Journal of Cloud Computing
a SpringerOpen Journal

REVIEW Open Access

Virtual machine introspection: towards bridging the semantic gap

Asit More* and Shashikala Tapaswi

Abstract

Virtual machine introspection is a technique used to inspect and analyse the code running on a given virtual machine. Virtual machine introspection has gained considerable attention in the field of computer security research. In recent years, it has been applied in various areas, ranging from intrusion detection and malware analysis to complete cloud monitoring platforms. A survey of existing virtual machine introspection tools is necessary to address various possible research gaps and to focus on key features required for wide application of virtual machine introspection techniques. In this paper, we focus on the evolution of virtual machine introspection tools and their ability to address the semantic gap problem.

Keywords: Virtual machine introspection

Introduction

Security and safety are two principal factors governing future cloud computing research and development. Research in virtualisation technology has fuelled cloud computing growth and directly contributed to its development. Our work of reviewing virtual machine introspection (VMI) techniques predominantly targets cloud computing enabler virtualisation, with security as its major concern.

VMI is a technique initially suggested by [1] in 2003. They defined VMI as a method of inspecting a Virtual Machine (VM) from the 'outside' for analysing the software running on the machine. Over the past few years, VMI has seen concrete contributions, and various methods have been suggested to inspect VM data from the outside. The difficulty in interpreting the low level bits and bytes of a VM into a high level semantic state of a guest Operating System (OS) is called the "semantic gap problem" [2]. To interpret the low-level binary state information about the VM, a virtual machine monitor (VMM) must incorporate knowledge of the hardware architecture or guest OS [3].

In majority of VMI techniques, VM which is observing the results of introspection is different than the VM being introspected. The main motivation behind VMI is to analyse every possible change taking place in a guest OS due to the deployment of a given set of code over its entire lifecycle. It is also possible that in presence of monitoring code, deployed code may behave differently than its legitimate behaviour. Presence of monitoring code on a guest VM puts some limitations on execution of monitoring code like, VMI code could start after OS being loaded properly and it could continue till guest OS starts its shut down routine. Introspection from outside the guest VM addresses one or more of the above stated issues. Hence, introspection from different VM is preferred over the other options.

VMI, which has its roots in cloud enabling technology virtualisation, has the potential to change security deployment in cloud environments. The last couple of years have seen considerable progress in exploring various techniques for VMI. Path-breaking applications of VMI have been developed in relation to cloud security, cloud intrusion detection and cloud access management. There are evidences of intrusion detection systems and rootkit detection methods which have been proved effective only because of use of VMI in their implementation [4-6].

The contributions of this paper are as follows:

- It thoroughly inspects VMI techniques and outlines their advantages and weaknesses.

*Correspondence: asit_5@yahoo.com
ABV- Indian Institute of Information Technology & Management, Gwalior 474015, India

- It summarises various possible attacks and threats to VMI techniques.
- It proposes a VMI technique based on microprocessor architecture features.

We expect the following outcomes from our manuscript. We believe that it will provide a guide for future developers of VMI tools looking to develop various applications for cloud security and malware detection based on VMI. This paper is organised as follows: Section 'An overview' describes the basics of virtualisation and provides an illustration of the semantic gap problem. The later part of this section is dedicated to the taxonomy that we used to classify VMI tools. Section 'Characteristic properties of VMI' reveals the properties of an ideal VMI technique. Section 'Memory introspection' describes memory introspection, Section 'I/O Introspection' defines I/O introspection, and Section 'System call introspection' covers system call introspection. Section 'Process introspection' is dedicated to process introspection, and Section 'Other techniques' describes a range of possible techniques for VMI. Section 'Proposed architecture for VMI' describes the proposed architecture for VMI. Section 'VMI applications & future' outlines some predominant VMI applications. Section 'Security issues in VMI' discusses possible attacks on VMI techniques and the VMI architecture. Section 'Conclusion' presents the conclusion to our survey. Table 1 provides at a glance comparison of VMI techniques reviewed in this document.

An overview

Different terminologies are applied to the virtualisation framework. We adopt the following terminology throughout this paper: A *Guest VM* is a virtual machine running on a given hypervisor. The Guest OS is an OS system running on a particular guest VM. A *Secure VM* is a VM dedicated to security applications. Unless otherwise stated, the guest VM introspection is done through the same secure VM.

Virtualisation & hypervisors

The virtualisation technique is used to create a virtual environment for computing by virtualising hardware, I/O and processors. This virtual environment is possible with the help of a special layer of software named a VM monitor (VMM) or hypervisor. The VMM is the interface between the hardware and the VMs running on the system. Depending on the logical position of VMM in operating system architecture, VMMs are distinguished into two major types.

- Type I Hypervisor
- Type II Hypervisor

Type I hypervisors run directly on available hardware, eliminating the need for other layers, such as an OS, and providing high efficiency compared to its counterpart. Xen [7], VM Ware ESX [8] and Microsoft HyperV [9] are well-known Type I hypervisors. As these hypervisors run directly over hardware, they are also known as "bare metal hypervisors".

Type II hypervisors have the in-between interface of the OS to communicate with hardware. They usually depend on an OS to provide device drivers for hardware interaction. KVM [10], QEMU [11] and the VMWare workstation [12] are well-known examples of this type of hypervisor.

Semantic gap problem

The semantic gap problem in virtualisation was first stated by [2]. To extract meaningful information about the current state of a VM, detailed knowledge of the workings of the guest OS is required. It is very difficult to derive a complete view of a guest OS from outside a guest VM due to the highly dynamic nature of modern OSes. Various features, such as demand paging, parallel computing and multithreading, make the architecture of an OS very complex and volatile. View creation becomes extremely complex. This problem is known as the semantic gap problem. The preliminary aim of VMI is to generate a complete view of a guest VM. Hence, the evolution of VMI has been guided by the question: "How efficiently can the given VMI technique bridge the problem of semantic gap".

Characteristic properties of VMI

VMI is applied in widespread domains. With some listed in Section 'Introduction', Section 'VMI applications & future' details additional applications. It is obvious that some properties of VMI are application domain specific. Still, there are a few important properties that all VMI tools should possess, irrespective of their application domain. Some of the properties are listed below:

- **Minimum performance impact:** The main goal in virtualisation is to share resources between available guests. The implementation of introspection techniques should place as little burden as possible on the operation of the existing system. Introspection techniques should not place a burden on the hypervisor and real hardware resources.
- **Minimum modifications to hypervisor:** Introspection techniques should work independently and make minimum modifications to the hypervisor code. This is important in the application of VMI during minor revisions and in future versions of the VMM.
- **No modifications to guest OS:** Real-world hypervisors provide support to almost every possible

Table 1 Comparison of VMI techniques

Category	Technique	Location of code			VMM transparency	VMM alteration	Guest support	Advantages
		Guest VM	Secure VM	VMM				
Memory introspection	Using Xen Libraries	N	Y	Y	No	Required	PV Guests	Safety of VMI code
I/O Introspection		N	N	Y	No	Required	All Types	Driver and I/O access inspection
System call introspection	Using VT support	N	Y	Y	No	Required	All Types	Processor support makes introspection less complicated
	By Hardware Rooting	N	Y	N	No	Required	All Types	Protection from DKSM attacks
Process introspection	Using Hooks	Y	Y	Y	Yes	Required	All Types	Reverse remote control possible
	Using Shadow Page Tables	Y	N	Y	Yes	Required	All Types	Trusted Introspection code execution
	Using CFG	Y	N	Y	Yes	Required	All Types	Novel approach for code malfunction detection
Other techniques	Code Injection	Y	N	Y		Required	All Types	Secure and less prone to attacks
	Function Call Injection	Y	Y	Y	No	Required	All Types	Novel approach
	Page Flag Inspection	Y	Y	N	No	No	PV guests	Detects packed & encrypted malwares
	Process Out-grafting	Y	Y	Y	Yes	Required	All Types	A novel approach
	Live Kernel Data Redirection	Y	Y	Y	Yes	Required	All Types	Choice of selection for introspection programme
Proposed technique	Event Injection	Y	N	Y	Yes	Required	All Types	Secure & almost every introspection code can be used

OS as a guest. If the introspection code needs to be modified for each guest OS, its widespread applicability becomes questionable. Even minor revisions and periodical patches to a particular OS may create problems.

- **Transparency in operation:** The operation of VMI technique should be transparent to the hypervisor, the guest VM and any program on the guest VM.
- **Hypervisor independence:** The VMI technique should not depend on any exclusive feature of the hypervisor architecture. It should be applicable to any type of hypervisor, irrespective of its implementation technology.
- **No side effects:** The implementation of introspection tools should not generate any unwanted results, which may lead to malicious behaviour of system components. VMI tools should also not produce any extraordinary results, which may lead in the detection of its existence.
- **Security of monitoring component:** VMI modules can be located in the hypervisor, guest VM or secure VM. These modules must be secure from external attacks. If a VMI module is present in a guest VM, special protection must be provided to preserve its integrity.

Taxonomy of VMI

There are different possible events related to a guest VM and a guest OS running on it. These events can be grouped to have introspection at various degrees A brief overview is given below:

1. Memory Introspection
2. System Events Introspection
    (a) System Call introspection
    (b) Interrupt Requests Introspection
    (c) I/O Device Driver Introspection
3. Live Process Introspection

Based on the above-mentioned classification, we have divided the introspection techniques according to different types. Figure 1 describes a possible taxonomy for VMI.

Memory introspection

Memory introspection deals with live memory analysis. When the OS is running, all the important data structures are in the main memory. The main memory contains process control blocks (PCBs), registry entries, loadable kernel modules, kernel data structures and page tables, etc. The main memory also contains pages related to data segments and code segments of the process being executed. Information related to the OS can be retrieved by examining the content of the main memory. The majority of malware analysis tools inspect program behaviour by examining main memory contents of the given program. A variety of VMI techniques are available to access the main memory of a guest VM from a secure VM. These can be used for tasks such as intrusion detection or process analysis of the guest VM. A range of memory-based VMI techniques are summarised in the remainder of this section.

Introspection using Xen libraries

A guest VM can be introspected from a privileged domain (Dom 0) associated with a Xen hypervisor [7]. Dom 0 is a control domain of Xen, and it provides access to every data structure, driver and library implemented by Xen. *libxc* is a control library for Xen. The memory of the guest VM can be monitored using the function *xc_map_foreign_range()*, which belongs to the same library. A special high-performance disk driver named *blktap* made for Xen's paravirtualised guest VMs monitors disk access and data transfer. In the case of a guest VM, memory access needs to address translation from the virtual to the physical address and then again from the physical to the machine address. Xen has implemented shadow page tables for the same purpose. The introspection of a paravirtualised guest VM is possible using libxc, a blktap driver and the xen store library .

Xen_Access [13] is a good demonstration of memory and disk introspection with the Xen hypervisor. The introspection code remains safe, as it resides in a secure

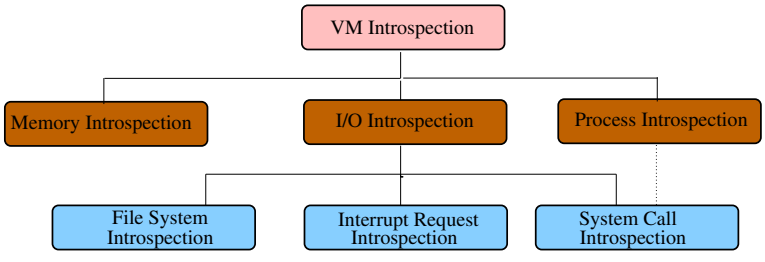

Figure 1 Classification of VMI techniques.

VM (Dom 0). However, there is a possibility that malware could change the kernel data structure, causing Xen_Access to produce irrelevant results. Xen_Access has achieved performance improvement in memory access by caching Xen Store mapping on a least recently used (LRU) basis, which is analogous to translation look-aside buffers (TLB). Xen Access provides very limited traces of file access, with only the creation and the deletion of a file traceable. Xen_Access also provides very limited support for hardware virtualisation machine (HVM) domains. This restricts its widespread application to OSs.

I/O Introspection

I/O introspection deals with device drivers and other utility hardware communications. dAnubis [14] is the technique suggested for VM introspection from outside of it. This method is the successor of Anubis and exclusively monitors Windows device drivers and kernel behaviour. It generates a detailed report of malware activities on machines running Windows. It is claimed that it detects kernel patching, call hooking and direct kernel object manipulation (DKOM). For kernel-side malicious code, the analysis needs to be performed at a higher privileged level than the privilege level of the kernel itself. It is only possible via out of the VM analysis as a hypervisor is available at the higher privileged level than a kernel of the guest OS.

The focus of dAnubis is on monitoring all communication channels between the rootkit (device driver affected by a rootkit) and the rest of the system. All necessary information, such as exported symbols, data structure and layouts are extracted from the Windows OS. To reconstruct the necessary information, kernel symbols and data structures are extracted from the Windows OS by using a technique mentioned by [15]. dAnubis has been proposed for detailed analysis of rootkits. This tool is capable of conducting memory analysis and detecting attacks, such as call table hooking, DKOM, runtime patching and hardware access.

Stimulator: Malware is activated by some triggering event. dAnubis has a stimulator engine that generates such events. dAnubis works only on Windows OS. It is a malware analysis engine and not a malware detection engine.

System call introspection

The system call is a request by program for service from the kernel. The service is generally something that only the kernel has the privilege to perform, such as doing I/O. Hence, system calls play a very important role in events such as context switching, memory access, page table access and interrupt handling. In case of the virtualisation technology (VT) support [16] enabled processors, the transition of a guest VM to the hypervisor and vice versa is managed by special system calls. To maintain the integrity of the system, specific system calls are banned from execution by a guest VM.

Introspection using virtualization support

It has already been shown [13,17] that VT microprocessor support features can be used for introspection activities. Useful information related to guest VM implementation can be retrieved by monitoring the VM control structure (VMCS) of the processor. This region is dedicated to handling virtualisation support. Intel's VT-supported microprocessors have two modes of operation: VMX root operation and VMX non-root operation. The VMX root operation is intended for hypervisor use. The VMX non-root operation provides an alternative IA-32/64 environment controlled by a hypervisor. There are two transitions associated with these two operation modes: 1) a transition from the VMX root operation to the VMX non-root operation (i.e. from the hypervisor to the guest VM) called *hypervisor entry* and 2) a transition from the VMX non-root operation to the VMX root operation (i.e. from a guest to the hypervisor) called *hypervisor exit*.

The CR3 register is responsible for holding the page table address for currently running processes. Access to the CR3 register by the guest VM causes hypervisor exit. The hypervisor-based VMI module handles the hypervisor exit. A communication channel is opened between the VMI module in a secure VM and the VMI module in the hypervisor by setting a covert channel for communication. The channel is set through the VMCS region using an I/O bitmap. On receiving the CR3 change signal, the VMI module obtains access to the page tables. This enables tracking of current processes that are being executed.

Aquarius demonstrates the application of Intel VT and AMD technologies for effective out of VM introspection. Bit Visor [18] hypervisor was used for introspection purposes. Some modifications were made to the Bit Visor to inspect the guest's system call activities.

Introspection by hardware rooting

An introspection approach that relies only on guest OS knowledge might face attacks that change the architecture of the guest OS. Hardware rooting offers a solution to this type of attack, preventing malware from ever changing the structures of virtual hardware. Any trace which begins from hardware assistance has very less probability of such attacks. The hardware rooting mechanism thwarts possible kernel data structure attacks mentioned in Section 'Kernel structure manipulation'.

Hardware rooting exploits system call trapping using an interrupt descriptor table register (IDTR) and an interrupt descriptor table. The IDTR value is set by the processor. Genuine interrupt descriptor table gets accessed using system call trapping. Every time the value of the CR3

register needs to be changed, an interrupt needs to be generated. The VMI method traces this interrupt to detect process switching. In this way, the value of the CR3 register, along with the value of the first valid entry in the corresponding top-level page directory, is accessed. The value of CR3 register is unique for every process. It helps to identify the required process executing inside the VM.

Nitro [3] is another tool based on the hardware rooting technique. Nitro claims to work on any operating system and have defined rules for OS portability. The unique feature of Nitro is its rule set. Simple changes in a rule set enable it to work with almost any available OS. These rules have provision for determining locations of system call arguments, variables, etc. The locations of these arguments is variable according to the implementation of the OS. Generally, they reside in stack or CPU registers.

Nitro has modified QEMU [11], which is a monitor for KVM VMM [10]. All administrative commands to Nitro are given through the same monitor that is used by the KVM hypervisor. It is stealthier to direct kernel structure manipulation (DKSM) [19] types of attacks, as it depends on CPU data structures. Importantly, its performance is dramatically improved compared to its predecessor, Ether [20]. The major drawback of Nitro is that it supports only the x86 Intel 64-bit architecture.

Process introspection

Many application domains of VMI are limited to monitor specific processes. A process could range from any legitimate code, such as API, user application or test code, to malicious code, such as like malware and rootkit. Process introspection should be able to debug any process at any point of time during its entire execution cycle. It should also be able to detect invocation of a specific module or code snippet. Process introspection helps in the analysis of code. Process introspection is also useful for malware behaviour analysis, debugging, etc.

Introspection by hooks

Generally, this type of VMI technique comprises two separate parts. The design goal is to use a guest VM for only a minimum amount of essential code and to use a hypervisor layer or a secure VM for the remaining code. It consists of two modules, a guest module and an out of guest module described below:

- Guest module: It includes hooks for intercepting guest OS events and a small specially crafted trampoline code to pass events signalled by the hooks to the hypervisor. A hook is a jump mechanism, and it is generally associated with OS system calls. This ensures that whenever some system call is invoked by a process, the hook is activated. Hook transfers the control flow of a process to another kernel component named the trampoline. The trampoline is a module that acts as a bridge for communication between hooks in a guest VM and a security driver running in a secure VM. It also receives commands from a secure VM.
- Out of guest module: It resides in a secure VM. This module is responsible for processing signals received from the trampoline. It consists of memory monitoring and Intrusion Detection System (IDS) tools, which inspect processes and memory on receiving calls from hooks. It also consists of various tools that analyse signals and makes decisions about the fate of running processes. Based on the decision, commands are issued to the guest OS to take preventive steps/measures.

The Lares [21] is made up of two distinct modules. High level implementation of the Lares is shown in Figure 2.

The Lares utilises a Xen hypervisor. Lares uses various features of Xen, including split device drivers and memory address translation, to provide a robust and secure introspection tool. It is appropriate for IDS or antivirus software development where immediate reverse action is needed upon detection of vulnerability. The trampoline mechanism distinguishes Lares from other introspection tools. However, the use of trampolines is a security bottleneck of Lares. Even the authenticity of calls generated by hooks is questionable because malware aimed at consuming system resources can invoke multiple false calls. False calls may lead to disturbances in working of legitimate programs.

An excellent feature of Lares is the availability of a reverse path from a secure VM to a guest VM. This feature is absent in almost every other existing VMI technique. The trampoline in Lares makes it possible to send signals to a guest VM resident code. This feature is referred to here as *reverse remote control*.

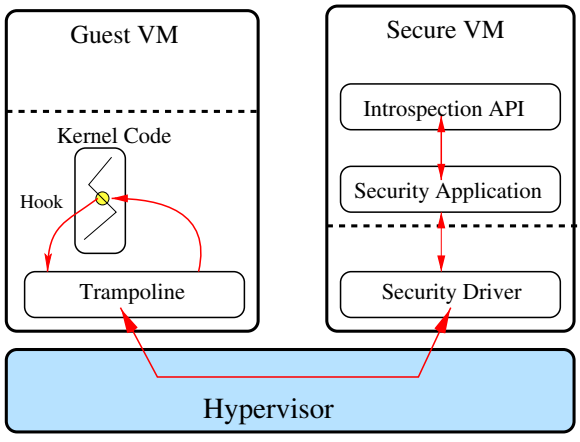

Figure 2 High level view of Lares architecture.

Introspection using shadow page tables

The hypervisor uses shadow page tables to convert a guest VM physical address to an actual machine address. Shadow page tables are accessible from the hypervisor and can be manipulated easily. The introspection code can be secured from guest VM-based applications using shadow tables and Intel VT technology features. Intel's VT support and virtual memory protection can be used to secure the monitoring code.

SIM [22] makes use of the above-mentioned techniques. A protected address space is allocated to a guest VM using memory mapping techniques. All methods and data related to the SIM are located in a special memory region, which is only accessible to the hypervisor. This region includes the following elements: a gate for transferring kernel calls, the SIM code and data, a separate copy of kernel code and data that are only read access and special call invocation checkers, which protect the SIM from attacks. Figure 3 shows the high-level architecture of the SIM. The gate is a special mechanism used by the SIM to enter and exit a protected address space (PAS). A separate copy of the kernel code is retained by the PASs because malware can easily infect kernel libraries. The SIM's introspection code uses its own copy of kernel libraries rather than trusting libraries provided by a guest OS. Hooks are placed within the kernel code to transfer a call made to the SIM module. On invocation of hooks, a hypervisor component of the SIM traces that call, and context transfer is done using the *SIM_SHADOW* page table. A *CR3_TARGET_LIST* is used to switch between page tables. A separate hypervisor level page table named *SIM_SHADOW* is created to replace the original shadow page table. A page table address of this shadow table is replaced inside the CR3 register to allow access to the SIM address space. The guest component of the SIM code is loaded onto the guest VM as a device driver.

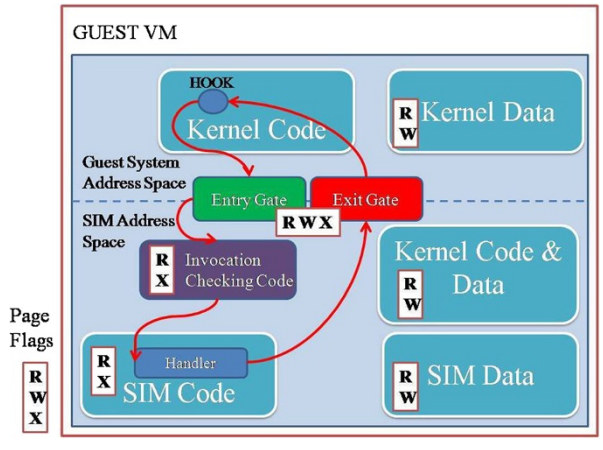

Figure 3 High Level Overview of SIM IN-VM monitoring. Page flags {R-Read, W-Write, X- Execute}.

SIM ensures that no code from a nontrusted address space can be executed while introspection is ongoing This method proved to be a milestone in VM monitoring. Robustness and efficiency are the main advantages of IN-VM monitoring tools.

Introspection using CFG

Another technique known as PsycoTrace, which monitors the processes running on a guest VM, was introduced by [23]. PsycoTrace [23] is a unique method that utilises context-free grammar (CFG) for process activity monitoring and detecting malware attacks. The technique consists of two phases.

The first phase makes use of some static tools and acts from inside the guest VM and utilises the guest VM. These tools are responsible for capturing the legitimate workings of the process to be monitored. A CFG for processes is generated using a *grammar generating algorithm* developed for PsycoTrace. The CFG was developed according to custom-made rules (e.g. system call invocation is considered a terminating symbol). Bison [24] was modified to use C code and system calls as an input and to generate a CFG for a given process.

In the next phase, the kernel of the guest VM is modified and injected with a module named HiMod. Himod is responsible for monitoring system calls generated by a given process. It stores parameters of every system call and notifies the analyst module. The analyst module is associated with a secure VM. Communication between the HiMod and the analyst module takes place via a communication channel. The analyst module validates every system call with CFG to detect malware infection.

Although PsycoTrace has a very innovative way of detecting malware attacks, it has some weaknesses. It is not capable of handling processes that use multiple threads, and the kernel modification code is not well secured from detection and attacks. PsycoTrace, on successful malware detection, lacks malware counter-defence mechanisms. However, PsycoTrace has zero possibility of false positives, which is an achievement. The major drawback limiting the use of PsycoTrace is its initial run, during which the source code of the process has to be monitored.

Other techniques

It is very difficult to classify some VMI techniques in the categories mentioned above. Although some have the capabilities to introspect two or more regions, few have the additional capability to introspect system calls and introspect interrupt requests from devices. This is possible with hypervisors like Xen that use a special data structure called an event channel for passing interrupts and system calls and techniques such as process monitoring

of system calls and memory. These abilities of hypervisors help in monitoring allied fields.

Introspection using code injection

Introspection is possible by implanting an introspection process (monitoring code) inside a guest VM with the help of the hypervisor. This implanted process is hidden under existing legitimate process. This technique is similar to camouflaging. The system consists of a victim process, which is used as a camouflage to hide the monitoring process. The victim process is a process or any user program that is used to replace itself by introspection process. The introspection process is a special program capable of executing certain code, which inspects system variables, parameters and the environment as per the introspection needs.

This introspection process resides in the address space of a secure VM. The hypervisor monitors every context switch to detect the loading of the victim process. On detection of the context switch to a desired victim process, it replaces all necessary pointers, such as the start processor instruction pointer (SIP). It ensures that instead of running the victim process, the monitoring code is initiated and run on a guest VM.

The memory required by the introspection process is provided by a secure VM at runtime. This ensures the address space of the monitoring process is hidden from processes running on the guest OS. It also ensure that the address space cannot be detected by malware programs running on the victim machine.

Although introspection using code injection looks promising, this method has the potential to alert malware that it is being monitored due to the reasons outlined below.

1. Every monitoring process is given explicit root privilege, enabling it to monitor all user-level applications.
2. The monitoring process exits on the request of the hypervisor or the secure VM. This is achieved by the hypervisor setting a control bit in a covert channel created exclusively for message passing.
3. An unkillable flag is used in the monitoring process, so that it cannot be killed in between the introspection process. This flag is set only for *init processes*.
4. *Fork* calls are blocked during execution of the monitoring process.

Gu et al. [25] implemented a similar technique and took various precautions to ensure the security of this technique. In the approach they used, all OS libraries needed by the monitoring process are compiled statically to avoid the use of guest VM libraries, which are possible baits for a malware. There is no restriction on the choice of monitoring processes: It can be a malware catcher or user code, which, in turn, can inspect processes running inside a guest VM. This achievement is remarkable.

The introspection technique rectified almost all security vulnerabilities detected with the process implantation technique Virtuoso [25]. Virtuoso restricts the selection of the monitoring process, and it can only use tools provided by the OS [25]. The advantage of using Virtuoso is that the user needs very limited knowledge of OSs, and little effort is required to build OS-specific introspection routines. The process implantation technique is divided into two phases.

The first phase is the training phase in which the monitoring process is executed repeatedly. This phase is small and runs parallel on the guest programme and calculates the data required by the monitoring process. A slicing algorithm and a trace logging algorithm are used to analyse this monitoring code for different loops, jumps and conditional statements. These algorithms reproduce the monitoring programme, with almost the exact instruction code sequence. Whenever introspection is required, this newly created code segment is mounted on guest VM environment by the VMM.

According to the authors [25] , Virtuoso has been tested on various OSs, such as Windows XP SP2 (kernel version 5.1.2600.2180), Ubuntu Linux 8.10 (kernel version 2.6.27-11) and Haiku R1 Alpha 2. However, it has a serious drawback: It requires continuous human intervention. In addition, if any loop or conditional flow was not exercised during training, there are chances of generating instruction sequences from such loops/conditional flows, which may lead to ambiguous execution. Moreover, the slicing algorithm cannot deal with interrupts, page faults and external references to remote addresses. However, Virtuoso is secure and much less susceptible to malware detection and attacks. Repeated execution of the training phase has shown excellent results in monitoring code generation.

Introspection with function call injection (FCI)

Function call injection is the amendment to code injection technique. This method utilises the APIs of secure VM OS to ensure the security from code manipulation. When the introspection application residing in the guest VM is called, the hypervisor and the introspection mechanism patch these calls with equivalent function of a secure VM. This makes this technique applicable to almost every OS having APIs for monitoring. It removes the need for hooks inside the guest and a trampoline. The API is called from the secure VM, thereby strengthening the overall security of function call injection.

Function call injection monitors guest data structures and API locations. By pausing the Virtual Central Processing Unit (VCPU) state, a special jump is introduced

to secure the API of the VM. This API is the code for VM introspection. As the API resides inside a secure VM, there is no possibility of malware infecting the API. This ensures that the monitoring code of the API has access to the data structures of the guest VM. This monitoring code generates the introspection results in a secure VM.

Before invoking the monitoring process, VMM runs another process named localised shepherding. The role of the localised shepherding process is to ensure the integrity of API monitoring. The shepherding process avoids switches in between the execution of the monitoring process. It is responsible for atomic execution of the monitoring process.

Syringe [26] is based on the function call injection technique. It utilises a VMWare ESX [8] server platform and the introspection tool VMWare VMSafe. VMSafe has a unique ability to debug guest VM execution during Syringe implementation. Syringe provides flexibility in terms of OSs and the selection of introspection tools. Syringe has no possibility of dynamic code outrage unlike its sibling Virtuoso. Syringe places a single VCPU restriction on guest VMs because placing multiple VCPU restrictions raises the possibility of the code being detected by malware. Keeping performance degradation in mind, atomic execution of the monitoring process is not always favourable.

Introspection using page flag inspection

The dependence of process implantation technique [27] on APIs of OS for introspection may lead to limited access to guest information. Malware that is either encrypted or packed (compressed) is very difficult to detect. Packed malware is generally stored in data pages as user data. Malware that resides in data pages will need to be page faulted, and NX flag[a] (in the case of x86 and DX in the case of AMD) needs to be set to make such pages executable.

Maitland [28] uses the Xen store utility and page flags for accessing NX flags. Maitland observes each page fault and makes these pages accessible to a security VM. The secure VM is equipped to detect malware signatures and inspects the shared pages for symptoms of malware. Maitland uses a split device driver utility, which it uses for paravirtualised guests of Xen. The application of Maitland to HVM (fully virtualised) guests requires major reforms in split device drivers. This restricts its use on Windows-based guest VMs. In turn, Maitland needs very little changes to the VMM, and its monitoring code for page faults consumes little resources. Even the code running on the guest VM incurs very little overheads. However, this code is easily detectable by malware, something that raises serious concerns with regard to its widespread adoption.

The design goal of Maitland is to develop a lightweight introspection tool. Maitland focuses on the detection of encrypted and packed malware over the cloud on VM guests.

Introspection using process out-grafting

Many past VMI solutions are sensitive to version of OSs, with even a simple patch for an OS having an adverse effect on their operation. Numerous attempts have been made to inject a function/process in guest VMs. Process out-grafting proposes a solution for monitoring specific processes from a number of guest VM processes. The approach used here is exactly the reverse of that used conventionally. Instead of grafting the monitoring process running on a guest VM, process out-grafting relocates the specific process on-demand from a guest to a secure VM. The advantage of out-grafting is that monitoring tools do not need any modification. They essentially view it a normal process running in a secure VM.

The way in which "on demand grafting" works is very interesting. It monitors the state of the VCPU of a guest VM for user mode execution. Out-grafting begins when VCPU is switched to user mode. Process grafting can be achieved by transfer of the execution context (e.g. registers) and memory page frames. For memory page frames, it depends on memory virtualisation support by the processor. Process grafting is achieved by directly marking an NX flag of the corresponding pages in the EPT[b] kernel state of a guest VM that is maintained during out-grafting, and page tables are synchronised with a secure VM.

Srinivasan et al. [29] used a similar method to process out-grafting called mode-sensitive split execution for introspection. The system calls by the out-grafted process are redirected back to the guest VM. The same is achieved by coordinating between the stub residing on the guest VM and a helper module residing on a secure VM in a loadable kernel module (LKM) mode. Only user mode instructions are executed in a secure VM, and all kernel mode instructions are redirected back to the guest VM. The file system of the guest VM is in read only mode to aid monitoring activity. This helps in tracing the system call execution of the out-grafted process.

Although the guest process is out-grafted, a secure VM is needed to handle system call migration, page fault handling and kernel mode execution. All these events are executed by the guest VM. The *exec_ve* calls from the process have to be executed by the guest VM. In short, only user mode execution is monitored by a secure VM. The fate of out-grafting depends upon the efficient application of the NX bit of the process pages. Malware that could mask the NX bit could easily evade process out-grafting technique. Another disadvantage of process out-grafting is that if the kernel of a guest VM is compromised, then entire process becomes vulnerable. Nevertheless, natural support to any monitoring tool without any modification is remarkable.

Introspection using live kernel data redirection

As mentioned in Section 'Introspection using code injection', introspection using code injection has been suggested as a novel approach for VMI. However, it is not fully automated, and it requires the intervention of human experts. In the case of code injection, a monitoring code is generated by repeated analysis. In contrast, in kernel data redirection, the monitoring code remains fixed, and the data required by the monitoring code are provided by the corresponding guest VM that is to be introspected.

The concept behind kernel data redirection is very simple. It consists of a secure VM with all leading introspection tools installed. The guest VM that is to be introspected shares its memory with a secure VM. This is achieved by mounting the guest memory on a secure VM, using VMM. The secure VM uses its own code to introspect the guest VM using data available from its shared memory.

The VM space traveller (VMST) [30], utilises kernel data redirection. VMST automates the introspection process. The secure VM deploys a separate module named the *syscall execution context identification module*. It is used for identification of introspection-related system calls. Another module named the *redirectable data identification module* is responsible for redirecting the required data of the guest VM to the monitoring process. To retrieve data from the memory of the guest OS, it exploits well-known taint analysis techniques [31,32]. A detailed overview of the VMST is given in Figure 4.

VMST provides a very novel approach to VMI, with secure execution of the monitoring process. It is transparent to use on most Linux kernels. The user needs to select and install a kernel (OS) and then install the memory in read only mode on a secure VM. With VMST, any system API/programme can be used for introspection. It also requires no user intervention, and the user can develop a tailored introspection programme. VMST only depends on a guest VM for memory access. Moreover, unlike Virtuoso, it does not need to mask interrupts and context switches.

Proposed architecture for VMI

We have already seen that VMI has very large scope with regard to security and privacy. VT-based processors provide additional support to hypervisors. Our proposed architecture for VMI is based on Intel VT technology. According to Intel's VT [33] architecture, if the valid bit in the *VM_entry_interruption _information_field* of VMCS is 1, a logical processor delivers an event to a guest OS after all the components of a guest VM state have been loaded. For delivering an event, a vector is used that points to a descriptor of a guest IDT.

This type of entry covers software interrupts, privileged software exceptions and traps. The VMM or introspection software running on a VMM can easily generate these types of interrupts. We have introduced a novel technique, which utilises this interrupt. Figure 5 represents the architecture of our proposed technique.

Our technique is divided into three modules residing at three different physical locations. Their operation is explained as follows:

- Controller Module: This resides on a secure VM. Whenever introspection is required, a command is given through this module. It is responsible for sending requests to a hypervisor-based module. As this module is part of a secure VM, it is part of a trusted computing base (TCB) and thus is secure.
- Injector Module: This module is located in the hypervisor layer. It listens for requests from the controller module. On receipt of an introspection request, it waits for the next VM entry. It detects the next *VM Entry* and introduces an artificial software interrupt by an event injection. The injection takes place after loading the IDT on a guest VM. The module is responsible for putting a particular vector entry in an event injection call. It corresponds to the

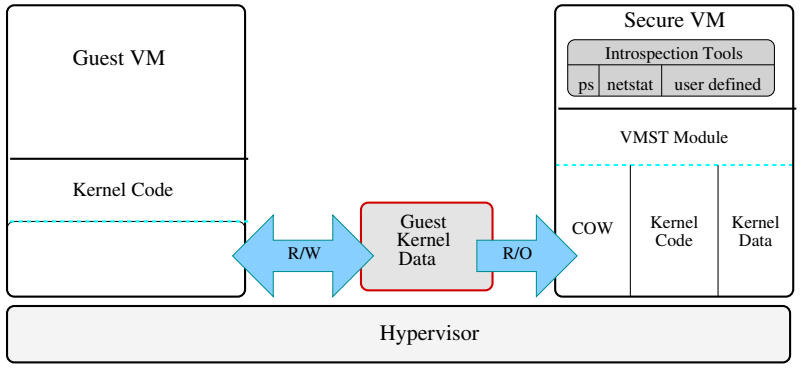

Figure 4 High level overview of VMST architecture.

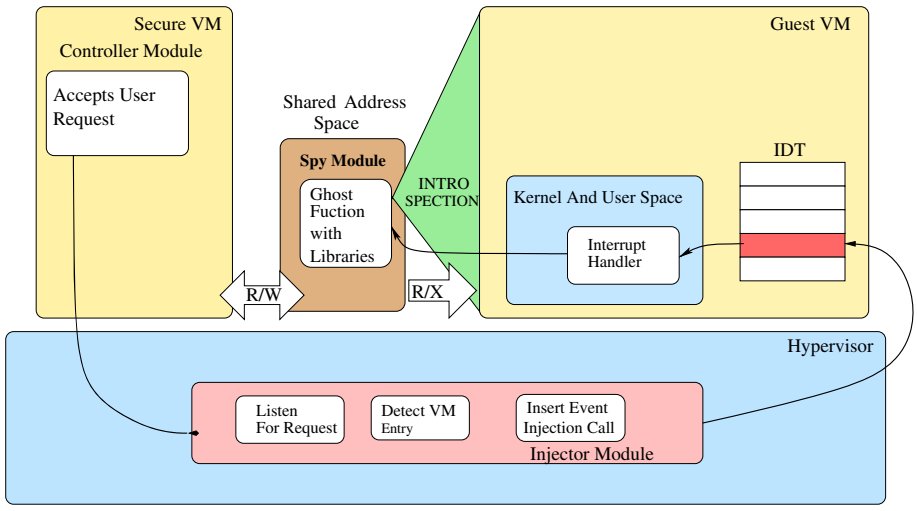

Figure 5 Working schematics of proposed technique.

user- defined interrupt descriptor in the IDT of a guest VM. Thus far, we have defined a single IDT entry, which corresponds to a single interrupt handler routine.

- Spy Module: This module has two parts. The first consists of an installation patch, which installs our own IDT entry and defines the interrupt handler routine for that IDT entry. The second part is hidden from the guest VM. The interrupt handler has a single role: It redirects every call to a ghost function. The *ghost function* contains the introspection code. The address space of the ghost function is different to that of the guest VM. The address space is mounted on the guest VM in read only and executable mode by the hypervisor, only after the controller module has invoked the introspection signal. The function first selects the required introspection type from the available options and then executes it. This function is pre-compiled, and the binary code of the function contains the libraries that are required during execution. This ensures the integrity of the code, preventing tampering . The results generated by this ghost function are saved in another part of the address space shared with the guest VM.

Salient features of our technique are as follows:

- Minimum Performance Impact: This VMI technique is invoked on demand i.e. if the valid bit in the VM entry interruption information field in VMCS region is 1, a logical processor delivers an event to a guest OS after all the components of a guest VM state have been loaded. This will trigger the Spy module and it will start the introspection. This ensures that no script or agent will be running on hypervisor or inside the guest VM for the entire lifecycle of VM. The code will be invoked on-demand and it will terminate on completion of its execution.
- Minimum hypervisor modification: The technique is based on Intel VT technology and solely depends upon it for functioning. Hence requires very few modifications to hypervisor.
- Transparency in operation: This technique makes very few changes to the hypervisor and also do not make any change in guest OS which makes this technique transparent in operation.
- No side effects: The technique does not produce any unwanted results and outputs. The laboratory testing revealed that the execution of guest OS with and without VMI technique had no effect on a hypervisor and guest OS execution. This ensures that there will not be any unwanted site effect on existing setup.
- Security of monitoring code: Our VMI technique is divided in three parts. Controller and Injector module works from Secure VM and Hypervisor respectively. These two modules are never exposed to Guest VM and to the entities inside it (i.e. Softwares, applications and even malware running in Guest VM). Spy Module interacts with Guest VM and runs various custom scripts as per the user need, on Guest VM. It is stored in a separate memory area which is not part of the address scape of a Guest VM. This ensures that entities on Guest VM cannot implicitly access, modify this introspection code, making it secure.

VMI applications & future

Malware detection

Day by day, malware detection is becoming a very crucial task, with advanced malware development strategies. The detection of encrypted malware is very challenging. Maitland [28] is a VMI-based development effort to detect encrypted malware. There is a new breed of malware, which successfully hides itself, when it becomes aware of malware detection code running on the system. VMI code is not usually detectable by malware because it is run from out of a guest VM most of the time . It also makes it easier to detect and monitor malware behaviour without letting malware detect it's being monitored. VMI techniques can help in providing cognitive immunity to systems affected by malware. On detection of kernel modifying rootkit infection, VICI restores the kernel back to an earlier state to provide cognitive immunity [34]. VICI exploits VMI for infection detection and restoration.

Another threat to security is through malware generation capable of attacking not only victim machines but also capable of detecting system execution environment . Such malware is equipped with techniques to detect whether a given OS is running on a VMM or bare hardware. This type of malware attacks VMMs and cloud setups. Such type of malware could also be detected using VMI techniques.

Hidden process monitoring

Many advanced malwares have the capability to hide themselves behind a legitimate OS process. It can cause greater infection, by detaching itself from a process tree or a process node structure maintained by an OS. Such type of malware may be present on the memory of an OS but not detectable by an OS data structure enquiry using legitimate tools, such as *ps*. Hidden process detection and monitoring is possible using Aries [35], which utilises VMI to detect hidden malware process. The application of process monitoring has been extended to different domains, such as web service monitoring [36]. It could be used to record client and service communication over a service oriented architecture (SOA). The interaction trace allows a human or software agent to analyse, replay or debug the code that was executed.

File system/memory management

It is possible to trace every possible activity between a guest OS and hardware using VMI. Lares [13] has already reported preliminary efforts in tracing file system access. Major problem in secondary memory access tracing is, involvement of primary memory (main memory) and the semantic gap problem. The OS loads files from the secondary memory to the main memory . All operations on file are performed at file copy on the main memory. Disk drivers (secondary memory) are included only in create, delete and write back activities. This restriction limits introspection of file system activities. However, the use of disk introspection has benefited by the development of trusted domain development policy [37]. PsycoTrace [38] has tried to bridge the semantic gap involved in file operation introspection. It is capable of providing access rights based on file handling solutions for guest VM users.

Honeypot development

Honeypots were developed with the intention of exposing them to as many attacks as possible. Their aim is to catch malware and log and record features of the malware. An ideal honeypot should record every possible event and activity taking place on it. VMI is a considerable solution for honeypot development. Hiding the honeypot implementation from attackers is a difficult task, and it is a key problem in the majority of honeypot implementations. The productivity of the honeypot depends entirely on it remaining undetected. The chances of detection are much lower when VMI-based monitoring is employed. Similar type of work is possible using VMI. The Qemu Honeypot [39] is an example of using VMI for honeypots. VMI was used in more elaborate ways in honeypot development [40] using a Xen_Access library [13]. Lengyel et al. [40] also provided a good example of the potential use of VMI for honeypots.

Security issues in VMI

It is clear from Section 'VMI applications & future' that the majority of VMI applications are related to the security domain. As stated in Section 'Characteristic properties of VMI', transparency remains a key feature for VMI techniques, specially for those applications of VMI which are developed for the security and privacy. In this section, we have summarise possible attacks on VMI techniques.

Kernel structure manipulation

VMI tools that depend upon memory analysis are victims of kernel structure manipulation. Memory introspection tools derive information on the state of a guest's OS state and related information by analysing the memory of the guest VM. These tools rely upon underlying data structures used by the kernel. In kernel structure manipulation, some changes are intentionally made to kernel data structures. There are three types of modifications possible:

- Syntax manipulation: Certain fields of kernel data structure are modified or changed.
- Semantic manipulation: The semantics of the data structure are changed. Although they might not show any malfunction, the results produced by VMI will be irrelevant.
- Combination of semantic and syntactic manipulation: This type of modification can result in VMI failure.

The above mentioned attacks can be implemented in various ways, as demonstrated previously by [19].

As stated in Section 'Characteristic properties of VMI', ideal VMI techniques should place minimum overheads on the operation on the hypervisor and the involved system. This is important not just for performance but also for security. Recent malware and attack scripts have examined request-response parameters to detect underlying VMI installation. Timing-based attacks have tried to target out-of-bound memory and query system resources to record hypervisor replies . In many instances, the original drivers are faster than the drivers that are patched for the VMI technique. Such changes to the drivers by VMI techniques, may get noticed by malware and could be used as an alarm to take note of presence of VMI technique on VM.

Conclusion

Beginning with an introduction to the semantic gap problem, we have summarised distinct techniques developed for VMI. VMI has grown steadily over the past years. Based on the analysis of VMI techniques presented herein, it appears that the use of VMI is dominant in the security domain. In turn, this makes VMI susceptible to attacks. In the coming years, the security weaknesses of VMI will need to be addressed to enable widespread adoption by the industry.

VMI has great potential in the future development of malware detection tools and intrusion detection systems. Even cloud platforms could benefit from the use of VMI in imposing access right mechanisms.

Existing VMI tools have limited introspection capabilities. No one tool can provide process, memory, file and I/O introspection. In addition, the introspection capabilities of these tools are mostly dependent on the underlying hypervisor architecture. These architectural features are modified or replaced over time, making the application of these tools questionable in the current scenario.

Very limited work has been done to fully introspect HVM guests. Some performance improvement features of HVM guests, such as *pass through drivers*, place limitations on VMI implementation. Introspection using VT support has tremendous potential to enable VMI but requires additional work. The VMI technique based on VT support described in the current paper could be used in the security domain.

Endnotes

[a]The NX bit, which stands for Never eXecute, is a technology used in CPUs to segregate areas of memory for use by either storage of processor instructions (or code) or for storage of data

[b]Extended page table: This page table is part of the memory virtualisation support of the microprocessor. It contains maps of guest-physical addresses to host-physical addresses.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

Both the authors made substantive intellectual contributions to the research and manuscript. AM carried out the survey of the available literature and drafted the manuscript. He is responsible for the overall technical approach and architecture, editing and preparation of the paper. ST provided insight and guidance in developing the VMI technique. She edited and revised the final manuscript. Both authors read and approved the final manuscript.

Acknowledgements

Authors are grateful to the reviewers of this manuscript for their expert advice. Authors are thankful to Indian Institute of Information Technology & Management, Gwalior (IIIT, Gwalior) for support.

Received: 14 March 2013 Accepted: 30 June 2014
Published online: 25 October 2014

References

1. Garfinkel T, Rosenblum M (2003) A virtual machine introspection based architecture for intrusion detection. In: NDSS. The Internet Society, San Diego, California, ISBN 1-891562-15-0. http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/13.pdf
2. Chen PM, Noble BD (2001) When virtual is better than real. In: Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on. IEEE Computer Society, Los Alamitos, CA. p 0133. http://doi.ieeecomputersociety.org/10.1109/HOTOS.2001.990073
3. Pfoh J, Schneider C, Eckert C (2011) Nitro: hardware-based system call tracing for virtual machines. In: Proceedings of the 6th International Conference on Advances in Information and Computer Security, IWSEC'11. Springer-Verlag, Berlin, Heidelberg. pp 96–112. ISBN 978-3-642-25140-5 http://dl.acm.org/citation.cfm?id=2075658.2075669
4. Carbone M, Conover M, Montague B, Lee W (2012) Secure and robust monitoring of virtual machines through guest-assisted introspection. In: Balzarotti D, Stolfo SJ, Cova M (eds). Research in attacks, intrusions, and defenses. Lecture Notes in Computer Science. Springer, Berlin Heidelberg Vol. 7462. pp 22–41. http://dx.doi.org/10.1007/978-3-642-33338-5_2
5. Butt S, Lagar-Cavilla HA, Srivastava A, Ganapathy V (2012) Self-service cloud computing. In: Proceedings of the ACM Conference on Computer and Communications Security. Raleigh, North Carolina. ACM, Raleigh, New York, NY. pp 253–264. http://doi.acm.org/10.1145/2382196.2382226
6. Harrison C, Cook D, McGraw R, Hamilton JA (2012) Constructing a cloud-based IDS by merging VMI with FMA. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on. IEEE, Liverpool. pp 163–169. doi:10.1109/TrustCom
7. Xen (2012) Xen homepage. http://www.xen.org/. Accessed date 15 March 2013
8. Ware VM (2012) Vmware esx homepage. http://www.vmware.com/files/pdf/VMware\discretionary-ESX\discretionary-and\discretionary-VMware\discretionary-ESXi\discretionary-DS\discretionary-EN.pdf. Accessed date 15 March 2013
9. Microsoft (2012) Microsoft hyper -v homepage. http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx. Accessed date 15 March 2013
10. KVM (2012) Linux kvm homepage. http://www.linux-kvm.org/page/Main_Page. Accessed date 15 March 2013
11. Qemu (2012) Qemu homepage. http://wiki.qemu.org/Main_Page. Accessed date 15 March 2013
12. Ware VM (2012) Vmware workstation overview. http://www.vmware.com/products/workstation/overview.html. Accessed date 15 March 2013
13. Payne BD, de Carbone MDP, Lee W (2007) Secure and flexible monitoring of virtual machines. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. IEEE, Miami Beach, FL. pp 385–397. doi:10.1109/ACSAC.2007.10

14. Neugschwandtner M, Platzer C, Comparetti P, Bayer U (2010) danubis – dynamic device driver analysis based on virtual machine introspection. In: Kreibich C, Jahnke M (eds). Detection of Intrusions and Malware, and Vulnerability Assessment, volume 6201 of Lecture Notes in Computer Science. Springer, Berlin Heidelberg. pp 41–60. ISBN 978-3-642-14214-7. doi:10.1007978-3-642-14215-4_3 http://dx.doi.org/10.1007/978-3-642-14215-4_3
15. Jiang X, Wang X, Xu D (2007) Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In: Proceedings of the 14th ACM conference on Computer and communications security, CCS '07. ACM, New York, NY, USA. pp 128–138. ISBN 978-1-59593-703-2. doi:10.1145/1315245.1315262 http://doi.acm.org/10.1145/1315245.1315262
16. Intel (2012) Intel virtualization technology. http://www.intel.com/technology/virtualization
17. Pfoh J, Schneider C, Eckert C (2010) Exploiting the x86 architecture to derive virtual machine state information. In: Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies SECURWARE '10. IEEE Computer Society, Washington, DC, USA. pp 166–175. ISBN 978-0-7695-4095-5. doi:10.1109/SECURWARE.2010.35 http://dx.doi.org/10.1109/SECURWARE.2010.35
18. Bitvisor (2012) Bitvisor hypervisor home page. http://www.bitvisor.org/. Accessed date 15 March 2013
19. Bahram S, Jiang X, Wang Z, Grace M, Li J, Srinivasan D, Rhee J, Xu D (2010) DKSM: subverting virtual machine introspection for fun and profit. In: Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems, SRDS '10. IEEE Computer Society, Washington, DC. pp 82–91. doi:10.1109/SRDS.2010.39. http://dx.doi.org/10.1109/SRDS.2010.39
20. Dinaburg A, Royal P, Sharif M, Lee W (2008) Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and communications security, CCS '08, ACM, New York, NY, USA. pp 51–62. ISBN 978-1-59593-810-7. doi:10.1145/1455770.1455779 http://doi.acm.org/10.1145/1455770.1455779
21. Payne BD, Carbone M, Sharif M, Lee W (2008) Lares: an architecture for secure active monitoring using virtualization. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. IEEE Computer Society, Washington, DC. pp 233–247. doi:10.1109/SP.2008.24. http://dx.doi.org/10.1109/SP.2008.24
22. Sharif MI, Lee W, Cui W, Lanzi A (2009) Secure in-vm monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on Computer and communications security, CCS '09. ACM, New York, NY, USA. pp 477–487. ISBN 978-1-60558-894-0. doi:10.1145/1653662.1653720 http://doi.acm.org/10.1145/1653662.1653720
23. Baiardi F, Maggiari D, Sgandurra D, Tamberi F (2009) PsycoTrace: virtual and transparent monitoring of a process self. In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing. IEEE Computer Society, Washington, DC. pp 393–397. doi:10.1109/PDP.2009.45. http://dx.doi.org/10.1109/PDP.2009.45
24. Bison (2012) Bison - gnu parser generator. http://www.gnu.org/software/bison/. Accessed date 15 March 2013
25. Gu Z, Deng Z, Xu D, Jiang X (2011) Process implanting: a new active introspection framework for virtualization. In: Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems, SRDS '11. IEEE Computer Society, Washington, DC, USA. pp 147–156. ISBN 978-0-7695-4450-2. doi:10.1109/SRDS.2011.26 http://dx.doi.org/10.1109/SRDS.2011.26
26. Carbone M, Conover M, Montague B, Lee W (2012) Secure and robust monitoring of virtual machines through guest-assisted introspection. In: Proceedings of the 15th international conference on Research in Attacks, Intrusions, and Defenses, RAID'12. Springer-Verlag, Berlin, Heidelberg. pp 22–41. ISBN 978-3-642-33337-8. doi:10.1007/978-3-642-33338-5_2 http://dx.doi.org/10.1007/978-3-642-33338-5_2
27. Dolan-Gavitt B, Leek T, Zhivich M, Giffin J, Lee W (2011) Virtuoso: narrowing the semantic gap in virtual machine introspection. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11. IEEE Computer Society, Washington, DC, USA. pp 297–312. ISBN 978-0-7695-4402-1. doi:10.1109/SP.2011.11 http://dx.doi.org/10.1109/SP.2011.11
28. Benninger C, Neville SW, Yazir YO, Matthews C, Coady Y (2012) Maitland: Lighter-weight VM introspection to support cyber-security in the cloud. In: Cloud Computing (CLOUD) 2012 IEEE 5th International Conference on. IEEE, Honolulu, HI. pp 471–478. doi:10.1109/CLOUD.2012.145
29. Srinivasan D, Wang Z, Jiang X, Xu D (2011) Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In: Proceedings of the 18th ACM conference on Computer and communications security, CCS '11. ACM, New York, NY, USA. pp 363–374. ISBN 978-1-4503-0948-6. doi:10.1145/2046707.2046751 http://doi.acm.org/10.1145/2046707.2046751
30. Fu Y, Lin Z (2012) Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: Security and Privacy (SP) 2012 IEEE Symposium on. IEEE, doi:10.1109/SP.2012.40. pp 586–600
31. Chow J, Pfaff B, Garfinkel T, Christopher K, Rosenblum M (2004) Understanding data lifetime via whole system simulation. In: Proceedings of the 13th conference on USENIX Security Symposium - Volume 13, SSYM'04. USENIX Association, Berkeley, CA, USA. pp 22–22. http://dl.acm.org/citation.cfm?id=1251375.1251397
32. Newsome J (2005) Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the 14th Annual Network and Distributed System Security Symposium (NDSS'05). The Internet Society, San Diego, California
33. Intel (2005) Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture
34. Fraser T, Evenson MR, Arbaugh WA (2008) VICI virtual machine introspection for cognitive immunity. In: Computer Security Applications Conference, 2008. ACSAC 2008, Annual. IEEE, Anaheim, CA. pp 87–96. doi:10.1109/ACSAC.2008.33
35. Wen Y, Zhao J, Wang H, Cao J (2008) Implicit detection of hidden processes with a feather-weight hardware-assisted virtual machine monitor. In: Mu Y, Susilo W, Seberry J (eds). Information Security and Privacy, volume 5107 of Lecture Notes in Computer Science. Springer, Berlin Heidelberg. pp 361–375. ISBN 978-3-540-69971-2. doi:10.1007/978-3-540-70500-0_27 http://dx.doi.org/10.1007/978-3-540-70500-0_27
36. Vaculin R, Sycara K (2008) Semantic web services monitoring: An owl-s based approach. In: Hawaii International Conference on System Sciences. IEEE Computer Society
37. Ando R, Kadobayashi Y, Shinoda Y (2008) An enhancement of trusted domain enforcement using VMM interruption mechanism. In: Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for. IEEE, Hunan. pp 2222–2229. doi:10.1109/ICYCS.2008.341
38. Zhao F, Jiang Y, Xiang G, Jin H, Jiang W (2009) Vrfps: a novel virtual machine-based real-time file protection system. In: Software Engineering Research, Management and Applications, 2009. SERA '09. 7th ACIS International Conference on. IEEE, Haikou. pp 217–224. doi:10.1109/SERA.2009.23
39. Tymoshyk N, Tymoshyk R, Piskozub A, Khromchak P, Pyvovarov V, Novak A (2009) Monitoring of malefactor's activity in virtualized honeypots on the base of semantic transformation in Qemu hypervisor. In: Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2009. IDAACS 2009. IEEE International Workshop on. IEEE, Rende. pp 370–374. doi:10.1109/IDAACS.2009.5342958
40. Lengyel A, Neumann J, Maresca S, Payne BD, Kiayias A (2012) Virtual machine introspection in a hybrid Honeypot architecture. In: Presented as part of the 5th Workshop on Cyber Security Experimentation and Test. USENIX, Berkeley, CA. https://www.usenix.org/conference/cset12/workshop-program/presentation/Lengyel

doi:10.1186/s13677-014-0016-2
Cite this article as: More and Tapaswi: **Virtual machine introspection: towards bridging the semantic gap.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:16.