limitations of these defenses in terms of the requirements as follows. (a) Defenses in Class 1A and 1C are not end-to-end or focus only on control data. (b) Defenses in Class 1B and 1D might be defeated by the rootkits leveraging novel techniques or kernel vulnerabilities [13] and some of them are not very easy to be adopted because they are designed based on special hypervisors. (c) Defenses in Class 1C are not very scalable because they have to create multiple instances of anti-malware sofware to monitor multiple VMs.

To overcome the above limitations, in this paper we propose RootkitDet, an end-to-end defense against kernel rootkits in a cloud environment. RootkitDet works as follows. First, it detects the kernel rootkits by looking for suspicious code in the kernel space of the guest OSes. Second, once a rootkit is detected, it will do diagnosis to precisely identify kernel data structures that were maliciously modified by the rootkit. Third, it attempts to reverse the modifications. Due to the following insight, RootkitDet employs a simple detection idea. *Insight*: A registration procedure can be leveraged to enable separation between legitimate code and rootkit code in the kernel space. After a rootkit is detected in guest OSes, RootkitDet attempts to eliminate the effect of the rootkit. RootkitDet first performs static analysis on the suspicious code to collect certain characteristic information of the rootkit. Then, it tries to categorize the rootkit heuristically according to the

collected characteristic information. If the rootkit can be categorized, RootkitDet would be able to identify the kernel data structures that were malicious modified by the rootkit. Finally, RootkitDet reverses the modifications as follows: it restores the modified control data with pre-known values, and recovers the broken links between the modified non-control data and other data structures.

We have designed and implemented a RootkitDet system prototype atop KVM [14] (`qemu-kvm-1.2.0`).Our evaluation results show that RootkitDet can meet the requirements cloud environments have on kernel rootkits defense. In sum, our main contributions are as follows.

- RootkitDet offers end-to-end defense against kernel rootkits in a cloud environment: from detection to diagnosis to recovery. To the best of our knowledge, RootkitDet is the first work that focuses on end-to-end defense in cloud environments.
- RootkitDet is an effective defense. The evaluation results show that RootkitDet can detect a kernel rootkit as long as the rootkit inserts code into the kernel space of a guest OS. RootkitDet can do recovery (i.e., reverse the modifications by the rootkit) if the rootkit is categorized successfully.
- RootkitDet is a practical defense against kernel rootkits in cloud environments. The evaluation results show that the average performance overhead introduced in guest OSes is less than 1% when the max detection cycle is 16 seconds, and the total defense cost (CPU, network bandwidth) is linear in the number of the VMs being protected.

2 Threat Model and Assumptions

In this section, we present the threat model and assumptions for the kernel-level rootkits detection system.

2.1 Threat Model

A cloud user allocates several virtual machines to provide some services, web-based service most popular, to customers. We consider an attacker who intends to install a rootkit into the kernel of VMs to keep the control of the system and hide himself. Upon successful installation of a kernel-level rootkit, the attacker will control the entire VMs and do whatever attacks he wants to except system crash or DoS. Following are examples of attacks after a successful kernel-level rootkit installation: collection of confidential data, arbitrary modification of all memory contents, abuse of the computing capacity and network bandwidth.

To install the rootkit into the VMs, the attacker may take advantages of zero-day vulnerabilities in the kernel and application software running in the VMs to gain privilege of arbitrary code execution step by step. Due to the various objectives, the attackers have to craft specific code and insert them into the kernel space of the VMs. Return-oriented kernel-level rootkits are out of our scope because they can be used to install a rootkit but not run as long-term rootkits and are not reentrant for different processes. Due to similar reasons, DKOM(Direct Kernel Object Manipulation), ret-to-user and rootkits that are erased immediately after executed are also out of our scope.

2.2 Assumptions

We assume that modern CPUs of X86 architecture provide NX (non-executalbe) feature as part of page-based memory protection. For the sake of simplicity, we assume that the kernel of guest OSes supports loadable kernel modules(LKMs) and a LKM may be dynamically loaded into the kernel either explicitly by the administrator of the guest OS or implicitly by an application running in the guest OS. Moreover, we assume that kernel and modules may be vulnerable, but not malicious. Rootkits can be installed into the running kernel space but not exist in the kernel or modules when the kernel is built.

3 Overview of RootkitDet System

The goal of RootkitDet system is to provide an end-to-end defense against kernel-level rootkits in cloud environments. To achieve its goal, RootkitDet system takes three steps: detection, diagnosis and recovery. In this section, we describe the overview of RootkitDet system and its architecture.

3.1 Overview

The first step of RootkitDet system is to detect the kernel-level rootkits installed into the guest OSes. RootkitDet system identifies suspicious code, which is taken as the code of rootkits, in the kernel space of guest OSes. By "suspicious", we mean a memory region that is not supposed to hold any code or a region that holds illegitimate code. Legitimate code in the kernel space of a guest OS which is not infected by rootkits comprises kernel code and the code of benign LKMs. To separate the code of rootkits from legitimate code, we introduce a simple, practical and effective *registration procedure*.

Registration procedure is a requirement of RootkitDet system that the administrator of a guest OS registers the kernel and potential LKMs of the guest OS in advance. *Registration of the kernel* provides enough information to bridge semantic gap [11] in our system. To register a kernel that will run in a guest OS, the administrator should provide the source code, configuration file, *system.map* as well as the binary file of the kernel. The kernel of a guest OS should be registered prior to the execution of the virtual machine which the guest OS runs on. *Registration of LKMs* is critical to separating legitimate code and the rootkits. To register a LKM that is probable to be loaded during the lifetime of the guest OS, the administrator should provide the module's name and object file. A module should be registered before it is loaded into the kernel, even if the guest OS is running. We suppose that registration procedure is performed through a secure channel, which is unknown to the attacker.

To detect suspicious code in the kernel space of a guest OS, RootkitDet system reconstructs the page directory of the kernel space of the guest OS, identifies all executable regions and compares them with expected executable regions which hold legitimate code. RootkitDet system works as follows: First, it detects

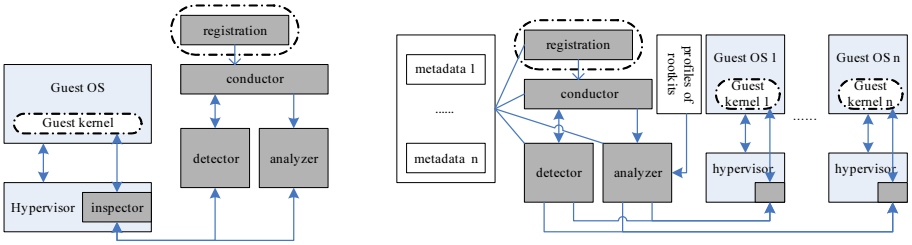

Fig. 1. Basic architecture (left) and scalable architecture (right) of RootkitDet

whether extra executable regions exist in the kernel space. Extra regions are different from that holds legitimate code. Second, it detects whether some code resides in unused space of modules. Finally, it detects malicious modifications to the legitimate code by computing SHA-1 checksums of the legitimate code and comparing them with expected values. Any mismatch means that legitimate code is modified by the rootkits.

The second step of RootkitDet system is to diagnose the detected rootkit. RootkitDet system attempts to categorize the detected rootkits and precisely identify the objects and data structures that are modified by the rootkit. To help categorization of a rootkit, we generate *profiles* of known typical rootkits in advance. RootkitDet system performs static analysis on the code of the detected rootkit to collect characteristic information, which is used to categorize the rootkit by matching with the profiles of known typical rootkits.

The profile of a typical rootkit includes: a) The tactic adopted by the rootkit to achieve its intention. We describe the tactic by a set of semantic actions, including external function calls, access to global variables and dynamic allocated data structures. b) The data structures that we should recover according to its tactic. In general, these data structures are dynamically allocated but we can find its location tracking down from a global variable with fixed location.

The final step of RootkitDet system is to recover the objects and data structures that were modified by the rootkit. The rootkit may make modifications to control data and non-control data. Control data are usually function pointers existing in kinds of data structures. Therefore, the expected values of control data are already known and it is easy to recover such modifications. By contrast, modifications to non-control data are various and usually there are no expected values for them. However, some modifications to non-control data break the links to other objects or violate some invariant that keeps in uninfected kernel. We can figure out how to recover such modifications in the kernel's context.

3.2 Architecture

As shown in Fig. 1, the basic system of RootkitDet comprises several components: registration, conductor, detector, analyzer and inspector. In our system, all components except inspector are independent of the hypervisor, and thus can run in a different OS running on a virtual machine or a physical machine.

Inspector. Inspector is integrated into the hypervisor to provide a reliable interface to access the kernel space memory and CPU registers of guest OS. This interface is used by detector and analyzer. It is worth noting that it is rarely necessary to stop running of the guest OS when the inspector reads or writes the memory of the guest OSes because our system accesses unusually changed memory during detection and recovery procedures in most of the time. Besides, inspector is easily developed in most cloud platforms due to its simplicity so that our system is easy to adopt by most cloud providers.

Detector. Detector performs three detection procedures to find out whether kernel-level rootkits exists in guest OS according to the commands coming from the conductor. In detection procedure 1, detector reconstructs the list of loaded modules and generates the list of executable regions in the kernel space, then compares them to find out whether extra executable regions exist besides the regions of the kernel code and registered modules. In detection procedure 2, detector checks whether some code resides in the unused space of each module. In detection procedure 3, detector calculates checksums for the code of the kernel and modules, and compares them with original ones, which are provided by the conductor, to check integrity of the legitimate code in the kernel space.

Detection procedure 1 and 2 might be bypassed because detector depends on the memory of guest OSes, which might be under the control of rootkits. For instance, a rootkit may tamper with the information of a module and change the module's code size to a bigger value, and put its code right behind the module's code, pretending itself as part of the module to escape from detection. We leave this problem to the conductor and the conductor resolves it when generating the original hash values for all of the modules.

Conductor. Conductor is the heart of our system. It periodically sends commands to detector to start detection procedures when the guest OS is running. Once rootkits are detected, it receives the detection report from detector, then raises an alert to the administrator and activates analyzer. Conductor also helps detector during detection procedure 3 by generating original checksums of the loaded modules of the guest OS as well as descriptions of each module, which are used to detect smart rootkits that escape from procedure 1 and 2.

Registration. Registration component stores information of the guest OSes provided by the administrator in registration procedure. It provides information of the kernel to bridge semantic gap in the three steps of RootkitDet system. Besides, it provides the necessary information of the kernel and legitimate modules to help RootkitDet system separate rootkits.

Analyzer. Analyzer diagnoses the code of the detected rootkit by performing static analysis to collect related characteristic information and attempts to categorize the detected rootkit heuristically. If the analyzer succeeds in categorizing the rootkit by matching the characteristic information with the profiles of known rootkits, it can finally perform recovery of the guest OSes.

The analyzer performs static analysis instead of dynamic analysis due to the following reasons. First, dynamic analysis is not applicable in practice due to its heavy overhead to guest OSes. Second, dynamic analysis requires the execution

of the code of rootkits to analyze its behavior while static analysis does not. Finally, the characteristic information collected through static analysis is enough in most cases although it is sketchy and rough.

In order to monitor multiple guest OSes simultaneously, we expands our system into a scalable architecture as shown in Fig. 1. For each guest OS, we generate related meta-data of the kernel in advance, which includes: (1) *system.map* which contains names and locations of the kernel symbols, (2) checksum of the kernel code which is used to detect modifications to the running kernel code, (3) definitions of important data structures that might be referred to by the rootkits or during recovery, (4) type information of important global variables and dynamically allocated objects and their relationship in the kernel. Our system takes advantages of the kernel's meta-data to detect kernel-level rootkits and perform recovery. Besides, only one kernel's meta-data is necessary if all of the guest OSes are using the same kernel. Furthermore, several guest OSes can also runs on the same hypervisor if the hypervisor supports multiple guest OSes.

4 Design and Implementation of RootkitDet

In this section, we describe the system design and the implementation of the prototype of RootkitDet system.

4.1 Detection

Registration procedure. As mentioned above, registration procedure comprises registration of the kernel and registration of the legitimate modules. When the kernel of a guest OS is registered, we generate the meta-data of the kernel. *System.map* is provided by the administrator of the guest OS. We compute the original checksum of the kernel code by analyzing the binary file of the kernel. Definitions of important data structures and type information of important global variables and dynamic allocated objects are excerpts of the source code of the kernel. By "important", we mean the data structures and objects that might be accessed directly or indirectly by known rootkits and that might be accessed to recover modifications caused by known rootkits.

A module can be loaded either automatically by applications or manually by the administrator. A module loaded into the kernel is identified by its name, which is obtained from the filename of its original object file. To guarantee that a legitimate module is not taken as a rootkit by our system, the module must be registered before it is loaded into the kernel. When a module is registered, we store its original object file with its original filename, and analyze it to extract information of its exported symbols, which are useful when calculating original checksums of modules that depend on it.

For the sake of efficiency and other purposes, self-modifying code might be used in the kernel and modules to leverage advanced features of CPU, which we need to take into account during the calculation of checksums. As self-modifying code runs only in initialization stage of the kernel, we can compute all possible

checksums of the kernel code by creating temporary VM instances with registered kernel and obtaining the kernel code after initialization stage. We generate the checksum of a module by simulating the relocation process of the module, and thus we replace customized instructions according to the state of the kernel to generate the proper original checksum of this module.

Detection procedures. To detect rootkits that insert code into the kernel space, our system performs three detection procedures as mentioned above. However, our system may raise false alarms in several situations. We discuss these problems and present solutions as follows.

Under some particular conditions, inconsistency between the executable regions and loaded modules may occur in the kernel of guest OSes, which causes a false positive in detection procedure 1. *Case 1:* When a module is loading, the kernel allocates another executable region for its initialization code, which is released immediately after the initialization code is executed. The temporary existence of initialization code of a module may cause a false positive. We confirm the detection of rootkits only when the detector continuously reports rootkits 3 times. *Case 2:* When a module is unloaded, the kernel doesn't release related regions until the total size reaches a threshold. The lazy clean-up may also cause false positive. We require a subtle modification to the kernel source code to release all free regions once a module is unloaded. This modification doesn't affect the efficiency of the kernel because unloading modules happens rarely in general.

Unused space usually exists below the code of a module because of the page-aligned allocation of memory. As far as we know, the kernel doesn't clear the memory regions allocated for modules before loading modules into them. As a result, the unused space may contain nonzero data, which cause a false positive in detection procedure 2. To eliminate this kind of false positive, we require a subtle modification to the kernel source code to clear the last page of memory regions allocated for modules.

The code of a module varies with the relocation address of the module when it is loaded into the kernel. We can't leverage previous work [9] to compute checksums of modules in detection procedure 3 because the original object files of modules are not required when the detector computes checksums in our system. To reduce the work of the detector, the original checksums of modules are provided by the conductor. The detector computes current checksums of legitimate code respectively, and compares them with original checksums. Any mismatch means modifications to the legitimate code.

Detection procedures are performed periodically instead of being triggered like Patagonix [15] due to the following reasons. First of all, the rootkits that are erased immediately after executed are out the scope of this paper, so periodic detection works properly in our system. Secondly, Patagonix also periodically performs a *refresh* to set all pages non-executable. Thirdly, our system focuses only on the kernel space instead of the space of all processes. The overhead of periodical detection is small. Fourth, unused space of modules should be checked although the pages are already legitimate to be executed. Finally, our system is more flexible to adjust periods of detection procedures.

```
static struct dentry* adore_proc_lookup(
          struct inode *i,
          struct dentry* d,
          struct namedata* nd)
{
       ...
       task_unlock(current);
       return orig_proc_lookup(i, d, nd);
}
```

```
...
mov  %fs:0xc1416454, %eax
incb 0x330(%eax)
mov 0x8(%esp), %eax
mov %ebp, %ecx
mov %edi, %edx
call *0xd0c0d4d4
...
```

```
...
c1416454   r--  current
d0c0d4d4   r--  c10e6a08
c10e6a08   --x  proc_root_lookup
...
```

Fig. 2. The example binary code snippet(middle), with its associated C snippet(left) and associated output of static analysis

4.2 Diagnosis

To categorize the detected rootkit, we investigate well known typical rootkits according to the intentions that rootkits achieve and the tactics that rootkits adopt. For each typical rootkit, we generate a profile to describe its tactic to achieve its intention as well as modified data structures and objects that we should recover.

Generating the profiles. In our implementation, we generate profiles of typical rootkits manually due to the following reasons. First, rootkits may achieve different intentions together, and understanding the intentions and related tactics of rootkits requires manual effort. Second, data structures and objects that are accessed in the same tactic might subtly vary with the kernel version. Third, rootkits may implement the same tactic in different ways.

Using the profiles. To apply the profiles of known rootkits during diagnosis, we translate the profiles into ones that coordinate with the kernel running in the guest OS monitored by our system. Then the profiles of known rootkits are ready to categorize the detected rootkit. Categorization is done by matching certain characteristic information (collected from the detected rootkit) against the set of pre-generated profiles.

RootkitDet system performs static analysis on the code of rootkit to collect characteristic information. The characteristic information is divided into two groups. One group is the control flow information. Usually, a rootkit calls to some kernel functions to achieve its intentions, which we name *external function call*s. The other group is the global variables and dynamically allocated data structures accessed by the rootkit. In general, to access special data structure maintained by the kernel, the rootkit has to find it starting from a global variable and tracking down according to the relationship among different data structures. A global variable is actually a kernel symbol and usually accessed by its address which is constant. The characteristic information collected through static analysis is binary. RootkitDet system translates the characteristic information according to the meta-data of the kernel. Translated information is then used to categorize the detected rootkit.

We extract the instructions of the rootkit's code as discussed in Appendix 9.1, and suppose that we have figured out the code of the rootkit. Next, we address how we collect characteristic information of the rootkit through static analysis. We focus on external function calls and memory access during static analysis instead of the control flow of the code [7] [16]. Basically, what we need to do is to determine the values of CPU registers during static analysis. We create a static

machine with a special CPU and stack to execute the code of rootkit statically. First, we use a pair <val, flags> to represent the value of a register, in which *val* represents the value while *flags* indicates validation of each byte of *val*. We update the pair instead of the value of registers when we execute instructions. So are the values on the stack. Therefore, we specially tackle instructions accessing the stack. Second, when an instruction involves read of memory other than the stack, we update *val* by the value of the memory and set *flags* by a value indicating *val* totally valid. Finally, some instructions load hard-coded immediate values into registers. In that case, we also update the *flags* of the target register according to the size of immediate value and the instruction type. In consequence, the values of registers that we can determine during static analysis are independent of execution environments. In most cases, we can determine the external function calls and accesses to global variables of the kernel, which we can use to infer the behavior of the suspicious code. Fig. 2 presents an example.

4.3 Recovery

If RootkitDet system successfully categorize the detected rootkit, it attempts to recover the infected kernel according to the profile of the rootkit. Data structures and objects that are modified by the rootkit are described in the profile of the rootkit. Combined with the meta-data of the kernel, *recovery-driven profile* is derived from the profile of the rootkit. Recovery-driven profile describes how to locate the modified data structures and objects and how to recover them.

As mentioned above, we usually know the expected values of the control data, which are the locations of kernel functions. Therefore, the key to recover control data is how to locate it. Data structures and objects maintained by the kernel can always be found tracking down from some global variable. Moreover, the address of global variables are constant and can be found in the meta-data of the kernel. As a result, the recovery-driven profile for control data describes the tracking path from the global variable to the object containing the control data. For example, a rootkit may overwrite the pointers of functions registered with the virtual file system layer by the pseudo random number generator (PRNG) to disable the PRNG [17]. The pointers of functions registered by the PRNG are stored in structures *random_fops* and *urandom_fops*, which are located in the object *devlist*, a list of memory devices that is a global variable. Therefore, the recovery-driven profile for the functions registered by PRNG contains the address of *devlist*, offsets of *random_fops* and *urandom_fops* in *devlist* as well as the real addresses of the functions registered by the PRNG.

Non-control data is different because the original values are either lost forever or not easy to calculate. Moreover, non-control data is different in the way to locate the related data structures or objects. For example, a rootkit hides a process by removing related item from the *pid_hash* table. Then we can't find the process tracking down from the *pid_hash* table. The only way to find the process is tracking down from *init_task* and checking each process whether it is not linked into the *pid_hash* table. As a result, the recovery-driven profile for non-control data describes how to restore the broken links or resolve violations

of invariants as well as the tracking path from the global variable to the object containing the non-control data. If the original value of a non-control data are lost forever, we can not recover it. For example, we can not recover the entropy pool of PRNG if it is zeroed by a rootkit [17].

4.4 Implementation

In our implementation, we use *qemu-kvm-1.2.0* for creating instances of the guest OS, and compile *linux-2.6.32.60* for the guest kernel. Fig. 3 shows the internal components of the prototype of RootkitDet system.

Detector. We integrated the detector into qemu-kvm because the guest OSes are running as user processes on the host OS and the integration reduces inter-process communications. Moreover, we implement the inspector as part of the detector.

The detector consists of five components: inspector, data container, hash component, control component and communication component. Inspector is responsible for reading the registers and memory of the VM. Data container component constructs the necessary semantic data structures from the raw data of the VM's memory given by inspector according to the profile, and also stores data coming from the conductor through communication and control components. Hash components is used for calculating the current hash values for the kernel and modules' code. The communication component takes care of all of the communication with the conductor. The control component receives commands through the communication component from the conductor, then executes the commands and sends the response back to the conductor.

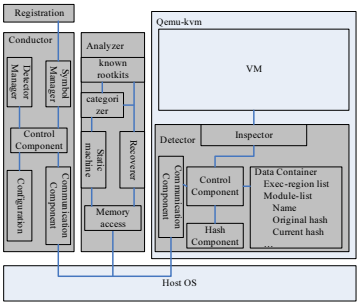

Fig. 3. Internal Components of RootkitDet

For the sake of flexibility, we implement the detector as a command-driven object, which is an I/O handler of qemu-kvm. It doesn't do anything until it receives a command from the conductor, and it goes back to the initial state as soon as it finishes that command. If the conductor doesn't send any commands to the detector, the VM runs the same as if there is no detector. Therefore, it is convenient for us to turn off/on this security feature of the VM when necessary.

Conductor. The conductor is a daemon process that periodically schedules detectors for monitored guest OSes and starts up the analyzer when rootkits are detected. It is also responsible for generation of original checksums of registered modules when requested.

To generate the original checksums of kernel modules, the conductor performs the same relocation work as the guest kernel does. The correct relocation work of a module depends on the following information: the original object file, the relocation address, the addresses of the used kernel symbols and the addresses of the used symbols of other modules. The conductor acquires the original ob-

ject file of a module from the registration component and obtains its relocation address from the detector. The conductor can figure out the address of a kernel symbol by referring to the meta-data of the kernel. We create a database storing the relative addresses of symbols exported by registered modules. The conductor can calculate the absolute address of a symbol exported by a module by looking it up in the database and adding it up to the relocation address of the module. To resolve the potential dependency among modules during the relocation work, the conductor calculates original checksums after collecting the relocation addresses of all loaded modules from the detector. Consequently, the conductor can generate original checksums for all of the loaded modules and send them back to the detector.

We generate the original checksum of the kernel code in advance because the kernel code is constant and never changes after it starts up. We take the original checksum of the kernel code as part of the meta-data of the kernel.

Analyzer. The analyzer is actually an independent program in our prototype system, and is started up by the conductor when the detector reports that a rootkit is detected. Therefore, we save the resources consumed by the analyzer if no rootkits are detected, which is tenable in most times.

Once the detector reports that a rootkit is detected, the conductor starts up the analyzer immediately. Analyzer collects the characteristic information through static analysis, translates combining the meta-data of the kernel, and attempts to categorize the rootkit according to profiles of known rootkits. If it successfully categorizes the rootkit, it recovers modified data structures and objects according to the recovery-driven profile of the rootkit.

5 Evaluation of RootkitDet System

In this section, we present the evaluation results of our RootkitDet prototype. Our evaluation has two goals. The first goal is to evaluate RootkitDet's effectiveness for detecting kernel-level rootkits that compromise the code integrity of the OS kernel and recovering modified data to eliminate the effect of rootkits. The second goal is to measure the overhead introduced to guest OSes and extra resources consumed by RootkitDet.

All experiments are conducted on Dell PowerEdge M610 Server with a 2.40GHz Intel Xeon E5645 and 6GB memory. The hypervisor is qemu-kvm-1.2.0. The host OS is Ubuntu-12.04. We used Debian-squeeze with kernel version 2.6.32 as our guest OS. The detector is integrated into qemu-kvm, and thus runs with the guest OS. The conductor ran on another computer as a user process. They communicated with each other through TCP connections.

5.1 Effectiveness

To evaluate the effectiveness of RootkitDet system for detecting kernel-level rootkits, we install four representative rootkits in the guest OS monitored by our system. As shown in Table 1, different detection procedures detect the rootkits that hide the code in different regions.

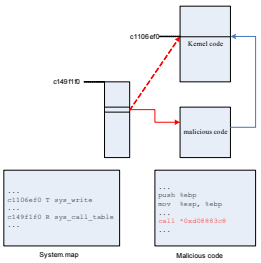

Fig. 4. hksc: hooking sys_write

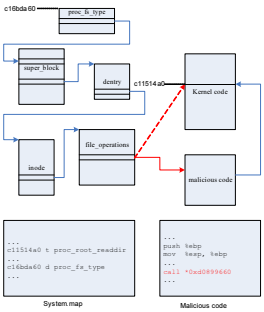

Fig. 5. hkproc: hooking proc filesystem

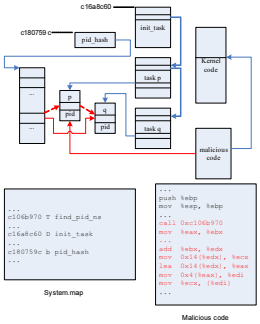

Fig. 6. hidepc: manipulating pid hash table

Adore-ng [18] is implemented as kernel module and an extra executable region appears when it is installed. Therefore, it is detected by procedure 1. **Enyelkm** [19] is also implemented as a kernel module and thus detected by procedure 1. In addition, it also hijacks the control flow of the kernel by modifying the system call dispatch routine to intercept several system calls, and thus it is also detected by procedure 3. Despite of the probability that a rootkit's code resides in dynamically allocated executable regions or unused space of modules, we don't find one in wild. We implement **Icmp-cmd** and **Icmp-cmd_v2**, which execute commands specified by crafted ICMP packets, to evaluate the RootkitDet's effectiveness in detecting such rootkits. The code of them resides in a dynamically allocated executable region and unused space of a module, and thus they are detected by procedure 1 and 2 respectively.

To evaluate the effectiveness of RootkitDet system for eliminating the effect of rootkits, we develop 3 rootkits, hksc, hkproc and hidepc, which adopt different tactics to hide a specific user process from the guest system administrators, and install them in the guest OS monitored by our system. After detection of them, our system successfully categorizes them and performs recovery to reveal the hidden process.

Table 1. Rootkit detection

Rootkit	Method to insert code	DP
adore-ng	module	1
enyelkm	module and substitution	1, 3
icmp-cmd	executable region	1
icmp-cmd_v2	unused space	2

Fig. 4 shows a rootkit hijacking *sys_write* system call to hide a specific process by tampering with what is displayed to the administrators of guest OSes. We recover the modified system call table to eliminate the effect of this rootkit. Fig. 5 shows a rootkit hooking the function pointer *proc_root_readdir* to hide a specific process by removing related *pid* entry in the *proc* file system. We find the hooked function pointer by tracking down from *proc_fs_type*, which is a global variable, and correct it with the real location of kernel function *proc_root_readdir*. Fig. 6 shows a rootkit hiding a specific process by removing related entry in the pid

Table 2. Application-level benchmarks of overhead to guest OSes

Benchmark	W/o Performance	W/i Performance	Relative Performance
Dhrystone	6040580.1 lps	6045164.7 lps	1.001X
Whetstone	630.6 MIPS	629.9 MIPS	0.999X
Lmbench(pipe bandwidth)	3843.2 MB/s	3810.3 MB/s	0.991X
Apache Bench(throughput)	569.95 KB/s	568.67 KB/s	0.998X
Kernel decompression	21.343 s	21.529 s	0.991X
Kernel build	1300.4 s	1292.9 s	1.001X

Table 3. Time of detection and recovery

Rootkit	Code size(byte)	detection time(ms)	analysis time(ms)	recovery time(ms)
hksc	407	< 1	14.6	3.7
hkproc	978	< 1	44.6	7.7
hidepc	565	< 1	29.1	204.8

hash table. We first find the *task_struct* of the hidden process by tracking down from *init_task*, and then relink it into the pid hash table to reveal the hidden process.

5.2 Overhead to the Guest OSes

To measure the performance cost introduced by our system to the guest OS, we run a set of application benchmarks to compare the performance of a guest OS that enables the detector with the one that does not. The application benchmarks and their configuration are presented as follows: 1) Dhrystone 2 of the UnixBench suite using register variables. 2) Double-precision whetstone of the UnixBench. 3) Pipe bandwidth of Lmbench measuring the performance of IPC interface provided by the kernel. 4) Apache Bench configured to issue 10,000 http requests (177B HTML file) through 1 client. 5) Kernel source code decompression using command `tar xjf` to extract the compressed tarball file of Linux 2.6.32 kernel. 6) Building a Linux 2.6.32 kernel.

We run detection procedure 1 in each second, procedure 2 in 4 seconds, procedure 3 in 16 seconds because of different complexity of them. Table 2 presents the results of these application level benchmarks. The second column shows the performance of the guest OS which doesn't enable the detector, while the third column shows the performance of the guest OS that enables the detector. The last column presents the relative performance. To reduce the effect of random factors, we run each benchmarks 10 times, and present the average results in the table. From table 2, we can see that the relative performance of the guest OS that enables the detector is above 0.99X, on both CPU intensive jobs and I/O intensive jobs. In other words, the performance cost is tolerable to most tasks.

Besides application level benchmarks, we also perform a micro-benchmark test on the detector. In the experiment result, detection procedure 1 costs the least

time, which is 189 μs; detection procedure 2 costs more time, which is 713μs; detection procedure 3 costs the most time, which is 47139 μs.

5.3 Performance

To measure the scalability of our system, we run multiple VM instances that enable the detector at the same time and measure the network bandwidth and CPU resources consumed by our system, i.e. the conductor. Fig. 7 shows the peak and average network bandwidths (input and output) consumed by the conductor are linear to the quantity of the VM instances. In addition, our system consumes 3% CPU cycles for every 10 guest OSes. As a result, our is scalable and efficient in the cloud environment.

To measure the efficiency of our system, we measure the time of detection and recovery against the 3 rootkits that hide a specific user process in the guest OS. Table 3 shows the evaluation result. The time of detection is less than 1 ms because the rootkits are implemented as modules which are detected by detection procedure 1. The time of analysis depends on the code size and pages of memory accessed by the code. The time of recovery mainly depends on the complexity of recovery.

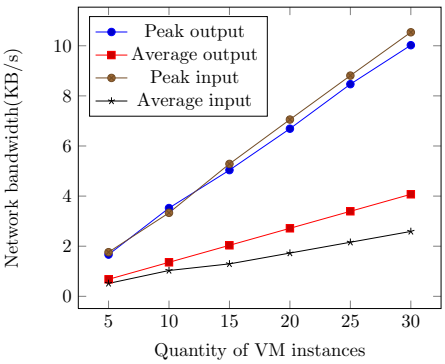

Fig. 7. Network bandwidth consumed by Conductor

6 Related Work

Kernel-level rootkits have been distributed in the underground hacker community for a long time [20]. In order to relieve the threat of kernel-level rootkits, many techniques or architectures are proposed. Most recently techniques or architectures leverages the security benefit of the hypervisor. They can be divided into 4 classes: (1) Detecting data modifications or violations of invariants in the kernel; (2) Preventing the installation of the rootkits by performing analysis on the code being loaded into the kernel space; (3) Defending kernel rootkits by cooperating with anti-malware software; (4) Protecting the kernel by restoring infected kernels to healthy state.

In class 1, SBCFI [5] detects persistent kernel control-flow attacks by identifying function pointers in kernel data structures to the kernel and modules' code. HookSafe [6] protects thousands of kernel hooks in a guest OS from being hijacked. Gibraltar *et al* [3] detects kernel rootkits by identifying data invariants in the kernel. The work of Petroni Jr *et al* [4] focuses on semantic integrity violations in kernel dynamic data. While these works focus on control/non-control data in the kernel, our system focuses on the code inserted into kernel space and attempts to perform recovery.

In class 2, the work of C. Kruegel *et al* [7] performs static analysis on the module that is being loaded and prevents it if it resembles the behavior of rootkits. Liveware [21] protects the guest OS kernel code and critical data structures from being modified. Our system improves by detecting the code added into the kernel space that is not in the form of a module. SecVisor [8], a tiny hypervisor that enforces page-level protection of the memory used by the code of the kernel and modules, prevents the installation of the kernel rootkits by ensuring the code integrity of guest OS kernel. NICKLE [9] protects the code integrity in the guest OS kernel by transparently routing guest kernel instruction fetches to shadow memory which contains authenticated code and is protected from write-access. However, they are not easy to be adopted in the cloud platforms based on different hypervisors because they require special features of the hypervisor.

VMWatcher [11] detects malware by providing semantic view of the guest OS to anti-malware software, and Lares [10] presents an architecture that gives the security tools the ability to do active monitoring. While they are cooperated with external security tools or anti-malware software, our system can defend rootkits alone and monitor more VMs with less effort.

VICI Agent [12], which belongs to class 4, applies different repair techniques to restore the infected kernel to healthy state after detecting kernel-modifying rootkit infections. However, it can be defeated by novel rootkits that insert new control-data in the kernel space instead of modifying existing control-data, such as Icmp-cmd mentioned in Section 5.

7 Discussions and Limitations

RootkitDet system is not perfect because of the following reasons. First, it cannot detect rootkits that are erased immediately after executed or that have no specific code in the kernel space, like return-oriented rootkits [22]. Second, it may not detect all of the code of a rootkit if the rootkit hides part of its code by switching NX-bit of the corresponding pages, therefore our system may lose some characteristic information of the rootkit during analysis. Third, it cannot prevent the installation of the kernel-level rootkits although it detects rootkits and recovers the kernel if possible. Fourth, it cannot certainly recover all modifications made by the rootkits, especially when categorization of the rootkits fails. Finally, the generation of instinct information of rootkits are not automatic. However, RootkitDet system are still useful and flexible in practice. In addition, it can perform quickly detection of kernel-level rootkits by only issuing detection procedure 1 and 2 because almost all of kernel-level rootkits in the wild introduce extra code into the kernel space and fewer and fewer of them modify the code of the kernel or modules. RootkitDet system provides the characteristic information of unknown rootkits to assist further investigation.

In future work, we can focus on the analysis and recovery of novel and unknown rootkits and automatic generation of rootkits' instinct information.

8 Conclusions

In this paper, we present the design, implementation and evaluation of the RootkitDet system, an end-to-end defense against kernel-level rootkit, which is efficient and practical in the cloud environment. RootkitDet system detects rootkits that insert code into kernel space of guest OSes, diagnoses the detected rootkit to precisely locate modifications caused by the rootkit, and attempts to recover the modifications. Our evaluation experiments show that the RootkitDet system can effectively detect kernel-level rootkits and reverse modifications if the rootkits are categorized successfully. In addition, the performance cost introduced to the guest OSes by our system is less than 1%, and the complexity of our system is linear with the quantity of the VM instances being monitored, which is acceptable in the cloud environment.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Stefano Paraboschi for their constructive feedback to improve the paper. Lingchen Zhang was supported by ARO grant W911NF-12-1-0055. Sachin Shetty was supported by ARO grant W911NF-12-1-0055, NSF grant HRD-1137466, DHS grants 2011-ST-062-0000046 and 2010-ST-062-0000041. Peng Liu was supported by AFOSR W911NF1210055, ARO W911NF-09-1-0525, NSF CNS-1223710, and ARO W911NF-13-1-0421. Jiwu Jing was partially supported by the National 973 Program of China under award No.2014CB340603 and the National 863 Program of China under award No.2013AA01A214.

References

1. McAfee: Rootkits, Part 1 of 3: A Growing Threat. white paper (April 2006)
2. McAfee: 2010 Threat Predictions. white paper, McAfee AVERT Labs (December 2009)
3. Baliga, A., Ganapathy, V., Iftode, L.: Detecting kernel-level rootkits using data structure invariants. IEEE Transactions on Dependable and Secure Computing 8(5), 670–684 (2011)
4. Petroni Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proceedings of the 15th USENIX Security Symposium, pp. 289–304 (2006)
5. Petroni Jr., N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 103–115. ACM (2007)
6. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 545–554. ACM (2009)
7. Kruegel, C., Robertson, W., Vigna, G.: Detecting kernel-level rootkits through binary analysis. In: 20th Annual Computer Security Applications Conference 2004, pp. 91–100. IEEE (2004)
8. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 335–350. ACM (2007)

9. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
10. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: IEEE Symposium on Security and Privacy, SP 2008, pp. 233–247. IEEE (2008)
11. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 128–138. ACM (2007)
12. Fraser, T., Evenson, M.R., Arbaugh, W.A.: Vici-virtual machine introspection for cognitive immunity. In: Annual Computer Security Applications Conference, ACSAC 2008, pp. 87–96. IEEE (2008)
13. Kemerlis, V.P., Portokalidis, G., Keromytis, A.D.: kguard: lightweight kernel protection against return-to-user attacks. In: USENIX Security Symposium (2012)
14. Linux-KVM: Linux-KVM, `http://www.linux-kvm.org/page/Main_Page`
15. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: Proceedings of the 17th Conference on Security Symposium, pp. 243–258 (2008)
16. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, S&P 2001, pp. 156–168. IEEE (2001)
17. Baliga, A., Kamat, P., Iftode, L.: Lurking in the shadows: Identifying systemic threats to kernel data. In: IEEE Symposium on Security and Privacy, SP 2007, pp. 246–251. IEEE (2007)
18. Stealth: Announcing full functional adore-ng rootkit for 2.6 kernel, `http://lwn.net/Articles/75991/`
19. eNYe Sec: eNYeLKM v1.1, `http://www.enye-sec.org/en/tags/enye-lkm/`
20. Halflife: Abuse of the Linux-kernel for Fun and Profit. Phrack Magazine 5(50) (April 1997)
21. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proc. Network and Distributed Systems Security Symposium (2003)
22. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX Security Symposium, pp. 383–398 (2009)

9 Appendix

9.1 Extracting Instructions

When a rootkit is detected by our system, we get a suspicious executable region where the code of the rootkit locates. To analyze the rootkit's code, we need extract instructions of the rootkit's code from the executable region first. The executable region is page-aligned and we don't know the exact location of the rootkit's code. Moreover, it is non-trivial task to distinguish code from data on X86 platforms. Therefore, it is difficult to find out the rootkit's code in the detected executable regions.

We notice that the code of rootkits usually comprises a set of functions and the instructions are continuous unless a jump instruction occurs. Several successive instructions of a function compose an instruction block ending with a branch, jump, call or return instruction. If an instruction block ends with a return instruction, no more blocks follows it in logic. If a block ends with a call instruction, the next block starts right behind the call instruction. If a block ends with a jump instruction, the address of the following block can be calculated from the address of the jump instruction and its content. If a block ends with a branch instruction, two blocks follow it in logic: one is just behind it and the starting address of the other can be calculated from the address of the branch instruction and its content.

Consequently, we can figure out all of the instruction blocks of a function as long as we find the first block. That is to say, we should find the first instruction of a function. We search the first instruction from the first byte of the executable region which starts at the lowest address if multiple regions exist. If the first instruction starts here, we can figure out the whole function. Otherwise, we should encounter an illegal instruction in all probability during the process of extracting instruction blocks. If it is not the first instruction, we march on by one byte. We repeat this step until we figure out the first instruction.