IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON INTERNET-OF-THINGS (IOT) BIG DATA TRUST MANAGEMENT

Received September 8, 2017, accepted September 26, 2017, date of publication October 2, 2017,
date of current version November 7, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2758356

CloudVMI: A Cloud-Oriented Writable Virtual Machine Introspection

WEIZHONG QIANG, GONGPING XU, WEIQI DAI, DEQING ZOU, AND HAI JIN, (Senior Member, IEEE)

Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Corresponding author: Weiqi Dai (wqdai@hust.edu.cn)

This work was supported in part by the National Basic Research Program of China, 973 Program, under Grant 2014CB340600 and in part by the National Natural Science Foundation of China under Grant 61772221.

ABSTRACT IoT generates considerable amounts of data, which often requires leveraging cloud computing to effectively scale the costs of transferring and computing these data. The concern regarding cloud security is more severe because many devices are connected to the cloud. It is important to automatically monitor and control these resources and services to efficiently and securely deliver cloud computing. The writable virtual machine introspection (VMI) technique can not only detect the runtime state of a guest VM from the outside but also update the state from the outside without any need for administrator efforts. Thus, the writable VMI technique can provide the benefit of high automation, which is helpful for automated cloud management. However, the existing writable VMI technique produces high overhead, fails to monitor the VMs distributed on different host nodes, and fails to monitor multiple VMs with heterogeneous guest OSes within a cloud; therefore, it cannot be applied for automated and centralized cloud management. In this paper, we present CloudVMI, which is a writable and cross-node monitoring VMI framework that can overcome the aforementioned issues. CloudVMI solves the semantic gap problem by redirecting the critical execution of system calls issued by the VMI program into the monitored VM. It has strong practicability by allowing one introspection program to inspect heterogeneous guest OSes and to monitor VMs distributed on remote host nodes. Thus, CloudVMI can be directly applied for automated and centralized cloud management. Moreover, we implement some defensive measures to secure CloudVMI itself. To highlight the writable capability and practical usefulness of CloudVMI, we implement four applications based on CloudVMI. CloudVMI is designed, implemented, and systematically evaluated. The experimental results demonstrate that CloudVMI is effective and practical for cloud management and that its performance overhead is acceptable compared with existing VMI systems.

INDEX TERMS Virtual machine introspection, cloud management, security monitoring.

I. INTRODUCTION

Cloud computing and IoT are tightly coupled because cloud computing can offer several advantages to IoT, such as on-demand, broad network access, resource pooling, rapid elasticity, and measured services. We observe considerable growth in cloud-based services for devices, and much of this growth is from Amazon Web Services, Google, and Microsoft, such as the Amazon Echo service that uses a cloud computing service to handle voice control for connected devices. Although IoT technology is creating tremendous benefits, new security risks are also introduced to the front-end devices and to the back-end cloud services.

Virtual machines (VMs) are an essential building block of IaaS (Infrastructure as a Service) in the cloud, through which primary cloud resources and services are provided to the cloud users. Thus, effectively managing the VMs within a cloud is one of the key requirements for the reliable delivery of cloud computing [1]. The virtual machine introspection (VMI) [2] technique was proposed to monitor

the state of a guest VM from the outside, which brings many benefits, such as strong isolation, higher-level privileges, high stealthiness, and so forth. Thus, the VMI technique has been used for intrusion detection [3]–[5], malware analysis [6], and memory forensics [7], and it is a critical and indispensable component for cloud management [8].

As we know, the semantic gap [9] problem is the key challenge of VMI, which is the problem of how to obtain the high-level OS semantics (e.g., data structure and process) from the low-level bits and bytes in physical memory. To date, many solutions have been proposed to solve the semantic gap problem, such as LibVMI [10], Virtuoso [11], and VMST [12], among others. In particular, the concept of writable VMI was first proposed in Exterior [13], in which the semantic gap is bridged by redirecting key kernel data accesses of the introspection program into the memory of the monitored VM. The writable VMI can not only inspect the state of the guest VM but also update the state from the outside without any need for administrator efforts.

Thus, the writable VMI not only shares all of the benefits of the read-only VMI technique, such as strong isolation and higher-level privilege, but also brings another unique benefit of high automation [14]. This means that it can automatically respond to events in the guest OS without the requirement of any user privileges of the guest OS. For instance, when an intrusion is detected inside a guest OS, it should be responded to in a timely manner to ensure the security of the guest OS. The current solutions are normally to deploy an automated response program with root privilege inside the guest OS to respond to the intrusion. However, any in-guest response program can be disabled by attackers because they run at the same privilege level. Conversely, the writable VMI technique can automatically respond to events in the guest OS from outside of this guest OS without any in-guest response program and root privileges.

Therefore, writable VMI is in principle suitable for automated VM management. However, the current solution of writable VMI in Exterior [13] cannot be applied to the complex cloud environment due to the limitations of *overhead* and *practical usefulness*. In terms of overhead, Exterior introduces 23× overhead on average for the introspection tools compared with the same tools running inside the guest OS, which causes a substantially longer response time for introspection, thus making it unable to achieve real-time monitoring. Without real-time monitoring, Exterior will most likely fail to respond to short-lived events, such as ephemeral malicious processes.

In terms of practical usefulness, on the one hand, modern cloud applications typically span across multiple VMs, thus requiring a distributed application-level monitoring view across system boundaries [15]. However, Exterior cannot achieve cross-node monitoring, which means that it cannot monitor VMs that run on different host machines. Thus, Exterior cannot provide a more accurate and holistic monitoring view of VMs and applications to offer centralized cloud management. On the other hand, the monitoring program executed in Exterior cannot monitor the monitored VMs that run different guest OSes from the monitoring VM, which means that the guest OS of the monitoring VM must be exactly the same as the guest OS of the monitored VM. However, a cloud deployment typically contains large amounts of VMs that run heterogeneous OSes. Therefore, in Exterior for example, in the case that a host runs ten guest VMs with different OSes, the administrator has to create ten monitoring VMs to monitor these guest VMs, which is too impractical to be applied to cloud management.

To address the aforementioned concerns, we introduce CloudVMI, a writable and cross-node monitoring framework for the cloud. CloudVMI solves the semantic gap problem by using a system call (syscall for short) redirection mechanism, which redirects the execution of key syscalls issued by the introspection process into a process inside the monitored guest VM. Inside the monitored guest VM, the code of the guest OS will be reused to executed redirected syscall as usual. Since the syscall interface in different OSes is backward compatible, CloudVMI supports the introspection program in inspecting VMs with different guest OSes, which greatly improves the generality of VMI tools. Moreover, CloudVMI uses network communication to transfer the redirection syscall request; thus, it can achieve cross-node monitoring. However, CloudVMI is architecturally not sufficiently secure because it faces some security threats due to the reuse of the code in the untrusted guest OS. To ensure the security of CloudVMI, we analyze all potential security threats and implement some protective measures to defend against them. To further highlight the capabilities of CloudVMI, we develop four applications based on its prototype for automated cloud management.

In summary, we make the following contributions in this paper:

- We develop CloudVMI, which is a writable and cross-node virtual machine introspection framework, by redirecting the key syscall execution of the monitoring program into the process inside the monitored guest VM. CloudVMI has two merits compared with existing writable VMI techniques: lower overhead and practical usefulness, including strong generality and the capability of cross-node monitoring.
- We analyze all potential security threats in CloudVMI, and we implement some effective protection measures to defend against these security threats based on memory protection and consistency checking.
- We build four practical applications based on CloudVMI to demonstrate automated cloud management and cloud analytics, including rootkit detection and recovery, network topology discovery, real-time resource monitoring, and virus file scanning.
- We have implemented the entire system prototype based on the Linux kernel and KVM virtualization platform. We have performed an empirical evaluation based on legacy Linux utilities. The experimental results

demonstrate that CloudVMI is effective and practical and that it only introduces a negligible performance overhead.

The remainder of this paper is organized as follows. Section II presents related work. Section III describes the system overview of CloudVMI. Section IV elaborates the detailed design of CloudVMI. Section V describes the applications implemented to demonstrate CloudVMI's capability. Section VI presents the evaluation results. Section VII discusses the limitations of our work. Section VIII presents the conclusion.

II. RELATED WORK

The concept of VMI was first proposed in the Livewire [4] system, which can monitor the memory state and register state of a guest OS from the outside. XenAccess [16] was developed as the monitoring library for the Xen hypervisor, which can be used to monitor the memory state and disk state of guest VMs with Linux and Windows kernels. LibVMI [10] was designed to make the VMI tools work across multiple virtualization platforms (e.g., Xen and KVM) to monitor guest VMs with Linux and Window kernels. However, LibVMI fails to support the functionality of disk introspection; thus, introspection tools, such as `ls` and `df`, cannot be provided based on it. All of these works require detailed pre-knowledge (i.e., memory layout) of a guest OS before introspection, which means that updating or patching a guest OS will make existing monitoring be incompatible.

Rather than requiring the detailed pre-knowledge of a guest OS, Virtuoso [11] captures the internal syscalls of a guest OS or instructions of the application in a guest OS to generate a VMI program that can run outside of the monitored VM to realize the same function as internal procedures in the guest OS. Process out-grafting (Pog) [17] migrates a suspect process from a guest VM to a trusted VM, in which a legacy security tool (e.g., `strace`) is used to monitor the behavior of the migrated process. Process implanting (PI) [18] injects a process execution into a monitored VM, in which the execution result of the injected process will be fed back to a trusted VM. VMST [12] and Exterior [13] redirect key kernel data accesses to the kernel memory of the monitored VM, by which the legacy utilities, such as `ps` and `lsmod`, can be directly used as the introspection tools without any modifications. Specifically, Exterior first proposes and implements a writable VMI framework. All of the above works have bridged the semantic gap without any pre-knowledge of guest OSes, but all of them introduce significant overhead and cannot monitor guest VMs that run on different host machines. Moreover, certain works lack generality.

Recently, two systems, HYPERSHELL [19] and ShadowContext [20], have been proposed to bridge the semantic gap via the syscall redirection technique. Both of these systems introduce less overhead and can monitor VMs with heterogeneous OSes. However, HYPERSHELL is only used for single VM management from the outside in an automated manner, and it cannot be used for any security-related VMI applications due to the lack of any security protection measures. Conversely, ShadowContext is secure enough to defend against some real-world attacks based on some defensive measures. However, it can provide neither the capability of writable VMI nor disk introspection. Moreover, neither of these systems can monitor VMs distributed on different host machines.

Compared to current VMI techniques ([4], [10]–[13], [16]–[20]), CloudVMI is more suitable for automated cloud management due to its writable capability and practical usefulness.

III. SYSTEM OVERVIEW

A. MAIN IDEA

The primary goal of our work is to offer a writable and cross-node VMI framework with lower overhead for automated and centralized cloud management. With CloudVMI, we achieve this goal by redirecting the execution of the key syscall issued by the introspection process into the monitored VM, which can solve the semantic gap problem. Since VMM has the highest privilege that controls all virtualized hardware of the guest VM, it can force a guest OS to execute any syscall when the VM exits.

With syscall redirection, an introspection process will be initialized in a monitoring host machine, with part of its syscall execution being redirected into the monitored VM, which can be located on the same host as the monitoring host or on a different host. Thus, the introspection process will be executed across two process contexts, with one in the monitored VM and the other in the monitoring host machine. The introspection process itself will only execute those syscalls that will not contribute to the inspection of a guest state or the update of a guest state. In contrast, the syscalls that are critical to the inspection and update of a guest state will be redirected into the monitored VM.

In addition, since the syscall interface in different OSes is backward compatible unless it is intentionally made different [21], the VMI tool executed in CloudVMI can monitor heterogeneous OSes as long as this VMI tool is compatible with these OSes. Therefore, CloudVMI has high generality of monitoring, which greatly improves the practical usefulness of VMI tools. Furthermore, in the Linux system, the legacy native utilities (e.g., `ps, lsmod, uname`) are implemented based on a series of syscalls. Thus, these legacy programs can be reused as the introspection tools executed in CloudVMI without any modifications. This reuse of legacy programs will greatly improve the efficiency of monitoring and reduce the difficulty of developing VMI programs.

Figure 1 shows a holistic monitoring architecture to illustrate how CloudVMI achieves the aforementioned goal of cross-node monitoring. As shown in Figure 1, there are a monitoring host machine with the *monitoring controller* module and the *global hash table* and two remote host machines that host the monitored VM. Inside the monitoring host

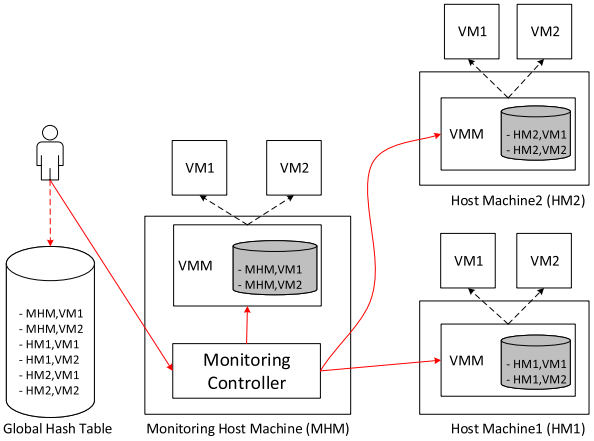

FIGURE 1. Cross-node monitoring architecture for cloud.

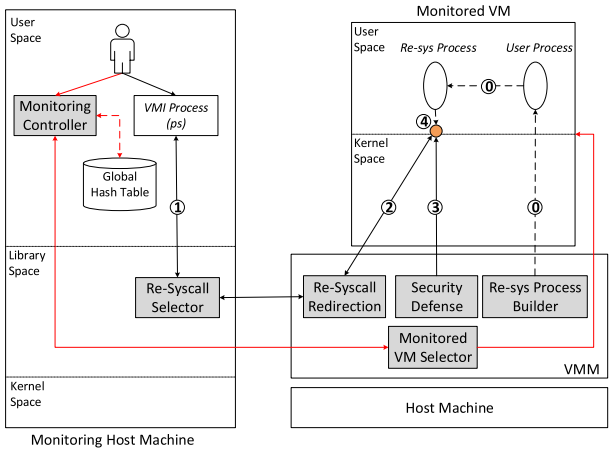

FIGURE 2. System architecture of CloudVMI.

machine (MHM), the administrator can run a VMI program to monitor any target VM even though this VM runs on a remote host machine (e.g., HM1 and HM2).

As shown in Figure 1, MHM will build a communication channel with any monitored host machine via the *monitoring controller* to transfer the syscall redirection message and corresponding parameters. For instance, when the administrator needs to monitor guest VM1 that runs on HM1, a communication channel will be built between MHM and HM1. Then, through the *global hash table* that contains the location information about guest VMs in the cloud, a VM can be selected as the monitored VM. Finally, the administrator can run an introspection program to monitor this remote VM. Therefore, with the *monitoring controller* in MHM, the administrator can monitor those VMs that are distributed on remote host machines, thereby achieving cross-node monitoring.

Assumption: In the CloudVMI architecture, we only focus on the x86 architecture and Linux OS with kernel-based virtual machine (KVM) hypervisor. Additionally, we assume that the monitoring host machine, the host machines except the VMs hosted by them, and the VMM are trusted. To ensure correct syscall execution in different versions of Linux kernels, we further assume that the syscall interface cannot be intentionally made different, which means that all guest OSes have a compatible syscall interface. In addition, our architecture needs to rely on a process inside the monitored VM to execute redirected syscalls, and some code will be injected into the monitored VM from the VMM. Thus, we assume that cloud tenants will not reject the code injection from the VMM.

B. SYSTEM ARCHITECTURE

The detailed architecture of CloudVMI is shown in Figure 2. CloudVMI consists of six components: monitoring controller, re-syscall selector, monitored VM selector, re-syscall redirection, re-sys process builder, and security defense. All the components are not inside the guest OS; thus, there is no need to modify the code of the monitored guest OS. Consequently, CloudVMI achieves full transparency [22].

The *monitoring controller* and *re-syscall selector* are located in the monitoring host. The former controls the monitoring of VMs within the cloud such that CloudVMI can achieve the goal of cross-node monitoring in the cloud. The latter intercepts each syscall of an introspection process and redirects certain important syscalls to the target monitored VM determined by the *monitoring controller*.

The remainder of these components are located in the KVM on the monitored host. The *monitored VM selector* records and synchronizes all VMs' information in each monitored host to the global hash table inside the monitoring host, and it will build a communication channel with the monitoring host when the *monitoring controller* has selected a target VM that is hosted in it. The *re-sys process builder* selects a process inside the guest OS as a re-sys process to execute a redirected syscall and feed back the execution results to the introspection process. *re-syscall redirection* prepares related redirected syscall parameters and injects a syscall into the re-sys process to start syscall execution. *Security defense* defends against all security threats to ensure the security of syscall execution and the reliability of the introspection results. The detailed description of these components will be provided in Section IV.

When the cloud administrator wants to monitor a guest VM that runs on the remote host machine, before a VMI starts to run on the monitoring host, the administrator needs to leverage the *re-sys process builder* to select and initialize the re-sys process inside each monitored VM. Then, the administrator can determine any target VM through the *monitoring controller* and the *monitored VM selector*. Subsequently, a communication channel between the monitoring host and monitored host will be built to transfer the syscall redirection request and syscall parameters. Finally, the administrator can run an introspection program in the monitoring host to monitor the target VM. The execution of the redirected syscall will be accomplished in the re-sys process that has been selected inside the target VM. There are five steps involved in a syscall redirection:

Step 1 When a syscall of the introspection process enters the library space of the monitoring host OS, the *re-syscall selector* will intercept this syscall. If this syscall does not need to be redirected, it will be directly trapped in the kernel space of the monitoring host OS. Otherwise, the *re-syscall selector* transfers the syscall number and the related parameters to the remote host through the communication channel built by the *monitoring controller*. Then, the introspection process suspends execution in the monitoring host until the redirected syscall finishes execution in the target VM.

Step 2 Inside the remote VMM, *re-syscall redirection* listens for the syscall redirection request. Once the *re-syscall redirection* receives a syscall redirection request from the monitoring host, including the corresponding syscall number and parameters, it will prepare the syscall data for the re-sys process inside the monitored VM by assigning these parameters to the corresponding CPU registers.

Step 3 Before the re-sys process starts to execute in the target VM, *security defense* secures the execution of the re-sys process. Then, the re-sys process is trapped in the kernel space of the monitored guest OS to execute the redirected syscall based on the transferred syscall number and parameters.

Step 4 During the execution of the re-sys process, if any kernel state update occurs to the guest OS, the re-sys process will directly update its kernel memory. If any user-space state update occurs, *re-syscall redirection* will transfer this updated syscall data back to the monitoring host based on the built communication channel.

Step 5 After syscall finishes execution in the monitored VM, *re-syscall redirection* transfers the message and updated syscall data back to the monitoring host to resume the execution of the introspection process in the monitoring host. Then, the introspection process continues its execution in the monitoring host OS.

TABLE 1. Security threats.

Surface	Attack Vector	Attack Method
User Space	Code	Modify the injected code
	Data	Tamper with the buffer
	Control Flow	Use `ptrace` to hook syscalls
Kernel Space	Code	Break the kernel image
	Data	DKSM and DKOM
	Control Flow	Hook the syscall

C. THREAT MODEL

We assume that all host OSes and VMMs in the cloud are trusted; however, the monitored guest VM is untrusted, which will face a variety of real-world security threats. Thus, the re-sys process executed in an untrusted VM will be insecure, which will directly affect the security of CloudVMI. Table 1 summarizes the attack surface in our threat model. As shown in this table, all security threats can mainly be divided into the two following categories.

1) USER SPACE

The re-sys process that runs inside the untrusted VM faces a variety of security threats in the user space of the target VM. First, some communication code needs to be added into the code segment of securityTable for intercepting the exit point of the redirected syscall execution. Thus, user-space malwares could modify this code to subvert the normal execution of the re-sys process, thereby tampering with CloudVMI. Second, some memory buffers will be allocated to store the syscall parameters; thus, the malwares can manipulate the data in this buffer to tamper with the results of introspection by overwriting the wrong data back into the buffer. Finally, the malwares can use the `ptrace` syscall to register a callback to intercept the control flow of the re-sys process, thereby subverting the introspection results.

2) KERNEL SPACE

In modern Linux OSes, all syscalls are handled inside the kernel space. Thus, when a syscall executes in an untrusted VM, it will suffer the subversion of kernel integrity, which includes the code, control flow, and data structure. First, malicious code can be injected by kernel rootkits into the kernel image of the guest OS [23], which can subvert the correct execution of the re-sys process. Second, most rootkits (approximately 96% [24]) hijack the syscalls' control flow of existing processes by modifying the interrupt descriptor table (IDT), syscall table or other kernel function pointers to implement their malicious code for hiding vicious activities. Finally, some kernel rootkits can directly manipulate kernel data structures (DKSM) [25] and kernel objects (DKOM) [26] to subvert introspection, under which the introspection results will be incorrect.

IV. SYSTEM DESIGN

In this section, we present the detailed design of the components in CloudVMI that were introduced in Section III.

A. TARGET VM SELECTION

1) TARGET VM SELECTOR

To select any VM as the monitored target VM, we have to know the destination address of the monitored host and monitored VM. Thus, inside each VMM, we create a local hash table to record such information of all VMs, including the destination addresses of each VM and the local host machine. This hash table will be updated in a timely manner as the VMs are booted and shutdown. To build a communication connection with the monitoring host machine, these VMs' information must be transferred and synchronized to the monitoring host machine. Thus, once a new VM is booted, the latest local hash table will be transferred to the monitoring host for updating the information of VMs within a cloud.

Subsequently, when managers select a target VM in the monitoring host machine, the information in the local hash table will be used to build a communication connection between the monitoring host and the monitored host. Then, the VMM can further determine the target VM in the monitored host based on the local hash table.

2) MONITORING CONTROLLER

Correspondingly, inside the monitoring host, we must know all VMs, including local VMs and remote VMs, that run on the different hosts. Thus, we create a global hash table to record all VMs' information in a cloud. The contents in this global hash table come from the local hash table inside each host machine. The *target VM selector* in each host will timely transfer the latest data to this global hash table such that managers can select any VM in the cloud. Before cloud managers run a VMI program in the monitoring host OS, they can select any VM as the target VM by accessing this global hash table. Once the target VM is determined, the monitoring host machine will build a communication connection with the monitored host machine that the target VM is located on. Immediately after the communication channel is built, a manager can run a VMI program to monitor the target VM. Moreover, this constructed communication connection will be used to transfer the syscall redirection message and its corresponding syscall parameters between the introspection process and the re-sys process inside the target VM.

B. RE-SYS PROCESS BUILDER

To execute a redirected syscall inside the monitored VM, we have to select an in-guest process as the re-sys process to execute the redirected syscall. Although we could randomly select a user-space process inside the monitored VM, the re-sys process should meet two essential requirements in consideration of security and practicality. On the one hand, since the re-sys process is used to execute the redirected syscall, it cannot be a critical user-space process. Otherwise, it would affect the normal execution of the guest OS. On the other hand, as discussed in Section III-C, the re-sys process could be preempted (or traced) by a process (or `ptrace` syscall) with higher privilege, which would disturb the execution of the re-sys process. Thus, we should select an unimportant user-space process with a high privilege as the re-sys process.

As we know, an `init` process with `PID 1` is a common user-space process with supervisor (highest) privilege, and it is only used during the initialization phase of the OS. Thus, this `init` process meets the two requirements mentioned above. To find the `init` process inside a monitored guest OS, we search it by injecting a `getpid` syscall into the current process in the monitored OS and inspecting the value returned by `getpid` to determine whether this value is `1`.

Moreover, to notify the introspection process to resume the execution in the monitoring host, we need to intercept the exit point of the redirected syscall in the monitored OS. For this purpose, we inject a sensitive instruction (e.g., `CPUID`) into the code segment of the re-sys process immediately after it has been selected. The injected instruction will cause the guest OS to become trapped in the VMM, in which we can simply find the exit point of the redirected syscall execution.

In addition, the re-sys process depends on the syscall parameters from the introspection process. Thus, we need to pass the corresponding syscall parameters to the monitored VM. Moreover, when the re-sys process finishes execution, we need to transfer the updated data back to the monitoring host if there is any memory update. For this purpose, for the monitored VM located remotely, we allocate a block of buffer in the monitored OS for the re-sys process to obtain the syscall parameters from the monitoring host. For the monitored VM located locally, to avoid too much data being transferred between the monitored VM and monitoring host, we allocate a block of buffer in a shared memory to transfer the syscall parameters between them.

However, since the virtual addresses used by the introspection process in the monitoring host OS may not be the same as the virtual addresses used by the re-sys process in the monitored OS, we create a mapping relation between the allocated buffer and the re-sys process's address space. Then, when the re-sys process needs to access a virtual address from the monitoring host OS, it will access the corresponding address parsed through the mapping relation.

In addition, the re-sys process runs inside an untrusted VM, which could be attacked by malwares or rootkits. Once it is tampered with, the introspection process will obtain incorrect results. Thus, we must take some security measures to defend against those attacks. The defensive mechanism is introduced in detail in Section IV-E.

C. RE-SYSCALL SELECTOR

The key idea of CloudVMI is to redirect the syscalls of the introspection processes in the host OS into the monitored VM to bridge the semantic gap. We use the dynamic library interposition technique [27] to intercept each syscall issued by an introspection process in the library space of the monitoring host OS. In this way, the administrator can enable the functionality of introspection in the host OS by using the `LD_PRELOAD` parameter for introspection tools.

Once a syscall is intercepted, CloudVMI will determine whether this syscall should be redirected. As we know, not all syscalls should be redirected because certain redirected syscalls would inadvertently crash an in-guest process or cause unexpected results. For example, if the `execve` syscall is redirected, it will most likely change the in-guest process image permanently. Thus, we have systematically examined all syscalls (more than 300 syscalls in total) to define a detailed redirection policy by modifying and recompiling a new `glic` library, and we classify syscalls that need to be redirected into the following three categories.

1) **Inspection only.** The purpose of introspection is only to obtain current guest OS states and perform inspection, such as `getpid`, `uname`, `time`, `getuid32`, `olduname`, `newuname` and so on, which will be redirected.

TABLE 2. System call redirection policy.

Type	System Call	Redirect Policy
Inspection-only	time, getpid, getuid32, sync, uname, getppid, getgid32, getegid32, getuid32, umask, clock gettime, olduname, newuname, getrlimit, gettimeofday, getpriority, syslog, getxattr, lgetxattr	Redirect
Update-only	nice, kill, rename, setpgid, sethostname, settimeofday, delete_module, sysctl, route, renameat,	Redirect
Mix-dependency	open, read, write, close, link, unlink, stat, fstat, access, ioctl, fcntl, dup, dup2, symlink, readdir, fchmod, socketcall, ipc, init_module , llseek, getdents, chown, ftruncate64, stat64, lstat64, fstat64, getdent64, fcntl64, fstatatfs64, fchmodat	Redirect
Others	Relation to process, memory: exit, fork, waitpid, clone, execute, mmap, mmap2	No Redirect

2) **Update only.** Similar to inspection only, there are many syscalls, such as `kill`, `nice`, `sysctl`, `route` and so on, which can dynamically modify kernel states of the guest OS. These update-only syscalls will be redirected.
3) **Mix-dependency.** Unlike inspection only and update only, some syscalls, such as `open`, `write`, `getdent64` and so on, form a syscall dependency because they need to interrelate to accomplish some operations. For instance, when a process reads a file using a file descriptor that depends on the return value of the `open` syscall, a dependency relationship is formed between `open` and `read`. Thus, we stipulate that any syscall interrelated with a redirected syscall must be redirected. To trace the dependency among these syscalls, we utilize dynamical taint tracing to taint the result of the redirected syscalls; then, any syscall that uses this tainted value will be redirected.

Except for the above three types of syscalls, the remaining syscalls will be executed by the introspection process as usual in the host. Table 2 shows this particular redirection policy.

D. RE-SYSCALL REDIRECTION

When a syscall redirection request arrives, the re-sys process will be scheduled to execute this syscall. Before the syscall begins executing, CloudVMI needs to build the in-guest execution environment required by this redirected syscall. Specifically, CloudVMI must set the corresponding values of current registers. CloudVMI reads the syscall parameters from the allocated buffer and assigns these values to the current corresponding registers (e.g., setting the value of the `EAX` register to the syscall number).

Subsequently, CloudVMI injects a syscall into the re-sys process to start to execute the syscall based on the current CPU registers. For the syscall injection, we depend on Intel's VT technique to inject an artificial software interrupt (e.g., `int 0x80`) into the re-sys process through an event injection from the VMM.

E. SECURITY DEFENSE

In Section III-C, we analyzed all potential security threats in an untrusted VM. In this section, we introduce defensive measures against those threats.

1) USER SPACE

a: CODE INTEGRITY

A malware could tamper with the code of the re-sys process. To defend against this attack in the user space, we leverage the memory virtualization technique (i.e., EPT) to protect the memory area of the re-sys process from being accessed by any process. When the re-sys process accomplishes initialization, we first locate its address space in EPT, and then we directly isolate this address space by disabling the write privilege of the code segment of the re-sys process. Compared with the common page table inside a VM, EPT in the VMM is more secure, and it cannot be tampered with by any in-guest attack, even with root privilege.

b: DATA INTEGRITY

The buffer that stores the data referenced by redirected syscall parameters could be tampered with by user-space malwares to manipulate the execution results of the re-sys process through rewriting data back into the buffer. However, it is very difficult for an attacker to alter the buffer. First, no malware can alter the execution results of a redirected syscall by preempting the re-sys process because of our EPT isolation and the highest privilege of the re-sys process. Second, the memory location of the buffer is randomly allocated, and no malware can locate it. Thus, the buffer is sufficiently secure even though we do not take any protection measures.

c: CONTROL FLOW INTEGRITY

Since we select the `init` process with the highest priority as the re-sys process, it cannot be preempted by any user process, and it cannot be traced or controlled by any process. Thus, the malwares cannot intercept the control flow of the re-sys process by using a `ptrace` syscall.

2) KERNEL SPACE

a: CODE INTEGRITY

To manage attacks to kernel code integrity launched by rootkits, we leverage the file signature to detect whether the kernel code is broken. If a kernel is broken, then the VM will be rebooted for reloading the guest OS with an unbroken image.

b: DATA INTEGRITY

Existing VMI frameworks are vulnerable to DKSM [25] and DKOM [26] attacks. CloudVMI also does not have any

effective way to defend against such attacks, which is beyond the scope of CloudVMI.

c: CONTROL FLOW INTEGRITY

Many prototype systems have been proposed to check and protect the integrity of the static global variables of the guest kernel, such as HyperCheck [28], ModChecker [29], and HUKO [30]. However, these measures need the help of other components, such as SMM (system management mode); thus, they cannot be directly applied to our system to defend against kernel rootkit attacks. As analyzed in Section III-C, most kernel rootkits change the syscall control flow to implement malicious activities (e.g., hiding malicious processes) by hijacking IDT, syscall table, or kernel function pointers.

Since the addresses related to the syscall control flow (sys-addresses for short) are generally static [13], we design a consistency check mechanism to compare the values of the static sys-addresses to determine whether the syscall control flow has been changed. First, we obtain the original sys-addresses from the VMM layer when each guest VM is booted, and we store them in the VMM. Then, before the re-sys process executes any redirected syscall, the same sys-addresses will be obtained again to compare with the sys-addresses saved in the VMM. If any difference is found, CloudVMI will issue warning information about the broken syscall control flow to the introspection process after it finishes execution.

Consequently, the consistency check can reveal the correctness of the introspection results, which helps the administrators discover the existence of kernel rootkits in a timely manner. Additionally, these saved sys-addresses could be used for rootkit intrusion recovery to ensure the security of the monitored VM by replacing current changed addresses with saved addresses in the VMM.

TABLE 3. Key features of applications.

Features	*AResponse*	*VScan*	*HMon*	*NTopo*
Writability	√	√		
Cross-node Monitoring			√	√
Disk Introspection	√	√		

V. APPLICATIONS

In this section, we describe four concrete applications implemented on top of CloudVMI. These applications highlight the features of CloudVMI, including writable capability, cross-node monitoring, and disk introspection.

In Table 3, we highlight the key features of four applications: AResponse, VScan, HMon, and NTopo. AResponse is an application that can detect the hidden activities of the rootkits inside the monitored VMs; in particular, it can further automatically respond to the hidden activities without any administrator efforts. VScan is a disk virus scanner, and it can scan the disk file of monitored VMs from the outside. It can also automatically remove the virus files from the outside without any manual operations. HMon is a cloud-wide resource monitoring application that can monitor the resources of VMs distributed on multiple host machines. Thus, it can provide a more holistic view for cloud monitoring. NTopo is for the discovery of network topology in the cloud, which can discover a real-time network connectivity between different VMs or applications.

A. AResponse

AResponse is a security application for rootkit detection and recovery. On the one hand, it can timely discover the hidden activities of the rootkits, such as hidden processes and hidden modules, inside the monitored VM. On the other hand, when the hidden activities are discovered, AResponse will automatically remove them from the compromised VM without any administrator efforts. Compared with traditional automated response programs that are deployed inside the VM, AResponse has two merits. (1) It is more secure because it cannot be tampered with by any in-guest malwares. (2) It does not need the user login and root privilege of the guest VM to remove the discovered rootkits, which offers high automation.

For the monitored VM, AResponse periodically inspects its runtime states by running the `ps` and `lsmod` utilities in CloudVMI. When AResponse receives warning information about a broken control flow, it will perform a cross-view comparison between the current results and previous trusted results to find the hidden processes and modules, respectively. Once a hidden process is detected, AResponse will immediately run `kill` to remove it. Similarly, when a hidden module is detected, it also runs `rmmod` to remove it.

However, removing the hidden activity is just a response and not a recovery. Most kernel rootkits hijack the syscall control flow to hide in-guest activities by modifying the sys-addresses. Thus, once a hidden activity is detected, the sys-addresses will be broken (except for DKSM and DKOM attacks). Fortunately, the original sys-addresses have been saved in the VMM when a guest OS is booted, as discussed in Section IV-E.2. Thus, AResponse will further recover the integrity of a syscall control flow of the monitored VM after removing the hidden activities by replacing current inconsistent sys-addresses with the sys-addresses saved in the VMM.

To validate whether AResponse can discover the hidden processes and automatically remove them from the compromised VM, we first ran AResponse in the monitoring host to monitor a VM and obtain its process list. Then, we installed a kernel rootkit `adore-ng` inside this VM to hide an active in-guest process. As shown in Figure 3, warning information about broken control flow is shown in the current output of AResponse, and a process with `PID 1687` is included in the previous output but not in the current output. Thus, we can conclude that AResponse successfully discovers a hidden process inside the monitored VM. When a hidden process is found, AResponse will execute the `kill` utility

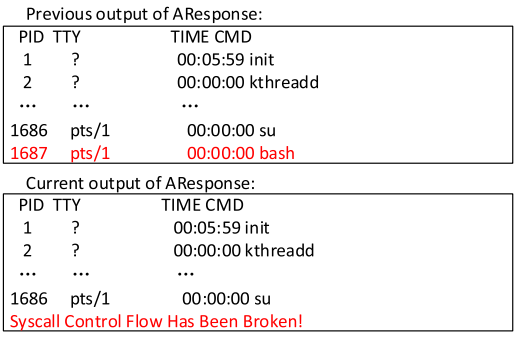

FIGURE 3. Hidden process discovered by AResponse.

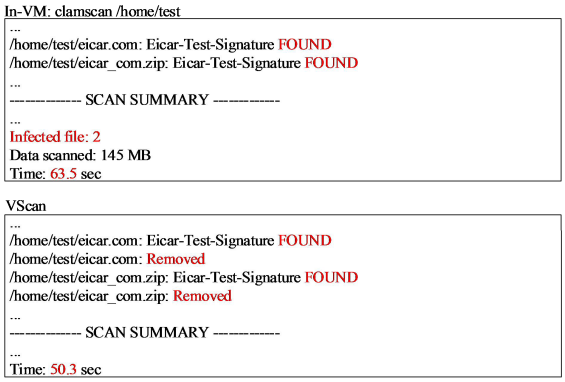

FIGURE 4. *VScan* finds and removes the virus files.

to remove it. To validate its effectiveness, we developed an in-guest program `kps` executed inside the monitored VM to traverse the kernel `task_struct.tasks` list and output all the `pids` of the monitored VM. Consequently, the process with `PID 1687` is not included in the set of `pids`, which proves that AResponse is effective for automatically removing the hidden process.

B. VScan

VScan is a hypervisor-level virus scanning application that runs outside of the monitored VM. Since CloudVMI uses syscall redirection to bridge the semantic gap, some legacy security tools that are implemented depending on syscalls can be executed in the monitoring host for security analysis. In fact, VScan is built over the popular open source anti-virus project ClamAV [31]. VScan can directly execute the `clamscan` utility in the monitoring host to scan files inside the monitored VM. Compared with the traditional ClamAV tool executed inside the monitored VM, VScan has two unique merits. (1) VScan has a negligible impact on the monitored VM when it is executed in the monitoring host, which will not affect the normal running of applications inside the monitored VM. (2) Since VScan is executed in the monitoring host, it will not be affected by additional `VMexit` caused by disk `I/O`. Thus, it will improve the efficiency of virus scanning with lower overhead.

VScan further provides the capability to automatically remove virus files based on the writable capability of CloudVMI. Before VScan begins scanning, we can create a scanning rule with `remove` arguments. When VScan finds a virus file inside the monitored VM, it will automatically remove this file from the compromised VM. Furthermore, VScan can scan multiple VMs distributed on multiple host machines, which can provide the capability of centralized VM management for the cloud.

To validate the capabilities of virus scanning and virus removal in VScan, we copied approximately 165 megabytes of files with two virus files (i.e., `eicar.com` and `eicarcom.zip`) under a test directory (e.g., `/home/test`). Then, we ran the `clamscan` utility inside the monitored VM to scan this directory. Subsequently, we ran VScan with a `remove` argument in the monitoring host OS to scan the same directory. The result of in-VM scanning and VScan is shown in Figure 4. `Clamscan` spends approximately 63.5 seconds to scan all files, and the two virus files are both found in `/home/test`. `VScan` spends less time, 50.3 seconds, to scan all files. Moreover, `VScan` automatically removes the two virus files found in the monitored VM, which confirms that VScan is effective in removing virus files from the outside based on the writable capability of CloudVMI.

C. HMon

HMon is a real-time resource usage monitoring application for the cloud, and it can monitor the resource usage of a guest VM, similar to what the Linux `top` utility provides. However, compared with the in-guest `top` utility in Linux, HMon has two enhancements for the cloud. First, most cloud applications typically span across multiple VMs and host machines, thereby requiring a holistic view of resource usage [15]. Thanks to the feature of cross-node monitoring in CloudVMI, HMon can monitor multiple guest VMs in the cloud. Thus, it can provide a holistic monitoring view of resource usage. Second, HMon operates outside of the monitored VM; thus, it can provide a VMM-level monitoring view rather than a system-level monitoring view. For example, it is able to know the resource usage of each VM on the host or each process on the host. To obtain a holistic view, we can successively calculate the resource utilization of the host (RUH), each VM (RUV), all VMs (RUV*), and each process on the VM (RUPV). Then, we can calculate the resource utilization of each VM on host (RUVH) and each process on host (RUPH) according to RUH, RUV, RUV* and RUPV. Equation 1 and Equation 2 show this calculation.

$$RUV_1H = \frac{RUV_1}{RUV*} \times RUH \tag{1}$$

$$RUPH = \frac{RUP}{RUV_1} \times RUV_1H \tag{2}$$

HMon monitors the resource usage of guest VMs by running legacy tools (e.g., `pstree` and `free`) to collect the runtime states of VMs, such as the usage of CPU or memory. To achieve close-to-real-time monitoring [32], HMon should collect these runtime states with a high frequency.

The frequency of monitoring will influence the performance overhead of the monitored VM. If it is too high, then it will cause a large amount of overhead. If it is too low, it cannot meet the requirement of real-time monitoring. Thus, we have to make a trade-off between overhead and real-time monitoring. HMon can dynamically adjust the frequency according to the resource utilization. For instance, when CPU usage is low, we argue that the VM is most likely secure; then, HMon will reduce the monitoring frequency. Conversely, when CPU usage is high, the VM may be suffering from an attack because the malwares or intrusion will cause a drastic increase in CPU usage. Accordingly, HMon will increase the frequency of monitoring to achieve real-time monitoring.

```
top - 22:23:23 up 6 min,  3 users,  load average: 1.89, 1.22, 0.56
Tasks: 112 total,   2 running, 108 sleeping,   0 stopped,   2 zombie
Cpu(s): 93.1%us,  5.2%sy,  0.3%ni,  0.0%id,  0.0%wa,  1.3%hi,  0.0%si,  0.0%st
Mem:   1034528k total,   236868k used,   797660k free,    31456k buffers
Swap:   901112k total,        0k used,   901112k free,   119380k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0  2036  716  620 R 83.7  0.1   4:37.16 init
 1839 root      30  10  1708  732  544 D  4.6  0.1   0:04.23 updatedb.mlocat
 1001 root      20   0 26284  14m 5340 S  4.3  1.5   0:25.13 Xorg
 1675 xgp       20   0 78872  11m 8824 S  3.1  1.1   0:05.11 gnome-terminal
```

```
Mem:   1034528k total,  236620k used,  797908k free   31236k buffers
Swap:   901112k total,   0k used,  901112k free

USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1 66.6  0.0   2036   716 ?        Rs   03:16   4:37 init [2]
root      1839  6.3  0.0   1708   732 ?        RN   03:22   0:04 /usr/bin/updatedb.mlocate
root      1001  6.3  1.4  21864 15344 tty7     Ss+  03:16   0:25 /usr/bin/Xorg :0 -br -verbose -audit 0 -novtsw
itch -auth /var/run/gdm3/auth-for-Debian-gdm-UpyyUy/database -nolisten tcp vt7
xgp       1614  2.8  1.5  97320 16504 ?        S    03:17   0:10 nautilus
xgp       1675  1.6  1.1  78872 11436 ?        Sl   03:18   0:05 gnome-terminal
```

FIGURE 5. **Above: in-VM top; Below: HMon.**

Figure 5 shows the execution result of HMon compared to the result of in-VM `top` in the same VM. As shown, the output between in-VM `top` and HMon is almost identical, which proves that HMon is effective.

D. NTopo

NTopo is a cloud-oriented network topology analyzer of VMs or applications, and it can discover the network connectivity across VMs running on multiple host machines within a cloud or across applications running on multiple VMs. Compared with traditional network monitoring tools based on SNMP, which require the agent to be deployed inside the monitored VM, NTopo is executed outside of the VM without the need of an inside agent, which is non-invasive and sufficiently secure because it will not face the attack threats caused by in-VM malwares. Based on the network topology provided by NTopo, we can further offer other potential use cases, such as managing and optimizing network connectivity and locating and solving network problems [33].

For each VM in the cloud, NTopo extracts the information of per-process network connection by running legacy network management utilities (e.g., `netstat`). This connection information mainly contains the socket type and state and the source and destination IP addresses. As NTopo traverses all VMs, the entire network status information is collected, and NTopo will further handle the collected states to generate the network topology. In addition, NTopo further extracts counts of received, transferred, and dropped messages to discover network traffic statistics for each VM.

To validate the capability of network topology discovery provided by NTopo, we constructed 4 test VMs on different host machines. These VMs include (1) a *logServer VM*, which stores the logs from a remote log client; (2) a *logClient VM*, which produces logs and sends local logs to the *logServer VM* based on the `rsyslog` service; (3) a *localLog VM*, which only produces logs and stores them in the local database; and (4) a *logAnalyzer VM*, which obtains and analyzes logs from the other three VMs based on the `httpd` service. Then, we ran *NTopo* in the monitoring host to discover the network connection between these 4 VMs. Figure 6 shows the network topology generated by NTopo between these 4 VMs. As shown, NTopo discovers all the network connections among these VMs.

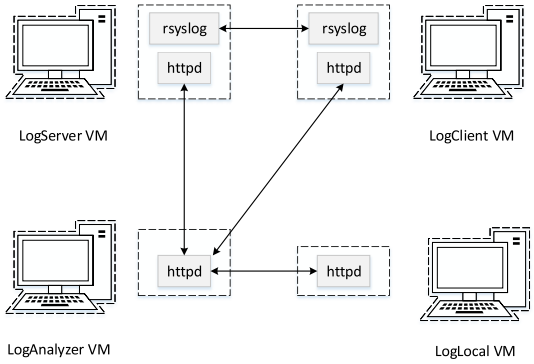

FIGURE 6. **Discovery of network topology.**

VI. EVALUATION

We have performed an empirical evaluation of our CloudVMI. In this section, we report our experimental results. All of our experiments were performed on an Intel (R) Core i5 2.40 GHz CPU with Intel hardware virtualization support and 8 GB of memory.

Below, we present four aspects of the evaluation results. First, Section VI-A describes the effectiveness of CloudVMI in terms of its unique capabilities of writable and cross-node monitoring. Section VI-B presents the performance impact to the VMI programs by comparing the overhead of the VMI applications using CloudVMI with the same programs executed inside the monitored VM and the overhead comparison between CloudVMI and LibVMI. Moreover, this section also presents the performance impact caused by CloudVMI on the monitored VM. Section VI-C shows the practical usefulness of CloudVMI by running a VMI tool to monitor multiple VMs with different guest OSes distributed on multiple hosts. Finally, Section VI-D presents the security of CloudVMI.

A. EFFECTIVENESS

In this section, we evaluate the effectiveness of CloudVMI to determine whether it can construct the in-VM semantic view from the outside. Since native Linux system utilities can be used as introspection tools without any modification, we selected 14 typical native tools (shown in Table 4)

TABLE 4. Effectiveness evaluation of CloudVMI. Local-VM means that the monitored VM is located on the monitoring host. Remote-VM means that the monitored VM is located on the remote host.

Categories	Utilities	Local-VM	Remote-VM
Read-only	ps	√	√
	uname -a	√	√
	lsmod	√	√
	free	√	√
	netstat	√	√
	iostat	√	√
	vmstat	√	√
	date	√	√
	mpstat	√	√
Writable	rmmod	√	√
	route	√	√
	sysctl	√	√
	nice	√	√
	hostname	√	√

as the benchmarks. To highlight the writable capability of CloudVMI, we classified these programs into two categories: one is `Read-only`, which can only inspect the state of the monitored VM, and the other is `Writable`, which can update the state of the VM. In addition, to examine whether CloudVMI can monitor the remote VMs that run on different hosts, we used these native tools to monitor the Local-VM that is located on the monitoring host and Remote-VM that is located on a different host from the monitoring host.

The concrete experiments were performed through the following steps. First, we ran each native program inside the monitored Local-VM and Remote-VM. Then, the same tool was immediately executed in the monitoring host OS using CloudVMI to monitor the Local-VM and Remote-VM. Finally, we compared the two pairs of results of the same suite of tools (i.e., Local-VM vs host and Remote-VM vs host) and checked whether they are the same in regards to semantics. As shown in Table 4, CloudVMI successfully constructs the in-VM semantic view from the outside.

B. PERFORMANCE OVERHEAD

Since the execution of an introspection process in CloudVMI will be divided into two process contexts, the introspection process itself in the monitoring host OS and the re-sys process in the monitored VM, we have to test two aspects of performance overhead. The first is the overhead of the introspection tools executed in the monitoring host. The second aspect is the performance impact to the monitored VM. We present these two types of overheads.

1) OVERHEAD OF THE INTROSPECTION TOOLS

In this experiment, we reused the native tools selected in 4 to test the overhead imposed by CloudVMI. Specifically, we used the `gettimeofday` function to retrieve a wall-clock time with microsecond accuracy. To improve the accuracy, we ran each of them 100 times in the monitoring host and monitored VM to respectively compute the average execution time and performance slowdown.

We first ran these native programs inside the monitored VM and calculated the average execution time. Then, we also ran these programs in the monitoring host using CloudVMI to monitor a Local-VM that ran on the monitoring host and a Remote-VM that ran on a different host to calculate the average execution time.

As shown in Table 5, CloudVMI introduces approximately 2.51× overhead on average when it is used to monitor the Local-VM and approximately 7.19× overhead when it is used to monitor the Remote-VM. This overhead mainly comes from the following two aspects. The first is the data exchange and synchronization between the monitoring host and monitored VM, and the second is the consistency check before the re-sys process executes each redirected syscall. We can find that the overhead of native programs when monitoring Local-VM is considerably less than the overhead when monitoring Remote-VM. The main reason for this result is that CloudVMI uses the shared memory rather than the communication channel to transfer the syscall parameters. Although CloudVMI introduces approximately 7.19× overhead when it monitors Remote-VM, the execution time spent by CloudVMI is in milliseconds. Compared to the time (approximately 6 seconds to list processes) spent in Virtuoso [11], the overhead of CloudVMI to native programs is negligible.

To further explain that the impact introduced by CloudVMI to native programs is acceptable for the cloud, we use the VMI tool LibVMI [10], which is the most commonly used VMI tool, to compare the performance impact to native programs. Since LibVMI cannot be used to monitor the VM that runs on a different host, we ran two typical VMI programs (i.e., `vmi_process_list`, `vmi_module_process`) provided by LibVMI itself to monitor a local VM. Additionally, we ran corresponding native programs (i.e., `ps`, `lsmod`) in the monitoring host using CloudVMI to monitor the same local VM. Table 6 shows the results of LibVMI and CloudVMI, and as shown, CloudVMI has almost the same overhead as LibVMI when running `ps` and `vmi_process_list`. However, LibVMI spends approximately 2.5 seconds to run `vmi_module_list`, but CloudVMI only spends 1.138 milliseconds to run `lsmod`, which has considerably less overhead compared to LibVMI. Thus, this experimental result demonstrates that our CloudVMI is acceptable for the cloud due to the low overhead.

2) PERFORMANCE IMPACT ON THE MONITORED VM

The performance impact on the VM falls into two scenarios. The first is to select and initialize the re-sys process before the redirected syscall is executed, which is called *redirection initialization*. The second is the redirected syscall execution that occurs in the re-sys process, which is called *redirection execution*. These two phases inevitably introduce a performance penalty to the running workloads at the monitored VM. Thus, if the monitored VM is not running during either the *redirection initialization* phase or the *redirection execution* phase, there will not be any performance overhead, which is called native VM. To quantify the overhead from these two scenarios, we used the standard micro-benchmarks and macro-benchmarks to measure the overhead. We ran these

TABLE 5. Overhead using CloudVMI. **Native Program** stands for the average execution time inside the monitored VM. **Local-VM** stands for monitoring local VM located on the monitoring host. **Remote-VM** stands for monitoring remote VM located on a different host.

Utilities	Native Program (ms)	Local-VM (ms)	Slowdown(X)	Remote-VM (ms)	Slowdown(X)
ps	7.685	24.862	2.24	121.193	14.77
uname	0.102	0.254	1.49	0.708	5.94
lsmod	0.186	1.138	5.12	1.839	8.89
free	0.067	0.427	5.37	0.969	13,46
netstat	11.882	25.089	1.11	50.724	3.27
iostat	0.862	4.065	3.72	10.860	11.60
vmstat	0.280	1.184	3.23	2.704	8.66
date	0.265	0.345	0.30	0.682	1.57
mpstat	0.483	3.137	5.49	4.023	7.32
rmmod	0.289	0.500	0.73	1.050	4.19
route	1.192	1.552	0.30	10.355	7.69
sysctl -w	0.016	0.022	0.38	0.034	1.13
nice	0.090	0.276	2.07	0.749	7.32
hostname	0.050	0.231	3.62	0.742	4.84

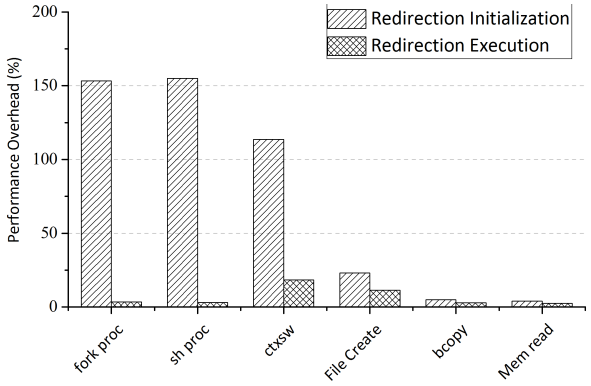

FIGURE 7. Test result of micro-benchmarks.

TABLE 6. The overhead comparison of VMI programs between CloudVMI and LibVMI.

Framework	VMI Tool	Time(ms)
LibVMI	`vmi_process_list`	23.084
	`vmi_module_list`	**2531.764**
CloudVMI	`ps`	24.862
	`lsmod`	**1.138**

benchmarks during *redirection initialization* and *re-process execution*, and we calculated the performance slowdown by comparing the execution time running during *redirection initialization* and *redirection execution* with the execution time running during the timespan of the native VM.

a: MICRO-BENCHMARKS

To evaluate the primitive-level performance slowdown, we used LMBench, which is a micro system performance evaluation tool in bandwidth and response time. We mainly focused on the overhead of the process creation (`fork proc`), C library function (`sh proc`), context switches (`ctxsw`), I/O-related operations (`File Create`), and memory-related operations (e.g., `bcopy` and `Mem read`).

As presented in Figure 7, for the *redirection initialization* phase, the large overhead primarily comes from the interception of the `getpid` syscall. However, this overhead cannot affect the monitored VM. In contrast, for the *redirection execution* phase, it introduces considerably less overhead because there is no syscall interception.

b: MACRO-BENCHMARKS

We used four real-world workloads to quantify the performance slowdown at the macro level: `bzip`, `kbuild`, `memcached` and `Apache`.

FIGURE 8. Test result of macro-benchmarks.

As presented in Figure 8, for the *redirection initialization* phase, the overhead mainly comes from the frequent `VMexit` caused by the `getpid` syscall interception. Hence, the workloads that include numerous I/O operations such as `kbuild`, `memcached`, and `Apache` will incur relatively large overheads. Regardless, the overhead with the maximum value of 89% in the case of `memcached` will not significantly affect the normal operation of the monitored VM. In contrast, during the *redirection execution* phase, all workloads have substantially less overhead because of less `VMexit`.

C. PRACTICALITY

In this section, we examine whether CloudVMI can be used to monitor multiple VMs with heterogeneous OSes even though these VMs run on different host machines. We used 4 physical nodes, including one monitoring host, and two VMs on each

TABLE 7. Test result of the generality.

Linux Distribution	Kernel Version	*Succeed?*
Ubuntu 9.04	2.6.28-11	√
Ubuntu 12.04	3.2.30	√
openSUSE 11.4	2.6.37.1	√
openSUSE 12.1	3.1.0-1.2	√
Debian 5	2.6.26-1	√
Debian 6	2.6.32-5	√
CentOS 4	2.6.9-89	√
CentOS 5	2.6.18-308	√

host (i.e., 8 VMs in total). Each VM ran heterogeneous guest OSes, most of which are the mainstream Linux distributions, including Debian, openSUSE, CentOS, and Ubuntu. Then, we ran a native program (e.g., `ps`) in the monitoring host OS using CloudVMI to monitor these monitored VMs. The test result is presented in Table 7, and as shown, CloudVMI is completely compatible with all of the guest OSes. Thus, because the syscall interface is seldom changed, CloudVMI has high generality, and it is practical to be applied in cloud environments.

D. SECURITY

As described in Section III-C, CloudVMI faces some security threats from the user space and kernel space of the monitored VM. To evaluate the security properties, we constructed one user-space malware and some kernel rootkits to simulate potential attacks.

1) USER-SPACE MALWARES

In the user space, we attempt to use the `ptrace` syscall to trace the execution of the re-sys process for overwriting the code injected into the re-sys process. The experimental result indicates that this operation is not allowed, and the code of the re-sys process cannot be changed. The reasons for this result mainly include two aspects. The first is that the re-sys process has the highest privilege that cannot be allowed to be traced or intercepted by the `ptrace` syscall (e.g., `PTRACE_SYSCALL`). The second is our EPT isolation mechanism that can defend against any modification to the memory pages of code segment. Thus, CloudVMI is sufficiently secure to defend against all user-space malwares.

2) KERNEL ROOTKITS

Most of the kernel rootkits hijack the syscall control flow by hooking the static kernel function pointers, including IDT, syscall table or other function pointers. In Section V-A, our consistency check mechanism can help *AResponse* discover the kernel rootkit and hidden process. In this experiment, we further took 7 common kernel rootkits and tested them with our consistency check mechanism. Not surprisingly, as presented in Table 8, the consistency check performs incredibly well, and it can successfully discover the control flow of a broken syscall, which means that CloudVMI can discover kernel rootkits and the control flow of syscalls in a timely manner. Thus, CloudVMI is also secure to defend against kernel rootkits.

TABLE 8. Rootkit detection with the consistency check mechanism

Rootkit	Target Attack Object	*Succeed?*
synapsys	Hooking Syscall table	√
suckit-2	Hooking Syscall table	√
kbdv3	Hooking Syscall table	√
kbeast-v1	Hooking Syscall table	√
override	Hooking Syscall table	√
hookswrite	Hooking IDT table	√
int3backdoor	Hooking IDT table	√

VII. DISCUSSION AND LIMITATIONS

Although CloudVMI can successfully bridge the semantic gap to offer the writable and cross-node VMI, it still has some limitations.

First, we have assumed that the syscall interface in the monitoring host and monitored VM is backward compatible. If there is any difference in the syscall interface between them, CloudVMI will be invalid. For instance, if a monitored VM uses a randomized syscall interface [21], then the redirected syscall issued by the introspection process will not be executed correctly by the re-sys process inside the monitored VM because of the different syscall number. Rather, we can perform a syscall translation between the monitoring host and monitored VM even though they have different syscall interface, and this would be a future work.

Second, we inject a `getpid` syscall from the VMM into a guest VM to select the `init` process as the re-sys process. However, this operation produces two side effects to the monitored VM. On the one hand, CloudVMI cannot be executed until the re-sys process is selected and completes the initialization; thus, this initialization phase will increase the startup time of CloudVMI and introduce a performance impact on the monitored VM. On the other hand, some communication code needs to be injected into the re-sys process during the initialization phase, which increases the scope of malware attack in the user space. By contrast, there is a better choice to select a kernel thread inside the monitored VM rather than a user-space process as the re-sys process.

Third, CloudVMI can only monitor VMs with the Linux kernel. In other words, when the monitoring host or monitored VM run another OS (e.g., Windows), CloudVMI will be invalid. Thus, to monitor VMs with Windows is considered as a future work.

Finally, we have implemented some security defense measures to defend against most security threats in the user space and kernel space of the monitored VM. However, we still do not have any measure to detect and defend against DKOM and DKSM attacks, and both of them threaten the security of CloudVMI and affect the results of VMI. How to effectively defend against DKOM and DKSM attacks is also considered as a future work.

VIII. CONCLUSION

We propose CloudVMI, a writable and cross-node monitoring virtual machine introspection framework, in which the

semantic gap is bridged via the syscall redirection. CloudVMI can be used to monitor multiple VMs with heterogeneous OSes even though these VMs run on different hosts, and it further provides a writable VMI capability that can update the state rather than only inspect the state of the monitored VM. Thus, CloudVMI is suitable for centralized and automated cloud management. Moreover, it is sufficiently secure to defend against a variety of attacks. We also built four typical applications on top of our prototype, namely, *AResponse*, *VScan*, *HMon* and *NTopo*, which highlight the features of CloudVMI. We evaluated CloudVMI in terms of effectiveness, overhead, practicality, and security. The experimental results demonstrate that CloudVMI introduces acceptable overhead for the introspection tools and has less performance impact on the monitored VM.

REFERENCES

[1] M. Sedaghat, F. Herández, and E. Elmroth, "Unifying cloud management: Towards overall governance of business level objectives," in *Proc. 11th IEEE Int. Symp. Cluster, Cloud Grid Comput.*, May 2011, pp. 591–597.

[2] T. Y. Win, H. Tianfield, and Q. Mair, "Virtualization security combining mandatory access control and virtual machine introspection," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2014, pp. 1004–1009.

[3] P. Mishraa, E. S. Pillia, V. Varadharajanb, and U. Tupakula, "Intrusion detection techniques in cloud environment: A survey," *J. Netw. Comput. Appl.*, vol. 77, no. 77, pp. 18–47, Jan. 2017.

[4] T. Garfinkel *et al.*, "A virtual machine introspection based architecture for intrusion detection," in *Proc. 10th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2003, pp. 191–206.

[5] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. 29th IEEE Symp. Secur. Privacy*, May 2008, pp. 233–247.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 51–62.

[7] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *ACM SIGOPS Operat. Syst. Rev.*, vol. 42, no. 3, pp. 74–82, 2008.

[8] H. Jin, W. Dai, and D. Zou, "Theory and methodology of research on cloud security," *Sci. China, Inf. Sci.*, vol. 59, p. 050105, May 2016.

[9] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Proc. 8th Workshop Hot Topics Operat. Syst.*, May 2001, pp. 133–138.

[10] *LibVMI*. Accessed: Aug. 30, 2017. [Online]. Available: http://libvmi.com/

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. 32nd IEEE Symp. Secur. Privacy*, May 2011, pp. 297–312.

[12] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proc. 33rd IEEE Symp. Secur. Privacy*, May 2012, pp. 586–600.

[13] Y. Fu and Z. Lin, "Exterior: Using a dual-VM based external shell for guest-os introspection, configuration, and recovery," *ACM SIGPLAN Notices*, vol. 48, no. 7, pp. 97–110, 2013.

[14] Z. Lin, "Toward guest os writable virtual machine introspection," *VMware Tech. J.*, vol. 2, no. 2, pp. 9–14, 2013.

[15] S. Suneja, C. Isci, V. Bala, E. De Lara, and T. Mummert, "Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 249–261, 2014.

[16] B. D. Payne, M. D. P. De A. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2007, pp. 385–397.

[17] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: An efficient out-of-VM approach for fine-grained process execution monitoring," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 363–374.

[18] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process implanting: A new active introspection framework for virtualization," in *Proc. 30th IEEE Symp. Rel. Distrib. Syst.*, Oct. 2011, pp. 147–156.

[19] Y. Fu, J. Zeng, and Z. Lin, "HyperShell: A practical hypervisor layer guest OS shell for automated in-VM management," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 85–96.

[20] R. Wu, P. Chen, P. Liu, and B. Mao, "System call redirection: A practical approach to meeting real-world virtual machine introspection needs," in *Proc. 44th Annu. IEEE Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 574–585.

[21] X. Jiang, H. J. Wangz, D. Xu, and Y.-M. Wang, "RandSys: Thwarting code injection attacks with system service interface randomization," in *Proc. 26th IEEE Int. Symp. Rel. Distrib. Syst.*, Oct. 2007, pp. 209–218.

[22] H. Zhang, L. Zhao, L. Xu, and L. Wang, "Cmonitor: VMI-based fine-grained monitoring mechanism in cloud," *Wuhan Univ. J. Natural Sci.*, vol. 19, no. 5, pp. 393–397, Sep. 2014. [Online]. Available: https://rd.springer.com/article/10.1007/s11859-014-1030-4?no-access=true

[23] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operat. Syst. Rev.*, vol. 41, no. 6, pp. 335–350, 2007.

[24] J. Joy, A. John, and J. Joy, "Rootkit detection mechanism: A survey," in *Advances in Parallel Distributed Computing*. Berlin, Germany: Springer, 2011, pp. 366–374. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-24037-9_36

[25] S. Bahram *et al.*, "DKSM: Subverting virtual machine introspection for fun and profit," in *Proc. 29th IEEE Symp. Reliable Distrib. Syst.*, Oct./Nov. 2010, pp. 82–91.

[26] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *Proc. 35th IEEE Symp. Secur. Privacy*, May 2014, pp. 605–620.

[27] T. W. Curry, "Profiling and tracing dynamic library usage via interposition," in *Proc. USENIX Summer Tech. Conf.*, 1994, pp. 267–278.

[28] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "HyperCheck: A hardware-assistedintegrity monitor," *IEEE Trans. Depend. Sec. Comput.*, vol. 11, no. 1, pp. 332–344, Jul./Aug. 2010.

[29] I. Ahmed, A. Zoranic, S. Javaid, and G. G. Richard, III, "ModChecker: Kernel module integrity checking in the cloud environment," in *Proc. 41st Int. Conf. Parallel Process. Workshops*, 2012, pp. 306–313.

[30] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity OS from untrusted extensions," in *Proc. 18th Annu. Netw. Distrib. System Secur. Symp.*, 2011.

[31] T. Kojm *et al. ClamAV anti-virus*. Accessed: Aug. 30, 2017. [Online]. Available: https://www.clamav.net/

[32] J. Hizver and T.-C. Chiueh, "Real-time deep virtual machine introspection and its applications," *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 3–14, 2014.

[33] Y. T. Zhou, M. L. Deng, F. Z. Ji, X. G. He, and Q. J. Tang, "Discovery algorithm for network topology based on SNMP," in *Proc. Int. Conf. Autom., Mech. Control Comput. Eng.*, 2015, pp. 1623–1628.

WEIZHONG QIANG received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is an Associate Professor with HUST. He is the author or co-author of about 30 scientific papers. His topics of research interests include system security about virtualization and cloud computing.

GONGPING XU received the B.Sc. degree in computer science from Hainan University, China, in 2014, and the M.Sc. degree in information security from the Huazhong University of Science and Technology, China, in 2017.

WEIQI DAI received the B.Sc. and Ph.D. degrees in information security from the Huazhong University of Science and Technology (HUST), China, in 2007 and 2014, respectively. He is currently an Assistant Professor with HUST. His research interests are mainly about system security.

DEQING ZOU received the B.E. degree in computer science and technology from Fuzhou University, China, in 1997, and the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), China, in 2004.

He is currently a Professor of computer science with HUST. He has applied 20 patents, authored or co-authored two books and over 50 research papers. His main research interests include system security, trusted computing, virtualization, and cloud security. He is on the editorial boards of four international journals, and has served as PC chair/PC member of over 40 international conferences.

HAI JIN (SM'06) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 1994. In 1996, he was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany. He was with The University of Hong Kong from 1998 to 2000, and as a Visiting Scholar with the University of Southern California from 1999 to 2000. He is a Cheung Kung Scholars Chair Professor of computer science and engineering with HUST. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of CCF and a member of the ACM. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001.