In Proceedings of the 5th International Conference on Network and System Security (NSS 2011), Milan, Italy. (c) IEEE 2011

CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model

Amani S. Ibrahim, James Hamlyn-Harris, John Grundy and Mohamed Almorsy
Centre for Computing and Engineering Software Systems
Faculty of Information & Communication Technologies
Swinburne University of Technology
Hawthorn, Victoria, Australia
[aibrahim, jhamlynharris, jgrundy, malmorsy]@swin.edu.au

***Abstract* — The Infrastructure-as-a-Service (IaaS) cloud computing model has become a compelling computing solution with a proven ability to reduce costs and improve resource efficiency. Virtualization has a key role in supporting the IaaS model. However, virtualization also makes it a target for potent rootkits because of the loss of control problem over the hosted Virtual Machines (VMs). This makes traditional in-guest security solutions, relying on operating system kernel trustworthiness, no longer an effective solution to secure the virtual infrastructure of the IaaS model. In this paper, we explore briefly the security problem of the IaaS cloud computing model, and present CloudSec, a new virtualization-aware monitoring appliance that provides active, transparent and real-time security monitoring for hosted VMs in the IaaS model. CloudSec utilizes virtual machine introspection techniques to provide fine-grained inspection of VM's physical memory without installing any monitoring code inside the VM. It actively reconstructs and monitors the dynamically changing kernel data structures instances, as a prior step to enable providing protection for kernel data structures. We have implemented a proof-of-concept prototype using VMsafe libraries on a VMware ESX platform. We have evaluated the system monitoring accuracy and the performance overhead of CloudSec.**

Cloud Computing; IaaS Security; Semantic Gap; Virtual Machine Introspection; VMware ESX; VMsafe APIs; Virtual Appliance.

I. INTRODUCTION

Cloud computing is a new computing paradigm that delivers reliable and scalable internet-based services. Infrastructure-as-a-Service (IaaS) is one of the main service models delivered by cloud computing. IaaS allows customers to increase their computational and storage resources on the fly without investing in new hardware. IaaS is characterized by the concept of resource virtualization. Virtualization enables running multiple Operating System (OS) instances - called virtual machines (VMs) - on the same physical server. The Cloud Virtual Infrastructure (CVI) in the IaaS model is composed of three main components [1]: a hypervisor, virtual switch (vSwitch) and hosted VMs. The hosted VMs are considered the main source of security threats on the IaaS platform, where the cloud provider is hosting VMs without being aware of their actual contents, and with no control over these VMs. This makes it easy to seize the hosted VMs and thus opens the possibility for a compromised VM to attack the other hosted VMs or the hypervisor itself, as VMs share the same hardware and hypervisor software. Such VMs cannot be trusted from the cloud providers' perspective to install their supported security software inside these VMs. This is because it becomes possible to tamper with the security software and alter its behaviour, whether by the cloud consumer (the VM owner) or by an external hacker. Hence, traditional in-guest security solutions that rely on the Operating System (OS) kernel trustworthiness do not map well to secure such virtualized systems. Thus new Virtualization-Aware Security Solutions (VASSs) need to be provided. Such VASSs must have the ability to actively monitor and protect the hosted VMs from outside the VM itself, without installing any security code inside the VM. As virtualization has complicated the security process for the IaaS platform, it has also enhanced the trustworthiness of the security process. This is by enabling monitoring of VMs externally, at a hypervisor level, by observing the hardware bytes e.g. memory pages and disk blocks. But this still presents a key problem of how to map these hardware-level bytes to useful OS abstractions to maintain security.

In this paper, we present *CloudSec* - a monitoring appliance that provides active, transparent, and real-time security monitoring for the hosted VMs in the IaaS cloud platform. *CloudSec* utilizes Virtual Machine Introspection (VMI) techniques to provide fine-grained inspection of the VM's physical memory, without installing any security code inside VMs. Monitoring volatile memory enables effective detection of user or kernel rootkits, as volatile memory must have imprints for such malware, even self-hiding malware. *CloudSec* actively reconstructs and monitors the dynamically changing kernel Data Structures (DSs) instances to enable effective detection and prevention for the kernel data rootkits such as Dynamic Kernel Object Manipulation (DKOM) and Kernel Object Hooking (KOH) rootkits. We explore the challenges of implementing such security monitoring system, and how we can map the introspected low-level raw bytes of memory into high-level OS data structures instances. We have implemented a proof-of-concept prototype using VMsafe libraries on a VMware ESX cloud platform.

The rest of this paper is organized as follows: Section II explores the VMI techniques, the semantic gap problem, and the key requirements for implementing an effective external VM monitoring system for memory protection purposes.

Section III describes the *CloudSec* high-level system architecture, and in section IV we explain in details the steps of implementing and deploying *CloudSec*. Section V describes an evaluation of our prototype, and section VI discusses the performance overhead of CloudSec and key directions for future research. Finally we outline our conclusions.

II. Motivation and Background

Virtual Machine Introspection (VMI) [2] enables monitoring of VMs from the outside, at a hypervisor level, but only hardware-level raw bytes can be observed. In contrast, from inside the VM, we can view high-level entities such as processes, I/O requests, and system calls. The difference between these outside and inside views is called the *semantic gap*. In order to make external VM observations useful for security monitoring, it is necessary to translate the hardware-level bytes to actual running OS information.

VMI also enables isolating the security solution from the other server workloads, by deploying the security solution in a dedicated VM that has a privilege access to the hardware through the hypervisor. This makes it much harder for hackers to detect the installed security software. Recently, *virtual appliances* [3] have been introduced as a new solution for deploying security VMs. A virtual appliance, like a VM, incorporates application, OS and virtual hardware. However, virtual appliances differ from VMs in that they are delivered as preconfigured solutions running a "Just enough Operating System" (JeOS). JeOS is a purpose-built OS that supports only the functions of the installed software [3]. JeOS occupies a much smaller footprint than a general-purpose OS and thus a JeOS is more stable and secure, reducing the number of vulnerabilities and exploits that can occur. Additionally, for VMs running on the same physical machine, significant memory redundancy is likely to exist. Memory de-duplication that is supported in most hypervisors e.g. Xen and VMware ESX, aims to eliminate the memory resource waste. However, because of mismatch in page alignment and use of physical pointers, not all memory redundancy can be effectively uncovered. Another way to eliminate this inter-VM redundancy is through kernel component refactoring. This takes out common kernel components shared by VMs running on the same physical machine and runs them on a separate virtual appliance [4].

A. Key Problems

Providing reliable monitoring using VMI techniques for the hosted VMs' volatile memory, in order to help detecting security anomalies and memory-based rootkits, requires solving two main problems:

1) Bridging the semantic gap by reconstructing the OS high-level semantic view externally, without relying on the running kernel memory or APIs. Such translation requires accurate mapping between the OS kernel structure of the VM, and the physical memory layout.
2) Providing real-time and active monitoring for multiple, concurrent VMs hosted on a single physical server without a noticeable VM performance overhead. Running real-time monitoring software impacts performance, because the security software typically needs to verify system activities such as calls and memory accesses before executing them. Active monitoring is also a difficult problem in VASSs. Active monitoring requires installing hooks inside the hosted VMs to suspend system activities until they are analyzed, and a major reason for moving to VASSs is to remove the security code from the hosted VMs [5].

B. Related Work

VMs have become widely used for deploying security software even before the cloud computing era, due to the benefits gained from virtualization. After the wide-spread adoption of the IaaS model, securing the VMs at a hypervisor level has also become a new trend in security research. Most VMI research to date has been done on the Xen hypervisor platform as this is an open source hypervisor.

Livewire [2] is the first intrusion detection system that applied VMI techniques. Livewire works offline and passively. Most recent VMI research has depended on installing code inside the hosted VMs whether to solve the semantic gap or to enable real-time and active monitoring. One of the most popular VMI research platforms for Xen hypervisor is XenAccess [5, 6]. XenAccess installs write-protected security hooks inside the guest VM to suspend VM events while they are being analysed, to enable active monitoring. PsycoTrace [7, 8] depends on installing hooks inside the VM to solve the semantic gap problem. KvmSec [9] also relies on installing security code inside the guest VMs to obtain high-level OS abstractions from inside the guest to solve the semantic gap problem. SIM [10] is an in-VM monitoring system that places a protected security code inside the VM. All of these approaches install security code in the hosted VMs to solve the semantic gap or to enable active monitoring. VIX Tools [11, 12] doesn't install code to solve the semantic gap, but it does not provide real-time monitoring, where it requires pausing the VM until reconstructing semantic gap. As discussed above, this violates our key requirements and exposes the security code hosted in VMs to detection. DKSM [13] has recently been introduced as an attack prototype that can defeat such security solutions that are OS-dependant in solving the semantic-gap. DKSM introduced two attack models: one changes the syntax of the kernel structure and the other changes the semantics. Such attacks can be detected because OSs like Microsoft Windows and Linux embody basic object-oriented design principles [14]. They structure their kernel data into set of predefined data structures and objects, with a relative virtual address schema. This structure of the kernel enables reconstructing kernel data structures through recursive traversing of physical memory using the OS global variables, without relying on the OS kernel that could be exploited by one of those attacks.

Virtual Appliance technology for deploying security software has had little attention to date in academic research on deploying security solutions. However, virtual appliance technology is used widely by security vendors such as McAfee [15] to deploy security solutions especially for virtualized data centres and cloud computing platforms. Security research

targeting cloud computing platforms (especially IaaS) is also still relatively limited. Most current cloud security approaches [16, 17] depend on deploying current security technologies, such as intrusion detection and prevention systems, inside the hosted VMs. However, some researchers have discussed the complexities of the cloud platform and the challenges of implementing security solutions for the cloud [1, 18, 19].

III. System Architecture

CloudSec utilizes VMI techniques to monitor the physical memory of hosted VMs at a hypervisor level. It then reconstructs externally a high-level semantic view of the running OS kernel data structures instances for the monitored VMs' OS. The key idea behind solving the semantic gap is how to map accurately between the underlying hardware memory layout and the OS kernel structure. As mentioned before, OSs like Microsoft Windows and Linux embody basic object-oriented design principles. They organise their kernel data into a set of predefined data structures and objects. This organisation of the kernel enables reconstructing kernel data structures through recursive traversing for the physical memory using the OS global variables, without relying on the OS kernel. However, the problem is more difficult in commodity OSs like Microsoft Windows, because Windows OS is not open source like Linux OS. In addition Windows kernel data structures are opaque - the addresses of OS global variables change from one Windows build to another. Fortunately Microsoft provides Microsoft Symbols [20] which provides symbols for each Windows build. These symbols can be used as a reference for reconstructing the kernel structures from the memory bytes instead of relying on false view of a compromised kernel.

A. Threat Model

We assume a trustworthy hypervisor, based on the assumption that the source code of the hypervisor is much smaller and more reliable than the existing OS kernels. Existing hardware-based protection technologies, including Trusted Platform Module [21] and Intel trusted Execution Technology [22], are also capable of effectively establishing a root of trust. This is by guaranteeing the loading of a hypervisor in a trustworthy manner. In other words, they can guarantee the load-time integrity of the hypervisor. Thus, the cloud platform (hypervisor including the vSwitch) is part of the Trusted Computing Base (TCB). Also, *CloudSec* is part of the TCB where it's completely controlled by the cloud provider and isolated from the other server workload. On the other hand, the hosted VMs are not part of the TCB and cannot be trusted to install any monitoring code. We assume a hacker has root privilege access to a VM including OS and applications, and they can modify any code or data in the OS kernel of the target VM.

B. CloudSec Architcture

Figure 1 shows the high-level architecture of *CloudSec*. The VMI layer is the core of *CloudSec* architecture; VMI layer is composed of two components:

1) The *back-end* component; enables the hypervisor to gain control over the hosted VMs to suspend any access to the physical memory and CPU (1) according to the access triggers installed by *CloudSec* using the front-end. The back-end extension notifies *CloudSec* (2) to perform the necessary security checks (3), before control is given back to the VM executed instructions (4). This enables *CloudSec* to perform active monitoring, without installing any hooks inside the VM to suspend instruction execution.
2) The *front-end* component; is a set of APIs that enable getting information about the monitored VM's running OS from the hypervisor and controlling accesses to physical memory and CPU registers. The *front-end* APIs enable *CloudSec* to install memory access triggers or timer-based triggers on the physical memory pages that need to be monitored. The *front-end* makes *CloudSec* an external extension of the hypervisor which enables transparent access to physical memory.

Communications between the *front-end* and *back-end* components are controlled by a communication channel conducted over a separate virtual network using a separate vSwitch.

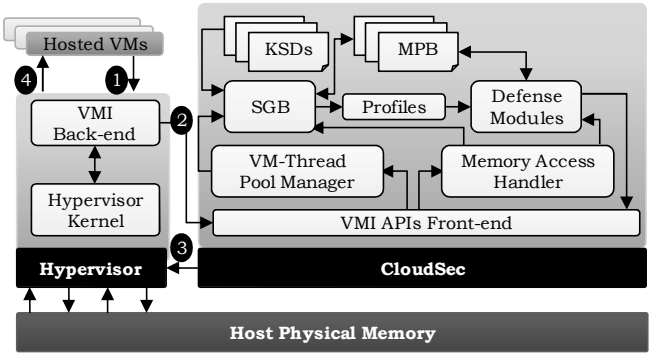

Figure 1. CloudSec high-level architecture

Whenever a hosted VM is powered on, the *CloudSec* VM is notified by the *back-end* via the communication channel. *CloudSec* then creates a separate thread for each newly activated VM using the VM-Thread Pool Manager. *CloudSec* first checks the control registers of the VM's CPU to know the memory layout of the VM's hardware, and queries the kernel version of the VM via the hypervisor to load the appropriate Kernel Structures Definition (KSD). The KSD differs from one OS to another. For Windows OS, we can get complete KSDs from Microsoft Symbol packages. For Linux, KSDs can be obtained from the kernel symbol table (*e.g.* System.map). After loading the appropriate KSD, *CloudSec* starts solving the semantic gap through our Semantic Gap Builder (SGB). The SGB reads specific physical memory pages, according to the corresponding OS global variables' addresses from the specified KSD. *CloudSec* does not have direct access to the VM's physical memory, which is controlled by the hypervisor. It instead uses the *back-end* to read these physical memory pages into the Memory Pages Buffer (MPB). Then, the SGB

starts mapping these physical memory bytes to the corresponding KSD, to obtain an OS view of the running VMs externally. This view includes reconstructing all kernel data structures e.g. the running processes, loaded modules, system table, and interrupt, local and general descriptor tables. After constructing the kernel data structures view externally, *CloudSec* creates a profile for each VM containing its reconstructed high-level semantic view, to be used by the Defence Modules. *CloudSec* then installs memory access triggers or timer-based triggers on the page(s) that contain the kernel data structures that needs to be monitored and protected according to the applied defence mechanisms. Whenever a memory access to such pages occurs, the *back-end* notifies the Memory Access Handlers (MAHs), and the hypervisor suspends execution. MAHs load the requested memory page(s) to the Defence Module or the SGB to extract kernel data structure updates (if required by the defence module). Then the defence modules analyse the current running kernel data structures for security threats, according to the applied defence mechanisms.

Live migration of the protected VMs can be done, if CloudSec is running on both source and target hosts, and they are both listening to the same network address and port number.

IV. IMPLEMENTATION

We have implemented *CloudSec* using the VMsafe APIs on a VMware® ESX 4.1. VMware has introduced VMsafe APIs to allow security experts to leverage VMI techniques for hosted VMs in the ESX hypervisor. VMsafe offers transparent access to the virtual hardware of hosted VMs. VMsafe is composed of three main libraries: vCompute, vNetwork and vStorage APIs[1]. *CloudSec* uses the vCompute APIs to access the physical memory of hosted VMs. Our implementation is based on the KSD of the Windows XP SP3, for other Windows versions, we just need to load *CloudSec* with the appropriate KSD from Microsoft Symbols.

A. Memory Management

One of the key requirements needed to solve the semantic gap is to understand the underlying memory layout that is used by the OS to map between physical memory pages and the OS's KSD. There are four main paging modes supported by the hardware to manage the physical memory layout. These paging modes are controlled by the control registers CR0 and CR4 of the VM's CPU. For example, if the CR0.PG bit is set and CR4.PAE bit is clear, then 32-bit paging is used. If the CR0.PG bit is set and CR4.PAE bit is set then Physical Address Extension (PAE) paging is used [23]. We focus in our current implementation on the 32-bit paging mode, PAE disabled. In this mode, memory is divided into pages or frames of 0x1000 (4096) bytes each. Each process has a Page Directory Table (PDT). Each PDT contains 1024 Page Directory Entries (PDE). Each PDE contains an address for a Page Table (PT). Each PT contains 1024 PT Entries (PTE), Each PTE points to the base address of a page. This gives 1024*1024*4096 or 4GB total Virtual Address (VA) space for each process. In Windows OS, the 4GB of virtual address space is divided into two halves: the first 2GB (from 0x80000000 to 0xFFFFFFFF) is for the kernel address space, and the second 2GB (from 0x00000000 to 0x7FFFFFFF) is for the user address space. Thus, to calculate a kernel PA from a given VA, we subtract 0x80000000. To translate a VA in a certain process user address space to the corresponding Physical Address (PA), we use the linear address translation mechanism which requires the physical address of the PDT for the given process.

Shadow Page Tables. The hypervisor introduces another layer of address translation; the mapping of the guest physical pages to the host physical pages. With Memory Management Unit virtualization, the hypervisor constructs "shadow page tables" that combine both the guest page tables and the additional layer of page tables to directly map a guest linear page to a host physical page on the hardware page table. As a result, there are three memory layers: guest virtual memory, guest physical memory, and host physical memory. As the hypervisor mediates interactions between VMs and hardware, the guest physical memory is controlled by the hypervisor through the shadow page tables, and the hypervisor doesn't provide any direct control over the host physical memory even for *CloudSec*.

B. Implementation Details

We selected two important Windows kernel structures to construct externally as a proof-of-concept for our prototype. These two data structures are often a target for hackers to install hooks, inject malicious code, or to hide malicious processes. These kernel structures are:

1. **EPROCESS Structures** - by locating such structures in physical memory pages, we can externally list all the running processes including their details (e.g. name, ID, threads, loaded modules, Export/Import Address Tables, Virtual Address Descriptors). Monitoring and constructing processes without relying on the kernel enables *e.g.* detecting hidden process, DLL injection and EAT /IAT hooking.
2. **KeServiceDescriptorTable Structure** - this structure references the System Service Dispatcher Table (SSDT) that contains a list of the native kernel APIs and their addresses, making it one of the central points of execution flow control in the kernel. Monitoring this structure externally without relying on the kernel enables detecting any system-wide hook.

1) EPROCESS Blocks

Processes in Windows are represented in kernel address space as executive process blocks called EPROCESS structures. An EPROCESS structure contains detailed information about a running process. For each running process, there is a dedicated EPROCESS structure and all EPROCESS blocks are structured in a doubly linked list called *ActiveProcessLinks*, as shown in figure 2.

[1] vCompute and vNetwork libraries are not for public use and their license can only be provided to security vendors and security researchers.

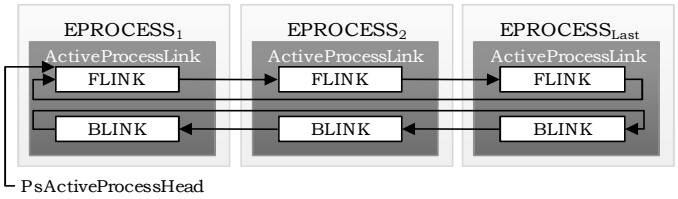

Figure 2. EPROCESS doubly-linked list

Doubly-linked list means that each block has a Flink entry that refers to the forward node in the doubly linked list and a Blink entry that refers to the back node. Each node in the doubly-linked list represents a running Windows process. Getting the address of one EPROCESS structure enables obtaining the rest of the running processes' EPROCESS blocks through traversing the doubly-linked list.

EPROCESS structures are dynamic data objects; their virtual addresses are assigned at run-time. The key challenge here is how to get the addresses of the doubly-linked list nodes taking into consideration that we can only read the VM's physical memory bytes. Our solution to getting the addresses of the processes' nodes is through recursive traversing for the physical memory using specific global OS variables that control the running processes list. In Windows OS, the first loaded process is always the "*System*" process and it's the first node in the doubly-linked list. The global variable "*PsActiveProcessHead",* points to the *ActiveProcessLinks* address of the *"System*" process, as shown in Figure 2. Figure 3 shows key data structures and items with their offsets in the EPROCESS structure that we have used to get process details, and Figure 4 summarizes our algorithm for building the list of the running processes and their details externally.

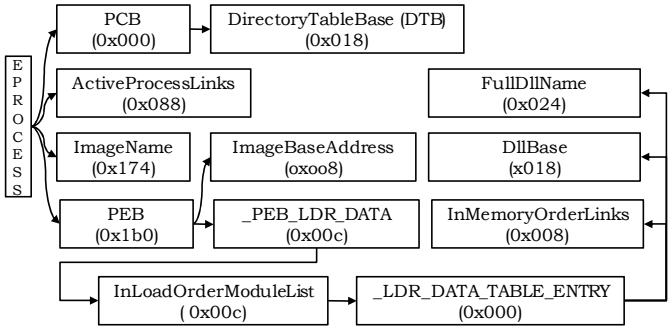

Figure 3. EPROCESS key data structures and items.

The *PsActiveProcessHead* address is static in Windows OSs and doesn't change in run-time, and from the KSD (from Microsoft Symbols) of XP SP3, its address is 0x805638b8. Subtracting the offset of the *ActiveProcessLinks* member (0x88) from the value pointed to by this address, gets the "*System*" process's EPROCESS VA. After translating this VA to its PA and reading the corresponding physical memory page, we can extract process details e.g. process name, ID, threads, Page Frame Number database and Virtual Address Descriptors from the EPROCESS structure according to the KSD offsets of these members.

To build the process loaded modules (DLLs), we first get the PEB structure VA (offset 0x1b0 in EPROCESS), and then translate it to PA to read its physical memory page. From the process PEB structure we extract the address of the PEB_LDR_DATA data structure, using their corresponding osffset. PEB_LDR_DATA contains pointers to doubly-linked lists for the process loaded modules in different orders: load order, memory order and initialization order. Each loaded module is represented with a LDR_DATA_TABLE_ENTRY structure which is part of the modules doubly-linked list and contains details *e.g.* module name, base address and size. To determine the end of the modules list, we check the Flink of current node against the address of the first module structure in the list. If the two are equal, then we have got all of the process loaded modules.

At this point, we have located the details of the first node "*System*" process. To extract the rest of the running processes, we traverse the doubly linked list of the *ActiveProcessLinks* structure for the next node, and repeat the previous steps to build the process details. To determine the end of the process list, we check the Flink of each EPROCESS against the address of *PsActiveProcessHead*. If they are equal, then the current EPROCESS structure is the last process in the list.

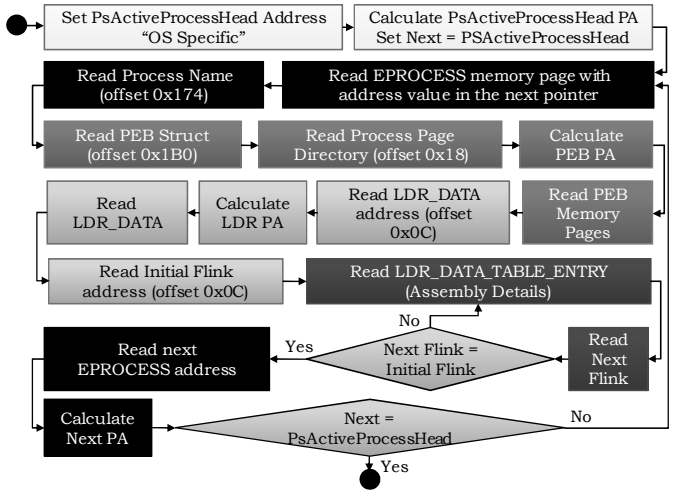

Figure 4. EPROCESS semantic gap construction.

A drawback of listing the processes through the doubly linked list is that this method becomes vulnerable to process hiding rootkits that modify the Flink and Blink addresses to hide a process. Mihir *et al*. [24] explore different techniques for detecting hidden processes with their limitations and drawbacks. They introduce a solution that is based on monitoring the scheduled threads of the processes. The authors also mention that their technique can be overcome by recent rootkits that build their own scheduler. To solve this problem, we update our process list whenever any creation or termination for a process occurs. We achieve this by installing memory access triggers on the physical memory page that contains the SSDT, to monitor *NtCreateProcess* and *NtTerminateProcess* functions that are exported by the SSDT. Each of the SSDT functions has a unique system call number. This number is loaded into the EAX register while the

processor initiates interruption to call a function from the SSDT. We check the EAX register value within each memory access to the SSDT physical page. If the EAX register holds the system call number of one of those functions, then we build our processes list again. If we find that a process does not exist in the current process list but exists in the initial process list, then we check the process ID, PDT and ThreadListHead members of this process's EPROCESS. Those members can't be changed during the process runtime [25]. If these members are still available with the same values, this indicates that the process has been hidden not terminated.

2) KeServiceDescriptorTable Structure

Figure 5 summarizes our algorithm for reconstructing the SSDT table. The address of the SSDT kernel data structure can be found within the Service Descriptor Table (SDT). The SDT is referenced by the *KeServiceDescriptorTable* global structure and its address is static in the Windows OS, thus we can get its address from the appropriate KSD. The first member of the SDT structure is the *KiServiceTable* (SSDT address), and the third entry (offset 0x0c) is the number of SSDT entries. To enumerate the SSDT entries, we traverse the physical memory page that contains the SSDT address until the number of read entries is equal to the value at the offset 0x0c. Reconstructing the SSDT, allows *CloudSec* to install write-memory-access triggers on the SSDT page, to detect SSDT hooking rootkits that target to install system-wide hooks by modifying the SSDT.

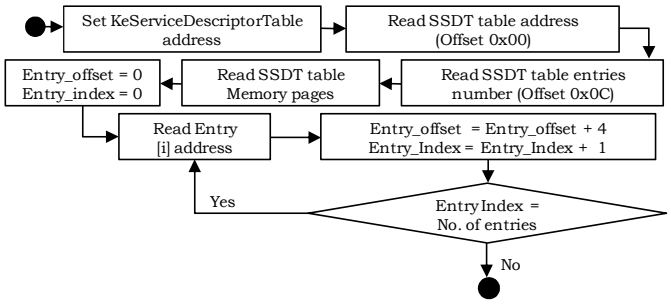

Figure 5. KeServiceDescriptorTable reconstruction.

C. CloudSec Deployment

Our evaluation and implementation platform for *CloudSec* is the Intel x86 family of microprocessors. The cloud platform hardware consists of a HP Z400 – 2.8 GHz Intel® Xeon® CPU (VT-x) with 6126 MB of RAM. This workstation runs VMware ESX 4.1. The ESX server hosts three VMs, as shown in figure 6. These machines were configured as follows:

- **CloudSec VM;** *CloudSec* VM is configured with 2 GB of RAM and two virtual CPUs. *CloudSec* is a virtual appliance running Ubuntu Linux 8.04 Server JeOS, and hosts the vCompute APIs and our monitoring software. Our monitoring software is a normal Linux C program written using vCompute APIs and Posix Threads APIs to support multi-threading , in order to enable monitoring multiple VMs concurrently. *CloudSec* is isolated from other server network workload in a separate virtual network by creating a dedicated vSwitch in the ESX hypervisor. *CloudSec* has a pre-defined network IP and port to enable monitoring other hosted VMs.
- **Monitored VMs**; our prototype contained two hosted VMs for validation purposes. The two VMs are each allocated with 1.5 GB of RAM and two vCPUs running Windows XP SP3 32-bit OS. The VMX files of these VMs were configured to allow introspection by the vCompute API.

A fourth VM was used for development purposes to write and test our monitoring code. As *CloudSec* runs a JeOS, and such OSes are GUI-less, we used an extra VM running CentOS 5.5 for the purposes of programming.

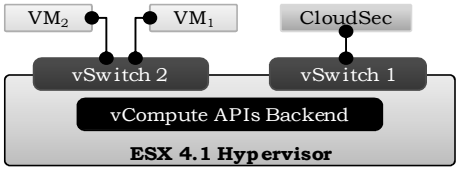

Figure 6. The experimental setup lab.

V. EVALUATION

We used *WinDbg* from the Debugging Tools for Windows [26] to validate that *CloudSec* bridges the semantic gap successfully. We compared the external view of mapping the introspected VM's physical memory to Windows OS kernel data structures using *CloudSec,* with the internal view of the VM using WinDbg. *CloudSec* accurately gets high-level OS information from outside a VM without installing any monitoring code inside the target VM.

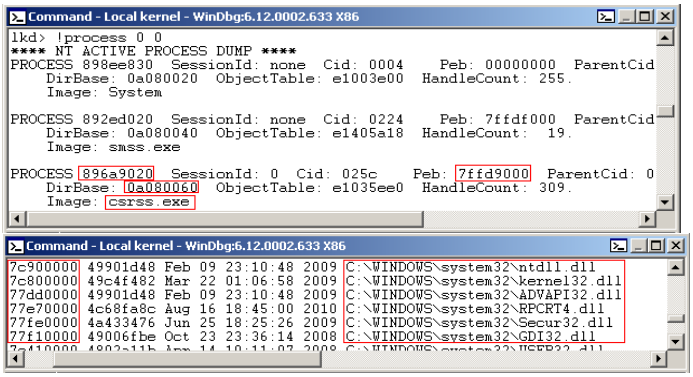

Figure 7. Internal VM view for the running processes list using WinDbg.

CloudSec provides detailed information about the running processes inside a hosted VM, from outside the VM. Figure 7 shows screenshots for the internal view for the running processes using *WinDbg*. We have selected the *csrss.exe* for the comparison, and other processes have been omitted for brevity. The top screenshot shows the *csrss.exe* basic information such as the virtual address of the EPROCESS, PEB, and the DirBase (DTB). The bottom screenshot lists the loaded DLLs for this process (six modules for brevity). Figure 8 shows the results of *CloudSec* external view of the EPROCESS blocks, focusing on the details of a process at the

address 0x896a9020. After constructing the semantic information we extracted the process name (*csrss.exe*), PDT and loaded modules and other information. The comparison shows that the results are identical in both the internal and external view. The memory-loaded modules are also the same with the same base address.

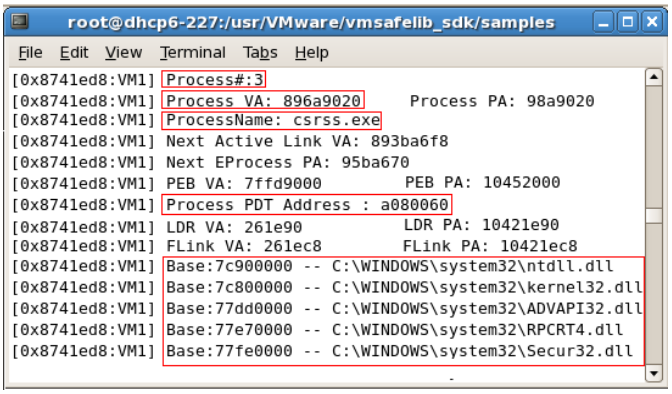

Figure 8. External view of the running csrss.exe process using CloudSec.

Figure 9 shows screenshots of the internal and external view of the SSDT using WinDbg and *CloudSec* respectively. The top screenshot is the internal view, and the bottom is the external view. This figure shows the top 10 entries in the SSDT table for brevity. As mentioned before, each entry has a unique number so entries are in order. By comparing the addresses in the internal and external views, we found that our interpretation for the physical memory pages to OS information matched accurately. The SSDT address is the same in both views, which is 0x80504480, and for example, the first SSDT entry address in the two views is 0x805145f6, which is the address of the *NtConnectPort*, because this function system call index is 0. For the other entries, the addresses of entries in both views are the same.

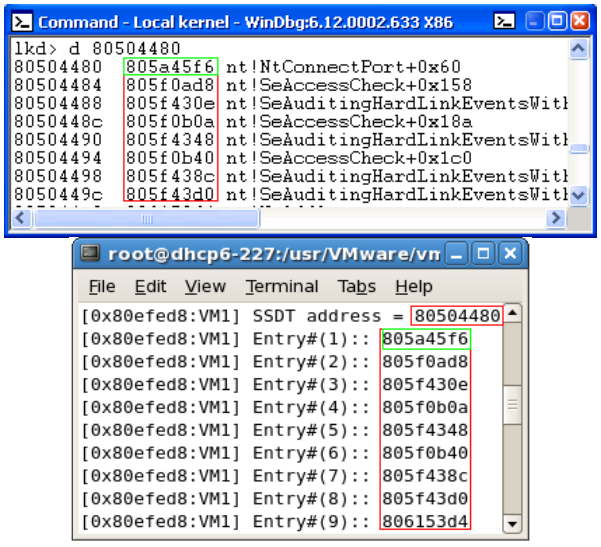

Figure 9. External and Internal memory views of SSDT.

VI. Discussion

To date, nearly all VMI research had been based on the Xen hypervisor. The Xen hypervisor itself is small (approximately 100 KLOC) relative to other hypervisors. However, Xen relies mainly on a Domain0 VM which is part of the TCB. This results in a large footprint hypervisor, increasing the possibilities of vulnerabilities and exploits that can occur. VMware ESX has a smaller footprint than Xen because the ESX hypervisor doesn't depend on any external components. VMI research that is based on the VMware ESX environment is not seen much in research because the ESX hypervisor is not open source like Xen.

Our experiments with *CloudSec* have shown that we can get rich, high-level OS information about any kernel data structure from outside the OS with no monitoring support additions inside the VM. We can also get further information about the kernel data structures, not just limited to the structures used in our example implementation. *CloudSec* detects data hooks rootkits *e.g.* SSDT hooking, process hiding and DLL injection rootkits but does not currently apply any defence mechanisms. We are extending *CloudSec* to provide pre-emptive protection for hosted VMs from kernel and memory rootkits, based on our architecture that has the ability to reconstruct any kernel data structure with their running instances.

CloudSec builds all the necessary information to bridge the semantic gap and install monitoring triggers once a VM is booted-up. The performance overhead of *CloudSec* depends on the interception overheads of the vCompute APIs and the amount of processing performed by our monitoring code. VMsafe APIs (including vCompute) reduce performance overhead because security inspections are processed in the hypervisor kernel [27]. To calculate the elapsed time for running our code, we called *CloudSec* code functions 1000 times and calculated the average time taken. Figure 10 shows the elapsed CPU clocks and time in milliseconds (msecs) for 1000 iterations for both the EPROCESS and SSDT construction and listing functions. Time is calculated by dividing the CPU clocks for each function calls by the CLOCK_PER_SEC (processor dependant variable). Listing the processes (including processes details and loaded modules) consumes about 0.96 msecs, and locating and reading the SSDT table consumes 0.03 msecs. These results indicate that the performance overhead is minor and thus enables real-time, monitoring of kernel data structures. The normal boot-up time for VMs hosted in our platform is 12 seconds, being the time from the time we press power on until the OS is loaded. *CloudSec* reads the SSDT entries before the OS is completely loaded by 7 seconds. As the VM is powered on, *CloudSec* queries the hypervisor to find out the version of the running OS kernel, and starts locating and reading the SSDT. For listing the running processes, *CloudSec* waits until processes are loaded into memory. Once a process is loaded, *CloudSec* reads the process immediately. These results show our external extraction process is very lightweight.

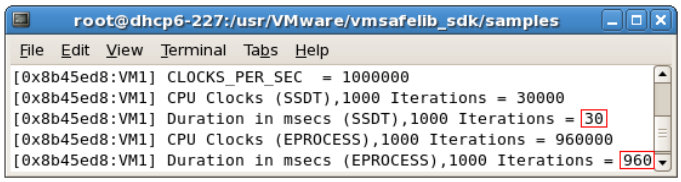

Figure 10. CloudSec Performance Overhead.

VII. SUMMARY

IaaS cloud platform requires new virtualization-aware security solutions that have the ability to externally monitor and protect the hosted VMs externally. In this paper, we presented *CloudSec*, a solution that provides active, transparent and real-time security monitoring for multiple concurrent VMs hosted on a cloud platform. *CloudSec* utilizes VMI techniques to monitor the hosted VM's memory externally, without installing any monitoring code inside the hosted VMs. It then constructs an accurate external semantic view of the VM's kernel data structures instances from hypervisor level. *CloudSec* is deployed as a virtual appliance to enable reliable and secure deployment for the monitoring software. We have implemented *CloudSec* on a VMware ESX hypervisor using the vCompute API. Our experiments showed that the constructed external memory view by *CloudSec* is identical to the internal view of the hosted VMs' kernel structures. We have shown that the performance overhead of traversing the physical memory to build the kernel data structures is acceptably low to support real-time external VM monitoring.

ACKNOWLEDGMENT

We would like to thank VMware Inc, especially the VMsafe team and Amitabh Chakrabarty, for providing the research license for the VMsafe APIs. We also thank Swinburne University of Technology for their scholarship support for the first and fourth authors.

REFERENCES

[1] Amani S. Ibrahim, James Hamlyn-Harris and John Grundy, "Emerging Security Challenges of Cloud Virtual Infrastructure," in Proc. of The Asia Pacific Cloud Workshop (co-located with APSEC2010), Sydney, Australia, 2010.

[2] Tal Garfinkel, and Mendel Rosenblum, "Virtual Machine Introspection Based Architecture for Intrusion Detection," in Proc. of The Network and Distributed Systems Security Symposium, 2003, pp. 191-206.

[3] VMware, "Virtual Appliances: A New Paradigm for Software Delivery," 2009, Available: http://www.vmware.com/files/pdf/vam/VMware_Virtual_Appliance_Solutions_White_Paper_08Q3.pdf.

[4] Tzi-cker Chiueh, Matthew Conover, Maohua Lu and Bruce Montague, "Stealthy Deployment and Execution of In-Guest Kernel Agents," in Proc. of The Black Hat, USA, 2009.

[5] Bryan D. Payne, Martim Carbone et al., "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in Proc. of The IEEE Symposium on Security and Privacy, Oakland, CA, 2008, pp. 233-247.

[6] Bryan D. Payne, Martim Carbone et al., "Xenaccess Library," 2009, Available: http://code.google.com/p/xenaccess/.

[7] Fabrizio Baiardi, Dario Maggiari, et al., "PsycoTrace: Virtual and Transparent Monitoring of a Process Self," in Proc. of The 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009, pp. 393-397.

[8] Fabrizio Baiardi, and Dario Maggiari et al., "Transparent Process Monitoring in a Virtual Environment," Electronic Notes in Theoretical Computer Science, vol. 236, pp. 85-100, 2009.

[9] Flavio Lombardi, and Roberto Di Pietro, "KvmSec: a security extension for Linux kernel virtual machines," in Proc. of The ACM symposium on Applied Computing, Honolulu, Hawaii, 2009, pp. 2029-2034.

[10] Monirul I. Sharif, Wenke Lee, Weidong Cui, et al., "Secure in-VM monitoring using hardware virtualization," in Proc of The 16th ACM conference on Computer and communications security, Chicago, Illinois, USA, 2009, pp. 477-487.

[11] Brian Hay, and Kara Nance, "Forensics examination of volatile system data using virtual introspection," Journal of ACM SIGOPS Operating Systems Review, vol. 42, pp. 74-82, 2008.

[12] Kara Nance, Matt Bishop, and Brian Hay, "Virtual Machine Introspection: Observation or Interference?," Journal of IEEE Security and Privacy, vol. 6, pp. 32-37, 2008.

[13] Sina Bahram, Xuxian Jiang, Zhi Wang, et al., "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," in Proc. of The 29th IEEE International Symposium on Reliable Distributed Systems, New Delhi, India, 2010.

[14] David A. Solomon, "The Windows NT Kernel Architecture," Computer, vol. 31, pp. 40-47, 1998.

[15] McAfee Inc., "McAfee Firewall Enterprise," Available: http://www.mcafee.com/us/products/firewall-enterprise.aspx.

[16] Amir Dastjerdi, Kamalrulnizam Abu Bakar, et al., "Distributed Intrusion Detection in Clouds Using Mobile Agents," in Proc. of The Third International Conference on Advanced Engineering Computing and Applications in Sciences, 2009, pp. 175-180.

[17] Jia Tiejun, and Wang Xiaogang, "The Construction and Realization of the Intelligent NIPS Based on the Cloud Security," in Proc. of The 1st International Conference on Information Science and Engineering, Nanjing 2009, pp. 1885 - 1888.

[18] Mohamed Almorsy, John Grundy et al., "An analysis of the cloud computing security problem," in Proc. of The Asia Pacific Cloud Workshop (co-located with APSEC2010), Sydeny, Australia, 2010.

[19] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, et al., "Cloud security is not (just) virtualization security: a short paper," in Proc. of The ACM workshop on Cloud computing security, Chicago, Illinois, USA, 2009, pp. 97-102.

[20] Microsoft, "Windows Symbol Packages," Accessed on Dec 2010, Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463028

[21] Trusted Computing Group, "Trusted Platform Module," Accessed on March 2011, Available: http://www.trustedcomputinggroup.org/developers/trusted_platform_module/.

[22] Intel, "Intel Trusted Exceution Technology," Accessed on March 2011, Available: http://www.intel.com/technology/security/.

[23] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1," 2011, Available: http://www.intel.com/Assets/PDF/manual/253668.pdf.

[24] Mihir Nanavati, and Bhavesh Kothari, "Hidden Processes Detection using the PspCidTable," MIEL Labs2010, Available: http://helios.miel-labs.com/downloads/process_scan.pdf.

[25] B. Dolan-Gavitt, et al., "Robust signatures for kernel data structures," presented at the Proceedings of the 16th ACM conference on Computer and communications security, Chicago, Illinois, USA, 2009.

[26] Microsoft, "Debugging Tools For Windows," Accessed on Dec 2010, Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463009.

[27] Seongbeom Kim, Banjot Chanana, et al., "An Introduction to VMsafe . Architecture, Performance, and Solutions," VMware, Inc., 2009, Availbale: http://www.vmworld.com/docs/DOC-2406.