Hardware-Assisted MMU Redirection for In-Guest Monitoring and API Profiling

Shun-Wen Hsiao, Yeali S. Sun, and Meng Chang Chen

***Abstract*— With the advance of hardware, network, and virtualization technologies, cloud computing has prevailed and become the target of security threats such as the cross virtual machine (VM) side channel attack, with which malicious users exploit vulnerabilities to gain information or access to other guest virtual machines. Among the many virtualization technologies, the hypervisor manages the shared resource pool to ensure that the guest VMs can be properly served and isolated from each other. However, while managing the shared hardware resources, due to the presence of the virtualization layer and different CPU modes (root and non-root mode), when a CPU is switched to non-root mode and is occupied by a guest machine, a hypervisor cannot intervene with a guest at runtime. Thus, the execution status of a guest is like a black box to a hypervisor, and the hypervisor cannot mediate possible malicious behavior at runtime. To rectify this, we propose a hardware-assisted VMI (virtual machine introspection) based in-guest process monitoring mechanism which supports monitoring and management applications such as process profiling. The mechanism allows hooks placed within a target process (which the security expert selects to monitor and profile) of a guest virtual machine and handles hook invocations via the hypervisor. In order to facilitate the needed monitoring and/or management operations in the guest machine, the mechanism redirects access to in-guest memory space to a controlled, self-defined memory within the hypervisor by modifying the extended page table (EPT) to minimize guest and host machine switches. The advantages of the proposed mechanism include transparency, high performance, and comprehensive semantics. To demonstrate the capability of the proposed mechanism, we develop an API profiling system (APIf) to record the API invocations of the target process. The experimental results show an average performance degradation of about 2.32%, far better than existing similar systems.**

***Index Terms*— API hooking, malware, MMU, profiling, virtual machine introspection.**

I. Introduction

With the development of hardware and virtualization technologies, cloud computing provides elastic and cost-effective services and has become the de facto platform of business applications. A cloud computing system is composed of a collection of tightly networked physical computers, each of which is a host machine on which can be installed several guest virtual machines (VMs). A virtual machine has its own operating system (called the guest operating system) and running applications/processes (called in-guest applications/processes). The hypervisor, generally residing in the host machine, manages the shared resource pool and ensures that the virtual machines are properly served and isolated from each other. In a virtualized environment, a guest machine behaves as an independent machine; the hypervisor normally does not intervene in in-guest activities. As a result, the hypervisor may miss the opportunity to prevent or mediate malicious activities that occur within the guest machine.

For example, malware such as viruses, Internet worms, trojans, and botnets [1], are developed to disrupt network systems, steal sensitive information, or take control of a guest machine. They typically leverage libraries or Windows API calls [2] to perform privileged tasks, e.g., network communication and system management. Since these tasks are performed in the virtual machine, a hypervisor cannot easily monitor such malicious activities at runtime from the outside the VM. To facilitate such monitoring, we develop a profiling system in the hypervisor that monitors and records function call invocations made by malicious in-guest processes. Whether it is a virus or a worm, as long as malware calls a function, the proposed profiling system hooks and records such invocations, making our system a threat-agnostic forensic tool.

Virtual machine introspection (VMI) techniques have been proposed to inspect VM processes for purposes such as intrusion detection [3]. From a cloud management perspective, such a VMI mechanism must meet several requirements, including high performance, high transparency (i.e., hiding the existence of the VMI monitor from the process being inspected), and high semantics (i.e., the result is human-comprehensible). In this study, we develop a hardware-assisted in-guest monitoring mechanism that meets the above requirements, and to demonstrate its capabilities and strengths, we also develop APIf, an in-guest target process API invocation profiling system.

KVM (kernel-based virtual machine) [13] is a hardware-assisted virtualization solution for Linux on x86 hardware that exploits Intel virtualization extensions (Intel VT-x) [27]. It runs privileged instructions to allow an in-guest process to run directly on a CPU to boost

Manuscript received May 30, 2019; revised September 23, 2019 and January 3, 2020; accepted January 16, 2020. Date of publication January 27, 2020; date of current version February 6, 2020. This work was supported in part by the Ministry of Science and Technology, Taiwan, under Grant MOST-107-2221-E-004-003-MY2 and in part by the Taiwan Information Security Center (TWISC) of NTU and AS. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Debdeep Mukhopadhyay. *(Corresponding author: Shun-Wen Hsiao.)*

Shun-Wen Hsiao is with the Department of Management Information Systems, National Chengchi University, Taipei 11605, Taiwan (e-mail: hsiaom@nccu.edu.tw).

Yeali S. Sun is with the Department of Information Management, National Taiwan University, Taipei 10617, Taiwan (e-mail: sunny@ntu.edu.tw).

Meng Chang Chen is with the Institute of Information Science and Research Center for Information Technology Innovation, Academia Sinica, Taipei 11529, Taiwan (e-mail: mcc@iis.sinica.edu.tw).

Digital Object Identifier 10.1109/TIFS.2020.2969514

VM performance. Such a technique replaces the conventional software-based CPU emulator. Cloud services including Amazon Web Services (AWS) [4] are built on top of KVM or modifications thereof, including the proposed monitoring mechanism to utilize KVM advantages. In a nutshell, the idea of our approach is to redirect in-guest memory access to a controlled, self-defined memory space in the hypervisor using Intel VT-x extensions. A proper instrumentation step and time point are carefully designed and selected to modify the extended page table (EPT) managed by KVM to support the proposed memory redirection mechanism. We modify KVM's memory management mechanism and make it available for the profiling mechanism to replace the in-guest memory. When a hooked in-guest memory address is executed, the replaced memory contains additional instructions that are executed to collect process execution information. The proposed mechanism allows the hooked functions to be executed in the guest machine at native speed, preventing detection by the target process by inspecting execution time differences. This security mechanism within the hypervisor supports time-critical, concealed applications, such as VMI-based profiling [5], intrusion detection [3] for monitoring in-guest activities, and hot-patching for dynamic software updates without restarting the guest operating system. Note that it is possible for malignant cloud providers to execute their own functions to undertake malicious activities such as information theft without being detected.

To develop an efficient and effective VMI monitoring system, several objectives must be achieved.

A. Enhanced Transparency

The proposed VMI monitoring mechanism is located at the hypervisor layer, rather than in the guest OS; hence, the existence of the introspection mechanism is not detectable by the target process in the guest machine. Furthermore, we also introduce several methods to hide the existence of the redirected memory space for storing hooking functions and data to maintain transparency.

B. Improved Performance

As the mechanism is implemented with a hardware virtualization extension (Intel VT-x), it is faster than conventional emulation-based systems [6]. Furthermore, unlike conventional hardware-assistance based profiling systems (which must switch between different VM execution modes), with the proposed mechanism, the interception of in-guest execution does not require guest-host switches, reducing overhead and improving performance [7].

C. High Semantics

Generally, naive introspection records only raw bytes in the memory and/or instructions executed by the CPU, which are incomprehensible by human beings. The proposed VMI monitoring mechanism follows work on hardware-assisted VMI [8], [9] that monitors system call invocation; however, we target an even higher level of semantic information in that we collect the API invocations of a target process. We further traverse associated tables in memory to obtain the symbolic representation and parameters of the API to provide high-level semantics and readability.

The main contributions of this paper are summarized below.

- We develop a novel hardware-assisted MMU (memory management unit) redirection mechanism on in-guest memory access to self-defined, controlled space in the hypervisor by leveraging the Intel VT-x extensions to manipulate the address translations to provide high performance and transparency. Due to the complexity of memory virtualization, memory redirection on MMU and in the hypervisor has not been realized before. However, the need for high-speed in-guest hooking still exists. To the best of our knowledge, we are the first to propose a hardware-assisted MMU redirection system on a KVM hypervisor for userspace function hooking. Thus, a security expert can put any code in the redirected KVM-owned memories for use in introspection.
- The proposed introspection mechanism can be used on cloud platforms to monitor in-guest activities and facilitate security and service-oriented applications with high performance, high transparency, and high semantics. It can be used to prevent malignant users from malicious behavior and from disrupting cloud services, and can be used for novel applications such as time-critical hot-patching. However, the mechanism may be used by cloud providers for malicious acts of their own such as information theft or data modification.
- To demonstrate the capabilities and strengths of the proposed mechanism, we develop APIf, an efficient and secure VMI-based profiling system that records Windows API invocations of an in-guest target. The experiments show APIf incurs a 2.32% degradation in average performance.

The remainder of this paper is organized as follows. Section II provides a background on virtualization technology. Section III surveys related work. Section IV discusses the design rationale. Section V presents the implementation of our system in detail. Section VI shows performance evaluations. Section VII discusses our system design, the threat model, and possible applications of the proposed mechanism. Section VIII concludes the paper and discusses future work.

II. BACKGROUND

A. Hardware Assisted Virtualization

Generally, in the x86 architecture, the protection ring is designed to manage program execution in the correct privilege state [11]. With four privilege rings (rings 0, 1, 2, and 3), a system can effectively prevent high-privilege instructions from being executed in a low-privilege state. However, because the guest OS is considered by the host OS to be a general application, the execution of privileged instructions in the guest is denied by the protection ring mechanism. Before the emergence of hardware-assisted virtualization such as Intel VT-x [12]), binary translation (e.g., QEMU — Quick EMUlator [6]) was used to solve this problem. Due to the

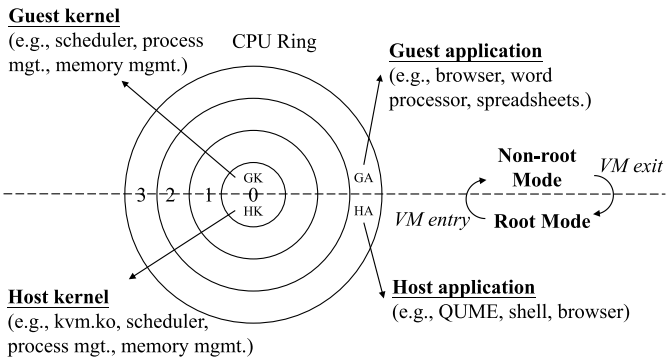

Fig. 1. Root and non-root modes in Intel x86 VT-x.

overhead of traditional binary translation, Intel announced Intel VT-x, a set of new processor extensions for the x86 architecture. Intel VT-x represents hardware-assisted virtualization technology designed to address the issue of ring compression by introducing new execution modes: VMX (virtual machine extension) root mode and VMX non-root mode, as shown in Fig. 1. The host operating system runs in root mode, while the guest operating systems run in non-root mode. Both execution modes support execution in four privilege rings, solving the issue of privileged instruction execution in guest mode. A hypervisor can switch the CPU between root and non-root modes by executing specific instructions to enter a VM (i.e., VM_entry) or exit a VM (i.e., VM_exit). Figure 1 shows the root and non-root modes in VT-x assisted virtualization and their relation to the protection ring.

Specifically, Intel VT-x [27] provides new instructions for switching between root mode and non-root mode. The VMRUN and VM entry instructions (VMLAUNCH and VMRESUME) switch control from root to non-root mode. Control is switched from non-root to root mode with a VMEXIT instruction. A critical data structure for interaction between the root (host) and non-root (guest) modes is the virtual machine control structure (VMCS), which stores both root mode (host) and non-root mode (guest) states. Upon VM entry, the state of the guest mode is loaded from the VMCS after storing the state of the root mode. Likewise, upon VMEXIT, the state of the root mode is loaded from the VMCS after storing the guest mode state.

B. Extended Page Table

Intel VT-x introduced a new hardware-assisted MMU using EPT (extended page table) to replace the traditional shadow page table [27]. Unlike soft-managed shadow page tables, EPT is implemented using hardware features so that the host machine only needs to maintain one EPT to perform second-level address translation (SLAT), which translates GPAs (guest physical addresses) to HPAs (host physical addresses), for all guest machines on the host. Compared to the shadow page table, EPT simplifies the process of second-level address translation and improves performance [28].

According to Intel's SLAT design [27], for each process in the guest virtual machine, its corresponding EPT base pointer points to the base address of the EPT. In a one-round

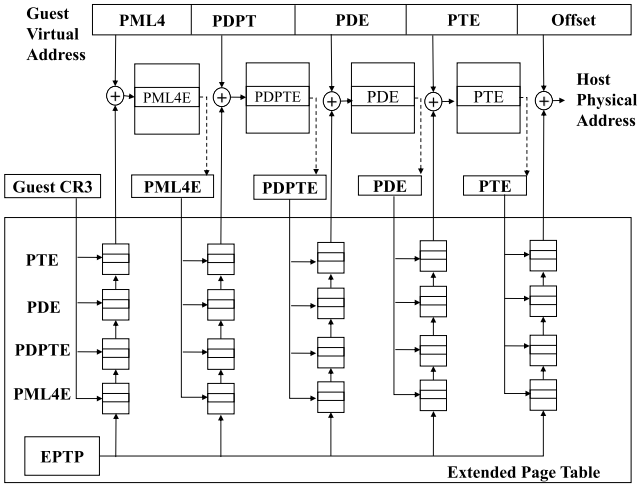

Fig. 2. Schematic structure of EPT address translation.

translation, the page table in the guest OS translates its GVA (guest virtual address) to a GPA and sends the GPA to KVM. Then, KVM translates the GPA to the corresponding HPA via EPT. Note that such translation is only valid in non-root mode, and it allows the userspace process in the guest machine to directly access the physical memory in the host through the MMU and EPT. Upon receiving the GPA from the guest, EPT walks through the multilayered page table to find the corresponding HPA and returns the result to the guest page table. In a 64-bit guest OS, address translation involves a four-layer guest page table. Thus, every time a GPA is to be translated to an HPA, EPT must perform a four-layer page walk. Figure 2 depicts memory address translation with EPT. A guest virtual address (GVA) is partitioned into five parts—the PML4 (page map level 4), the PDPT (page directory pointer table), the PDE (page directory entry), the PTE (page table entry), and an offset; the first fourth part is sent to the MMU to look up the base address of the next stage (see the top half of the figure). Each lookup goes through another 4-layer table lookup in the EPT (bottom half of the figure). After 5 iterations, the associated host physical address (HPA) is found.

C. Kernel-Based Virtual Machine (KVM) and QEMU

KVM [13] has now been widely adopted as the basic infrastructure of many cloud service providers. As KVM runs as a kernel module, it can manage hardware usage, for instance memory; however, as a kernel model, KVM cannot recognize file system access in the guest OS. Therefore, QEMU [6] was introduced as a user mode agent for in-guest file access, and as I/O event handlers or in-guest I/O interrupts. Below we describe the interaction between QEMU and KVM (see Fig. 3).

First, KVM creates a "`/dev/kvm`" device node as the communication channel between KVM and QEMU. QEMU creates the corresponding data structures and communicates through a set of `ioctl()` system calls to KVM, and KVM creates the corresponding data structure and allocates resources to initiate a VM. The KVM execution loop [13] then continues as below.

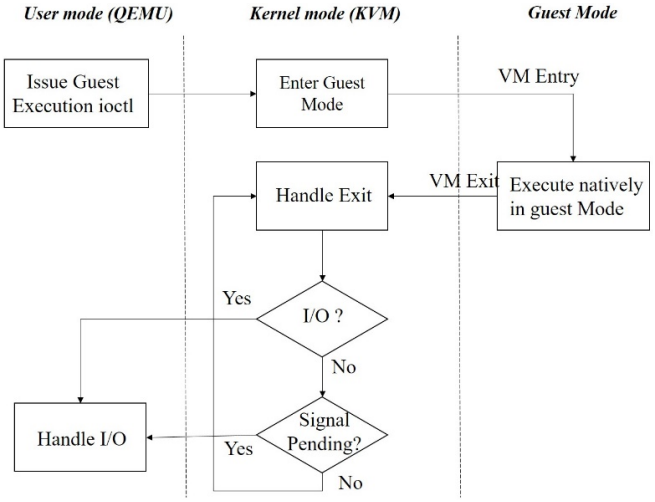

Fig. 3. KVM execution loop.

- User mode (QEMU): Once QEMU finishes the I/O tasks, it informs KVM to execute the code in guest mode and KVM takes over the CPU. QEMU resumes later when there is an external event that must be handled by QEMU, such as the arrival of network packets or timeout events.
- Kernel mode (KVM): KVM stores the host state to the VMCS and restores the guest state to perform VM entry to enter the guest mode (non-root mode). The guest code is executed natively on the physical CPU. Once the CPU exits guest mode due to an event such as an interrupt or a page fault, the CPU switches back to host mode (root mode) and KVM performs the necessary handling before resuming guest execution. If the exit is because of an I/O instruction or a signal queued to the process, then KVM exits to QEMU in user mode.

KVM is a built-in module in the Linux kernel that brings the following advantages.

- KVM utilizes built-in Linux subsystems to manage VM, including memory management, process management, and so on. Inspired by this design, the proposed VMI monitoring system is also implemented in KVM so that it can make use of existing features.
- Since KVM works as a kernel module, the proposed mechanism can access the privileged layer (ring 0); for instance, it can modify the EPT.

III. Related Work

Research has focused on VMI techniques such as page table manipulation based approaches, sandbox approaches, dual-VM approaches, emulator-based VMI, and hardware-assisted system-call VMI approaches.

A. MMU Manipulation of Normal Page Table

Lee et al. [14] introduce the page table manipulation attack (PTMA), a kernel exploitation technique to modify memory attributes through page table modification. This attack enables an attacker to rewrite memory attributes of protected memory (e.g., modify the highest 63rd bit to make the protected memory executable). PTMA is an effective technique because it targets the principle of the memory management mechanism. Lee et al. manage to find the targeted page table entry (PTE) in the master kernel page table and modify its attributes to evade PTE restrictions.

M. Seaborn reveals a working privilege escalation exploit [15] which uses row-hammer-induced bit flips [16] to gain kernel privileges on x86-64 Linux when running as an unprivileged userspace process. When running on a machine vulnerable to the row-hammer problem, the process is able to induce bit flips in page table entries (PTEs) to gain read/write access to its own page table, and hence gain read-write access to all of the physical memory. Furthermore, by filling the PTEs with different physical memory, attackers can manipulate the mapping between virtual memory and physical memory to make the corresponding virtual address point to a specific physical memory space. Thus, if the corresponding virtual address points to a physical page with read/write access, an attacker can read and write the physical page. As a result, an attacker can arbitrarily access physical memory spaces that should not be accessed.

The proposed mechanism also targets the memory management mechanism and modifies the access control of the protected physical memory region. However, instead of modifying the normal page table, we target EPT entries managed by the hypervisor. Moreover, we not only modify existing EPT entries, but our mechanism also creates new memory space in KVM and new entries in the EPT to store additional monitoring code which is accessible by both host and guest.

B. Sandbox

Conventionally, a sandbox creates an execution environment for a target program and monitors its execution. Willems et al. present a tool called CWSandbox [18] that executes the sample in either native mode or in a virtual Windows environment. CWSandbox performs API-level monitoring by API function hooking. Cuckoo [19] is an advanced, extremely modular, and open-source automated malware analysis system for malware analysis. Cuckoo Sandbox leverages virtual machine technology to be able to run on different operating systems (e.g., Windows and Linux), and provides high-level semantic execution information (e.g., API invocation tracing and general behavior of the accessed files) by the installed agent in the guest. However, in some scenarios, the presence of the in-guest agent may be detected by malware to evade such monitoring.

Both Cuckoo and CWSandbox provide good performance and high semantic information. However, they must install an additional in-guest agent and cannot be transparent to the target program. As a result, they are suitable more for monitoring tasks in the lab than for online or real-time monitoring.

C. Dual-VM-Based Approach

A VMI system with a dual-VM-based approach consists of a guest VM (GVM) and a secure VM (SVM), both installed with the same OS. The GVM is the virtual machine on which the

monitored applications run, and the SVM is a highly secure machine on which security applications run to avoid being compromised. Security applications in the SVM intercept and monitor execution of the programs in the GVM, for instance collecting information on system calls or API invocations. Lares [20] and Exterior [23] adopt such a dual-VM design. Lares requires that the GVM and SVM install the same version of the operating system. In addition, the kernel of the GVM must be modified so that when system calls are invoked on it, the system call handler in the GVM redirects these calls to the SVM and VMI programs installed in the SVM record the system calls for further security research. Similarly, Exterior, introduced by Fu and Lin, is also implemented using a dual-VM-based approach. The MMU of the GVM is modified so that the physical memory of the GVM is mapped to an additional memory space created by the SVM. Hence, VMI programs installed on the SVM can monitor the memory in the GVM.

CFWatch [21] also leverages a dual VM, but it monitors file operations access in the GVM from the SVM by monitoring critical file objects in the GVM's memory. Once the memory has changed, an interrupt is issued so that it can monitor file changes.

In sum, the benefit of the dual-VM design is that VMI programs hidden in the SVM are not easily detected and compromised by the monitored programs in the GVM. As the OSes of the GVM and SVM are the same, the semantic gap [22] between the GVM and SVM is narrowed. However, a dual-VM based VMI system requires at least two virtual machines and modification of the GVM, it incurs overhead due to VM communications and switches, and the modification of the GVM may increase the risk of being detected by monitored programs.

D. Emulation-Based Approach

An emulation-based (or emulator-based) VMI system is implemented in the software emulator, such as QEMU, to intercept the process of binary translation, and is able to record the execution of the guest virtual machine, instruction by instruction. Such a design allows for fine-grained tracking of the guest, but the instruction-level information thus obtained lacks high-level semantics and is thus difficult for analysts to understand. In addition, binary translation incurs high overhead, severely degrading the performance of emulator-based VMI systems. One noted emulator-based system is TEMU [25], which was introduced by Dong Song et al. in 2008. In TEMU, which was implemented on QEMU-0.9, the researchers extend binary translation by installing an additional plug-in to perform API hooking and taint analysis.

In sum, the strength of a software-based approach is that it can be more flexible than other approaches. However, the performance of such software-based systems is poor and the information collected is usually raw data, necessitating additional effort to bridge the semantic gap.

E. Hardware-Assisted VMI Systems

Due to the overhead of binary translation and soft-managed MMU (the shadow page table), the performance of emulation-based VMI system is called into question. In 2006, Intel introduced VT-x, a new x86 virtualization extension, the emergence of which inspired the birth of hardware-assisted VMI systems such as PMC, Ether [8], Nitro [9], CXPInspector [26], SIM [7], MvArmor [17], and Cuckoo [19] Sandbox. The difference between these systems is in their monitoring mechanisms and monitored targets. For example, Ether triggers its monitoring activities by page faults, Nitro is triggered by the interrupt descriptor table (IDT), and MVX intercepts the system call path to perform monitoring.

Ether [8] was the first VMI system to use a hardware-assisted approach. Ether is implemented on the Xen hypervisor [10], and monitoring programs are located in the hypervisor layer to keep the system invisible to programs in the guest. To record in-guest information, Ether intercepts in-guest execution by using page faults that trigger VMEXIT and change the CPU mode to the root mode. As a result, the hypervisor takes over execution and enables the monitoring programs to record in-guest system call execution. With hardware-assisted virtualization, Ether is more efficient than software-based VMI system. However, frequent VMEXITs incur heavy overhead.

Another VMI system with the hardware-assisted approach is Nitro [9]. Compared to Ether, Nitro is implemented on top of the QEMU/KVM architecture. Instead of using page faults to trigger VMEXIT, the authors modify the IDT to trigger a VMEXIT when an in-guest system call is invoked to enable monitoring components to be executed within the hypervisor (KVM). The researchers claim that Nitro performs 20% better than Ether and supports most x86-based operating systems (Windows and Linux). However, the performance of Nitro still suffers from the overhead for switching between the guest machine (non-root mode) and the hypervisor (root mode).

CXPInspector [26] was implemented by modifying EPT. At first, CXPInspector creates a cloned EPT and synchronizes the original and cloned EPTs. Next, the system applies the eXecutable pages (CXP) mechanism. The basic idea of CXP is to dynamically partition the main memory of the virtual machine into an executable part and a non-executable part. As the instruction pointer points to non-executable memory, a VMEXIT is triggered by a page fault interrupt and the hypervisor takes over execution from the guest OS to perform inspection. Meanwhile, in-guest API invocation can be intercepted by monitoring programs in the hypervisor and record corresponding API information such as arguments and return values. However, to intercept in-guest API invocation, CXPinspector still incurs the overhead associated with the switch between a guest machine and the hypervisor.

To avoid the overhead associated with frequent switches (VMEXIT), Sharif et al. introduce SIM, the secure in-VM monitoring framework [7]. Compared to prior VMI systems, SIM monitoring programs are installed on the guest virtual machine rather than on the hypervisor. SIM effectively reduces the frequency of switches and thus overhead. However, although SIM addresses the issue of switch overhead, monitoring programs installed in the guest OS still may raise the risk of being detected.

TABLE I
COMPARISON OF VMI SYSTEMS OF DIFFERENT DESIGNS. (HW-A = HARDWARE-ASSISTED)

System	Approach	Transparency	Performance	Semantics	Hypervisor
PMC [24]	PMC-based ins-level monitoring with debugging register	Yes	High	Low	KVM
Ether [8]	Trigger page fault to monitor syscall	Yes	Medium	Medium	XEN
Nitro [9]	Modify interrupt descriptor table to monitor syscall	Yes	Medium	Medium	KVM
SIM [7]	In-guest syscall monitoring with customized memory driver	No	High	Medium	KVM
CXP [26]	Modify EPT to monitor process via VM exit	Yes	Medium	High	KVM
Lares [20]	Kernel hook redirection via dual VMs	No	Medium	Medium	XEN
Exterior [23]	Memory map redirection via dual VMs	Yes	Medium	Medium	KVM
TEMU [25]	Taint analysis of CPU instructions	Yes	Low	Medium	QEMU
Cuckoo [19]	Python agent installed in Guest OS	No	Medium	High	N/A
Proposed (APIf)	MMU redirection by KVM ETP entry modification	Yes	High	High	KVM

There are also monitoring systems that take advantage of Intel hardware to monitor process behavior; however, these rely on hardware performance counters (PMCs) rather than virtualization technology. For example, PMC [24] traps hardware performance events to the hypervisor in order to intercept processes in the virtual machine.

Mishra et al. [38] categorized these systems into five types based on the introspection approach used: 1) guest OS hook based (e.g., SIM [7] and Lares [20]), 2) VM state access based (e.g., VMI-IDS [3], LibVMI [39] and Maitland [40]), 3) hypercall authentication based (e.g., Collabra [41]), 4) kernel debugging based (e.g., DRAKVUF [42] and SPEMS [43]) and 5) interrupt based (e.g., Nitro [9]). However, the proposed APIf does not belong to any of the category. Although APIf leverages guest OS hooking technique, APIf requires no modification of guest OS. In addition, comparing to the approaches in other categories, APIf requires no additional privilege domain (i.e., Domain 0 [40]), hypercall, debug berk point, syscall and interrupt.

In summary, compared to an emulation-based system, hardware-assisted techniques generally provide better performance. As for the dual-VM based approach, although it is helpful to bridge the semantic gap and provides good performance, it consumes more system resources than other approaches and requires additional modifications in the guest virtual machine. Considering transparency and cost-effectiveness, the dual-VM system is not a preferred approach. From the above survey, it is reasonable to adopt a hardware-assisted approach given its high performance and transparency. Inspired by SIM, we run a monitoring program in guest mode to avoid frequent switches between the guest machine and the hypervisor. Instead of installing the monitoring program in the guest kernel as with SIM or in the guest as with Cuckoo, our approach maps space in KVM to the target process via EPT to avoid switch overheads and maintain transparency. Table I contains a comparison of different VMI systems.

IV. SYSTEM DESIGN

With the proposed VMI mechanism, we seek to ensure transparency, performance, and semantics.

Transparency: Transparency implies no additional installation of monitoring programs or modification on the guest OS to reduce the risk of being detected.

Performance: Even with the hardware-assistance mechanisms, frequent VM entries and exits incur high overhead. To take this into account, the proposed in-guest monitoring activities are executed in guest mode rather than in host mode to reduce the frequency of switches.

Semantics: To facilitate automatic API hooking, a command-line interface allows the user to specify the names of APIs to be hooked.

Below, we continue to present the system design rationale, including the manipulation of MMU address translation and the system architecture. The proposed VMI monitoring system rationale is also presented.

A. MMU Redirection

Figure 2 shows the workflow of address translation between guest virtual addresses (GVAs) and host physical addresses (HPAs). First, when an in-guest process is about to access an address in guest memory, its GVA is translated to a guest physical address (GPA) via the page table in the guest. Then, the GPA is translated to a host physical address (HPA) via EPT. Through EPT translation, instructions or data stored in the GVA are fetched by CPU by using HPAs with the help of the MMU. Note that such access is possible only in non-root mode. Our monitoring mechanism intervenes and reverses the translation to allocate a new shadow leaf page (HPA) allocated by KVM that can be accessed by a target process using the GVA.

Figure 4 depicts the workflow of the proposed VMI monitoring system using shadow leaf pages. The rationale of the design is to use one-time instrumentation to trade for upcoming mode switches. When a task (i.e., process) in the guest is about to execute an instruction (GVA), the GVA is translated by PD and PT. At the instrumentation, the PT entries are modified to map to the shadow leaf pages to replace the original page. When the hooked address (GVA) is accessed by the CPU in guest mode, the MMU/EPT redirects the execution of the in-guest process and the CPU accesses the instruction/data on the shadow leaf page without additional guest/host switches. To ensure system performance and transparency, the cost is restricted to a one-time modification.

B. System Architecture

Figure 5 shows the system diagram of the proposed VMI monitoring system, in particular the three additional compo-

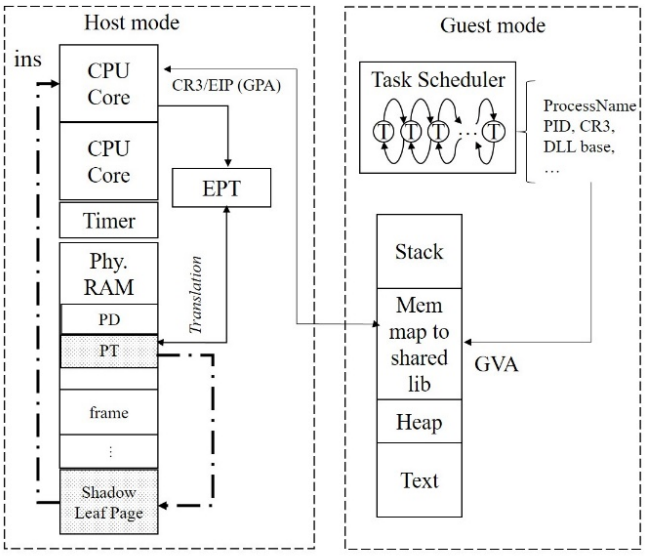

Fig. 4. Rationale of proposed VMI monitoring system using shadow leaf pages.

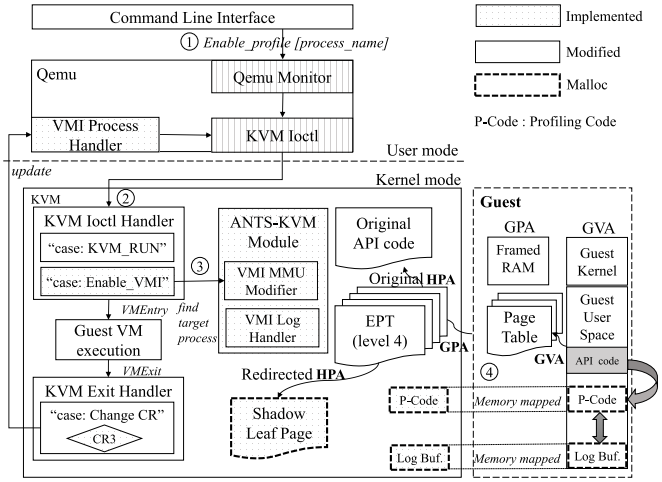

Fig. 5. VMI-based monitoring system diagram.

nents built on top of the QEMU/KVM architecture: the VMI Process Handler, the VMI MMU Modifier, and the VMI Log Handler. After the VMI Process Handler in QEMU receives the target process name from the command-line interface, it monitors the activation of the target process. As soon as the target process is active, the VMI Process Handler sends an `ioctl` signal to enable the VMI MMU Modifier and VMI Log Handler. The MMU Modifier creates the customized shadow leaf page and modifies the address translation so that the execution of the hooked address of the target process is redirected to the customized shadow leaf page. The VMI Log Handler manages the operation of the log buffer used for storing data from the guest (such as API invocation information in a profiling application). Henceforth, we describe the major components in detail by using a profiling application as an example.

1) VMI Process Handler: There are three major tasks for the VMI Process Handler. The first task is to know the target process. After accepting the target process name from the command-line interface (step 1 in Fig. 5), the Process Handler visits the guest kernel to access the current process list (details in Section V) to obtain the CR3 register value of the target process. If the current process list does not contain the target process, the Process Handler module is suspended until the next step. The second task is to design an instrument in the KVM Exit Handler (step 2). When a new process is spawned in the guest, the context switch of the process sets the control register (CR3_Change) [27] and the CPU exits to KVM by VMEXIT. This is a good time point for the Process Handler to inspect the activation of the target process and locate the addresses of the target APIs. The third task is to locate the GVA of the target API, and send the addresses to the MMU Modifier.

Currently, as address space layout randomization (ASLR) is widely adopted, the target GVA is randomized by the operating system. As a result, the second task is to locate the address that the target GVA must execute when the guest machine is rebooted. For profiling applications, we locate the base address of the DLL file of the target APIs (e.g., the address of `LoadLibraryA`) and combine the base address and offset to obtain the GVA of the target APIs.

2) VMI MMU Modifier: The VMI MMU modifier is responsible for allocating the necessary memory space in the host and modifying the EPT. After the Process Handler identifies the target process (step 3 in Fig. 5), the MMU Modifier allocates the needed memory space (step 4) by using the built-in kernel function `kzalloc()` for the use of shadow leaf pages (to store clone pages with inline hooking), the log buffer (used to store guest information), and the profile buffer (to store hook handling code), and then maps these in-host memory spaces (HPA) to the in-guest memory spaces (GVA). For EPT modification, the MMU Modifier first locates the address of the PT entry by using the target process CR3 value and the target GVA. By modifying this PT entry, the MMU Modifier redirects the GVA to the shadow leaf page. On these pages, we deploy the API hooks (P-Code) using the inline hooking approach to intercept the API execution in APIf, the proposed VMI profiling system. We describe the implementation in detail in the next section.

3) VMI Log Handler: The VMI Log Handler manages the operation of the log buffer, which provides an efficient channel between guest and host machines. The P-code in the profile buffer reads or writes data in the log buffer. The log handler handles the data and dumps it to a trace file in a ring buffer fashion [29] when KVM performs a regular VMEXIT so that buffer management does not incur any extra mode switches.

C. In-Guest Profiling Design

To demonstrate the capability and further explicate the instrumentation and utilization of the proposed VMI monitoring system, we implemented a profiling system that tracks API invocations of the target process in the guest. Previous works such as Ether and Nitro plant their profiling code in the hypervisor, necessitating a VM switch to execute profiling code.

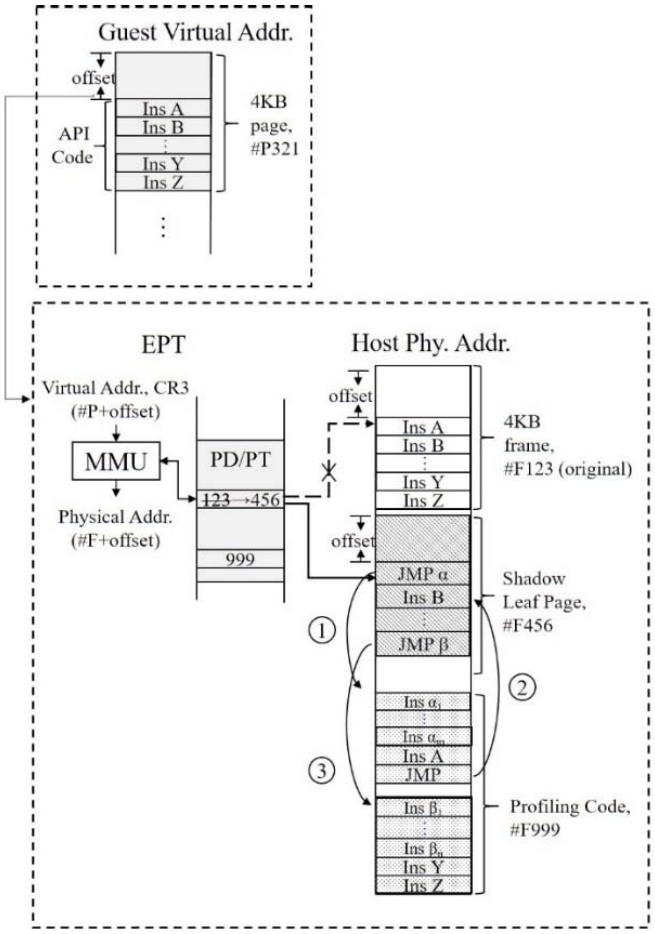

Fig. 6. EPT modification scenario.

To achieve transparency and avoid unnecessary VM switch overhead, the proposed in-guest profiling application executes in-host profiling code in guest mode. This calls for three steps of instrumentation.

1) *Enable in-host profiling code to be accessed in guest mode.* To enable in-host profiling code to be accessed in guest mode, the profiling system first creates an executable page in the host kernel for the target process. Then, it performs a reverse MMU address translation to map the host physical address to an unused guest virtual address.
2) *Redirect GVA to customized shadow leaf pages.* Figure 6 shows a simplified example of address redirection by modifying the EPT. Assume a 4KB page (in GVA #P321) which contains the original target API opcodes (instructions A to Z). After translation by the MMU, the address of this page is translated to an HPA frame (#F123) with control register CR3. After the VMI MMU Modifier intervenes, the original memory mapping (from #P321 to #F123) is changed by modifying the value of the PT entry to the redirected mapping (from #P321 to shadow leaf page #F456).
3) *Set up API hook handling on shadow leaf page.* However, replacing a page may not be enough. Sometimes we must execute extra code for different monitoring purposes. Figure 5 shows a template of the memory layout arrangement used by our system after modification. Originally, a function (instructions A to Z, Z is usually a RET instruction) is in #P321. Our design provides two instrumentation points—one before A and one before Z, so we can run a P-code (α_1 to α_m) before calling a function and another P-code (β_1 to β_n) before returning the function. First, shadow leaf page #F456 is a clone of #F123. Then, we rearrange the instructions (A–Z) and two P-codes (α_1 to α_m and β_1 to β_n) with three additional JMP instructions and one additional profile buffer (#F999). Thus the execution sequence becomes α_1 to α_m, A to Z, β_1 to β_n. Based on the template, Figure 7(a) shows the opcodes of `LoadLibraryA` in `kernel32.dll`. The instructions with the gray background are overwritten to delegate control to the P-codes. Figure 7(b) shows that the code after hook setup and the execution sequence of instructions should be the same as Fig. 7(a). In this example, the profiling code (not shown in the figure) records the API name, arguments, and return value. Moreover, in order to not affect the original execution sequence, we use the system stack (via PUSH/POP instructions) to save the original register values before entering the profiling code and restore the values after finishing profiling.

V. Implementation

We implemented the proposed VMI monitoring mechanism and profiling system on a computer with an Intel i5 dual-core CPU clocked at 2.5 GHz and 8 GB of RAM, with Ubuntu 14.04 and a Linux 3.16.0 kernel as the host OS. The virtual machine hypervisor was QEMU 2.3.0 and Linux KVM. The virtual machine was configured as a single-core virtual CPU with 2GB of RAM and Windows 7 64-bit SP1 as the guest OS. In the following subsections, we will describe the implementation of the proposed system, the obstacles we encountered, and the solutions provided.

A. Target Process State Retrieval

It is not trivial for the VMI Process Handler, which resides within the hypervisor, to retrieve the state of in-guest target processes. Retrieval starts by scanning the guest kernel debugger block (KDBG), and accesses `PsActiveProcessHead` [30], which serves as the head of a doubly, circularly linked list of Executive Process (EPROCESS) data structures. The EPROCESS structure is a process descriptor which contains critical information on the process such as the CR3 value, PID, and process name. By traversing the EPROCESS list, we retrieve the process state information. The user only needs to specify the target process name for the monitoring system to obtain this information (e.g., CR3) automatically. Figure 8 illustrates the procedure of target process state retrieval.

B. Enabling In-Host Code Access in the Guest Mode

To ensure the proposed VMI monitoring system is transparent to in-guest processes, we enable in-host code accessible in guest mode. Obviously, the host memory cannot be accessed in guest mode. Thus we set up an address mapping between an

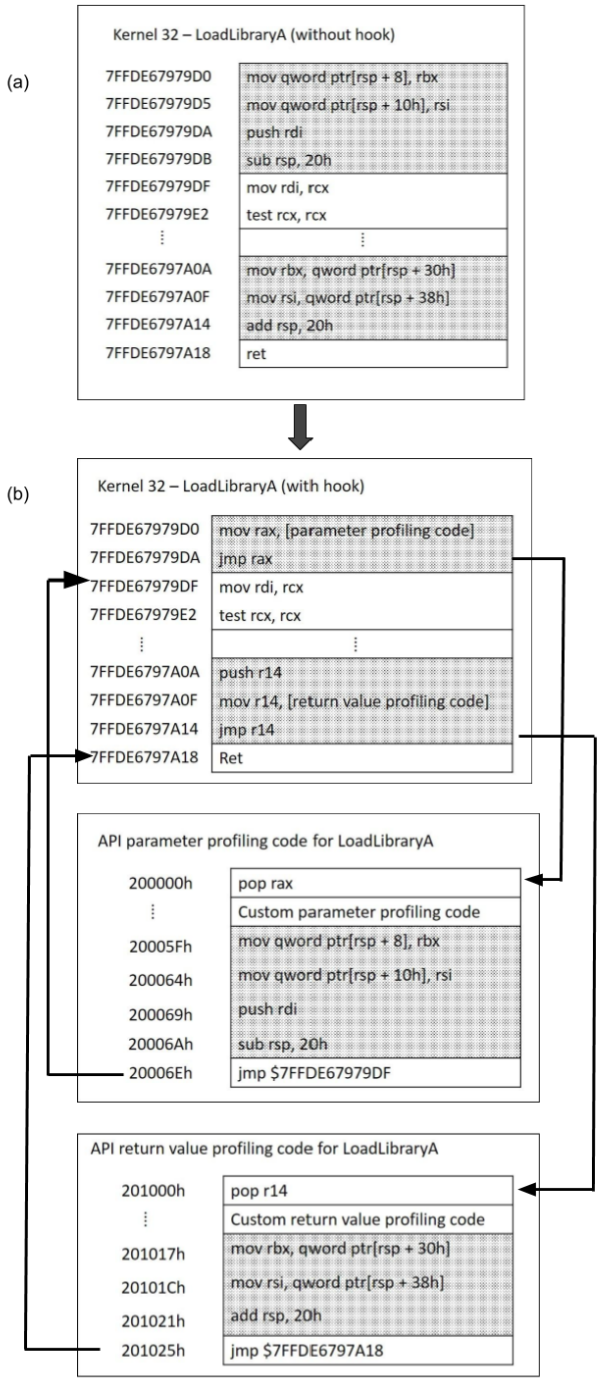

Fig. 7. Real-world example of in-line code overwriting for API hooking of `LoadLibraryA`.

allocated in-host memory space (HPA) and in-guest memory space (GVA) so that in-host code can be executed by an in-guest process using GVA. An executable page in the host kernel is created for the target process which is accessible via reverse MMU address translation [35]. First, we locate the PT table via the traversal sequence of PML4/PDPT/PD. Then, we find an unused memory address in the target process and create an executable page table entry on the PT for the address. A two-stage address mapping, from GPA to HPA (in the host) and from GVA to GPA (in the guest), is generated to facilitate the access.

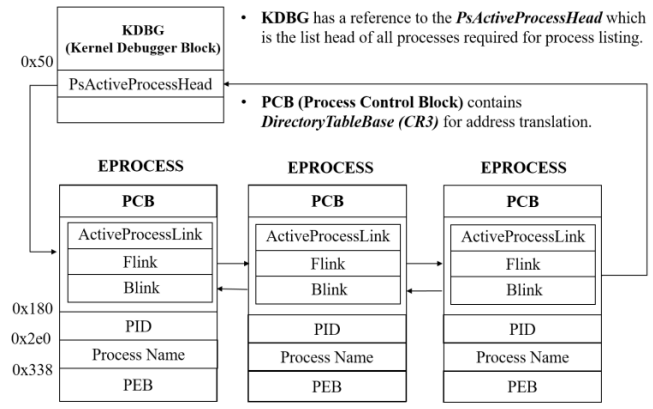

Fig. 8. Target process state retrieval in Windows kernel.

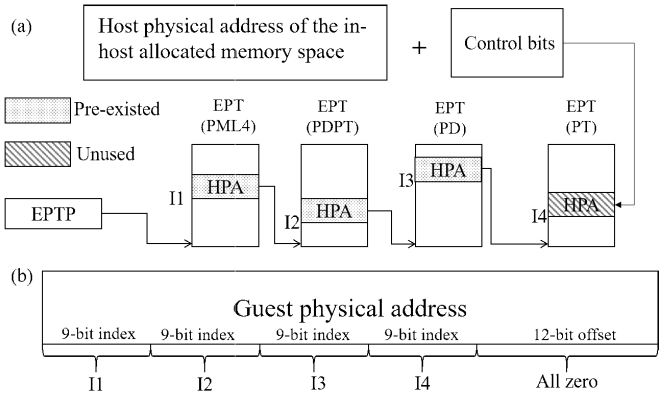

Fig. 9. Reverse address translation on EPT (GPA to HPA).

Figure 9(a) shows EPT traversal mapping a GPA to a given HPA in the host kernel memory. We start from the EPT pointer (EPTP) that locates the base address of the PML4 table. The EPTP is known when the target process is activated. Then, we begin a traversal on the PML4/PDPT/PD/PT tables by the following rules.

- In-use: During traversal, one in-use table entry is selected in PML4/PDPT/PD, respectively. As every table entry has several control bits to specify the state of this entry, we can easily select an in-user entry (at least one bit in control bits 0–2 is non-zero).
- In-memory: During traversal, we not only select an in-use entry but also select an in-memory entry. This indicates that the corresponding page is not paged out.
- 4KB page (7th control bit = 1, indicating a large page): We only select a 4k-page in the traversal.

Once a traversal path is found from PML4 to PD, then we find an unused entry in PT and occupy it. We fill the HPA in the unused PT entry and set its 63rd bit as executable. The path now forms a GPA, as shown in Fig. 9(b). Concatenating offsets (I1–I4) of each table together with 12 zero bits (i.e., a 4k-page) forms a GPA. Once we occupy the PT entry, we have a GPA that can be mapped to an HPA by EPT.

The second stage is to set the address translation from a given GPA to a GVA. First, we locate the base address of the PML4 table using the CR3 value of the target process

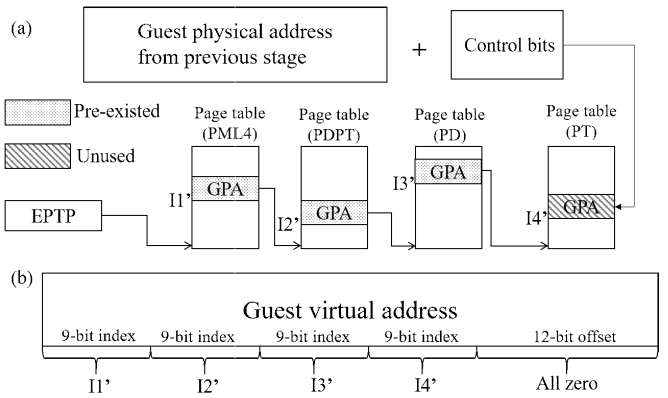

Fig. 10. Reverse address translation on page table in guest (GVA to GPA).

and start a similar traversal. Figure 10(a) shows EPT traversal mapping a GPA to a GVA. Similarly, once a path is found from PML4 to PT, the GPA from the previous stage and control bits (set as executable) are stored into an unused PT entry. Offsets (I1'–I4') of each table together with 12 bits of zeros form the GVA, as shown in Fig. 10(b). Now, the second stage of mapping GVA to GPA is complete.

Through this two-stage operation, we successfully construct the mapping between in-host allocated space and in-guest space. As a result, the in-host allocated spaces (e.g., shadow leaf page, profiling buffer, and log buffer) can be accessed by in-guest processes.

C. API Hook Handlings

To hook a target API, we must obtain its guest virtual address (GVA) and store the hook handler in a shadow leaf page. In the previous subsection we explained the mapping between a GVA of the target API and an HPA of a shadow leaf page. As mentioned, modern operating systems use address space layout randomization (ASLR), which randomly arranges the address space of sensitive data areas (e.g., stack, heap) of a process, to hinder some types of attacks. As a result, we must locate the address of the target API dynamically to hook target APIs correctly.

A virtual address is composed of a virtual base address and a relative address. The base address is dynamically allocated whenever the system is rebooted. The relative address is fixed and works as a relative offset. In our system, the base address is identified by scanning the DLL export table of a system default process to find the corresponding DLL base address, and then combining the relative address (a fixed offset) to dynamically locate the address of the target API. For example, we can locate the address of `LoadLibraryA`() by combining the base address of `kernel32.dll` from a system process (e.g., `winlogon.exe`) and a fixed offset. After locating the address of the target API, a shadow leaf page (which is mapped to the GVA of the target API) is created to store the API hook handler.

D. MMU Redirection Instrumentation Point

Although all the mechanisms for target API redirection are ready, we still require a right instrumentation time point before the first invocation of the target API in the target process, so that every hooked API invocation can be recorded. Generally, Windows systems adopt load-time dynamic linking, in which the DLL is initially mapped into the virtual address space of the process but is loaded into physical memory only when needed. Hence, the first API invocation in the target process incurs page faults and triggers a VM exit, EPT VIOLATION, in order to establish the address translation for the API. Figure 2 depicts the translation procedure via EPT: the bottom half of the figure shows that for address translation on the guest page table (GVA to HPA), five EPT VIOLATIONs are triggered, including one CR3 translation and four HPA translations.

Our system takes the fifth EPT violation as the instrumentation time point to modify the EPT for redirection. Furthermore, since API address translation on the EPT for a process is unique and exclusive, EPT modification on one process does not affect that of other processes in the virtual environment. This feature also allows us to perform targeted monitoring and to avoid interfering with other processes.

TABLE II
ADDED INSTRUCTIONS FOR EACH TARGET API

Target API	Ins.	Target API	Ins.
`CopyFileExA`	73	`RegQueryValueExA`	73
`CopyFileExW`	73	`RegQueryValueExW`	73
`CreateFileA`	71	`RegOpenKeyExA`	73
`CreateFileW`	71	`RegOpenKeyExW`	73
`DeleteFileA`	71	`RegSetValueExA`	71
`DeleteFileW`	71	`RegSetValueExW`	71
`CreateProcessA`	67	`RegCreateKeyExA`	73
`CreateProcessW`	71	`RegCreateKeyExW`	71
`WinExec`	69	`RegDeleteKeyA`	71
`TerminateProcess`	49	`RegDeleteKeyW`	71
`LoadLibraryA`	49		
`LoadLibraryW`	49		

E. Implementation of Windows API Profiling System

We record the behavior of an in-guest process via API execution traces that capture high-level behavior semantics. We selected several APIs (see Table II) from Microsoft MSDN in four operation categories: file I/O, process management, library invocation, and registry access.

- File I/O. We chose I/O-related APIs such as those for creating, reading, writing, copying, and deleting files.
- Process Management. To monitor malware activities at the process level, we chose APIs for process creation, launch, and execution.
- Library Invocation. We chose `LoadLibrary`() as one of the target APIs, as malicious programs often load self-provided libraries through dynamic linking.
- Registry Access. We chose APIs for registry search, opening, creation, closing, and deletion to monitor changes to system configuration by malware.

We hooked 22 Windows APIs using APIf, our profiling system. In order to provide high semantics, the profiling

```
API Name = '#RegOpenKeyExA'
Hkey = HKEY_LOCAL_MACHINE\
Argument = 'SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Setup'
Retrun Value = '0'
```

Fig. 11. An API invocation log.

system places extra code in the profiling buffer to further obtain API invocation parameters and return values from the target process stack. In addition, as some parameters are pointers (such as string objects and file handles), our profiling code supports further resolving them to the actual values to make the generated profile more readable and useful for analyzing malware.

The profiling code logs the API name, parameters (in the stack), and return values (in the RAX register) of the invoked API to the log buffer. The profiling code is implemented in x64 assembly. The logged data is stored in hex format. Figure 11 (in formatted ASCII representation) is an example of an API invocation trace in the log buffer.

F. Transparency

Transparency is another significant issue for VMI monitoring systems. We thus take several measures to ensure that our monitoring system is transparent to the target process in the guest machine. In general, transparency does not imply hiding the presence of a virtualized environment from the target process but hiding the presence of the monitoring mechanism.

1) Hiding Hooking Functions: In general, inline hooking is implemented by modifying the program. Comparing the program file itself (e.g., exe, dll) is a simple but useful method to detect if it has been modified. However, we deploy API hooks by modifying the EPT instead of modifying the program file directly. Therefore, such a straight comparison is ineffective.

API hooks can also be detected by memory integrity checking. Generally, hooking the `WriteprocessMemory` and `ReadProcessMemory` APIs and forging their parameters and return values could prevent against detection. For example, if the malware attempts to read the specified memory section to check the memory integrity by using `ReadProcessMemory`, we can hook the function and replace the contents of the reading buffer using the original program to avoid being detected.

2) Hiding EPT Modification: Because the EPT is out of reach of the guest machine, the target process can neither access the EPT nor is aware of any modifications to the EPT. Our system only makes use of the regular EPT_VIOLATION to perform one-time instrumentation; since no additional VM exit occurs, the target process cannot identify such instrumentation.

3) Time Handler: Malware can read a timer and calculate the execution time. If the time exceeds a certain threshold, malware may conclude that a monitoring system is present. Malware can use the RDTSC instruction to determine whether the execution time exceeds the threshold. The RDTSC instruction reads the TSC register and writes the value to the EAX and EDX registers. Generally, our system can intercept RDTSC instructions and forge the value of EAX and EDX to mislead malware.

VI. Evaluation

The profiling system was built on a PC with an Intel i5 dual-core CPU at 2.5 GHz and 8 GB of RAM, a host operating system of Ubuntu 14.04, and a Linux 3.16.0 kernel. The virtual machine was configured with one single-core virtual CPU with 2GB of RAM running Windows 7 64-bit SP1.

A. Code Size

We used the register-indirect jump technique to intercept process execution. We stored the address of the profiling code into the RAX register and then caused the process to jump to the address of the profiling code. This jump operation occupied two instructions. These are the assembly instructions used to implement the register-indirect jump operation:

```
mov rax, (addr. of~function)
; 48 b8 (8 bytes addr.)
jmp rax ; ff e0
```

The profiling code for collecting parameters and logging data used 49 to 73 additional instructions depending on the complexity of the corresponding hooked API and the number of parameters collected. Table II shows the occupied instructions of the profiling code for each API.

In addition to the profiling codes, three jump operations were needed to connect code in the shadow leaf page and profile buffer; thus the entire overhead of a hooked target API was between 55 to 79 instructions.

B. Macro-Benchmarking

In macro-benchmarking, we leveraged PerformanceTest 8.0 from PassMark [37], a commercial benchmarking product, to perform CPU, memory, and disk benchmark tests. Each test was performed on the guest OS with and without profiling tools. As the benchmark comparison we used Cuckoo [19], one of the most commonly-used open source profiling tools. The performance degradation is a straightforward indicator of the overhead incurred by the proposed profiling system (APIf) and Cuckoo. Table III shows the results. In all of the benchmark tests, our system incurs low overhead (between 0.0% and 7.38%). Disk Sequential Write and Disk Random R/W, the benchmark tests with the highest overhead, only show performance hits of around 7.38%. The overhead for memory operations is between 0.92% and 3.84%; the overhead of CPU operations is between 0.0% and 4.52%. These results show that the proposed system incurs little overhead on both CPU- and I/O-bound operations, whereas we support high transparency and high-semantic VMI profiling. It is interesting to note that these results are similar to the Nitro results [9]. The benchmarks with high overhead are both I/O-intensive tests, as the test program may invoke many Windows I/O-related APIs, incurring higher overhead.

C. Latency of Invoking Hooked APIs

We evaluated the execution latency incurred by the profiling system for each hooked API. The Windows system clock was

TABLE III

BENCHMARK TEST RESULTS. BASELINE IS PERFORMANCE RESULTS WITHOUT INSTALLING ANY PROFILING TOOLS, AND THE LAST TWO COLUMNS ARE THE PERFORMANCE OVERHEAD CAUSED BY THE PROPOSED PROFILING SYSTEM AND CUCKOO

		Overhead	
Benchmark	Baseline	APIf	Cuckoo
Integer math (M op/s)	1658.0	4.52%	51.30%
Floating point (M op/s)	912.0	2.85%	50.33%
Prime number (M prime/s)	7.7	0.00%	31.51%
Extended instruction (M)	5.3	1.89%	6.69%
Compression (KB/s)	1853.0	0.92%	14.57%
Encryption (MB/s)	241.2	0.62%	20.70%
Physics (frame/s)	137.4	4.51%	20.26%
Sorting (K string/s)	1224.0	0.41%	28.65%
Single-threaded (M op/s)	1299.0	1.62%	10.66%
Disk seq. read (MB/s)	214.1	1.82%	8.16%
Disk seq. write (MB/s)	202.4	7.07%	77.98%
Disk random R/W (MB/s)	209.9	7.38%	73.11%
Database (K op/s)	63.3	3.32%	92.84%
Mem. read cached (MB/s)	17497.0	0.56%	4.20%
Mem. read uncached (MB/s)	11353.0	0.92%	2.71%
Mem. write (MB/s)	8652.0	3.84%	5.84%
Memory latency (ns)	32.1	0.62%	2.59%
Memory threaded (MB/s)	11359.0	1.10%	4.58%

TABLE IV

AVERAGE EXECUTION TIMES AND MEASURED LATENCY FOR INVOKING HOOKED APIS

Target API	Execution time (μs)	Latency (μs)	Latency (%)
`CreateProcessA`	960	31.2	3.25%
`CreateProcessW`	624	46.8	7.50%
`WinExec`	569.1	102.4	17.99%
`TerminateProcess`	312	15.6	5.00%
`RegQueryValueExA`	6.02	31.2	518.27%
`RegQueryValueExW`	4.04	23.5	581.68%
`RegOpenKeyExA`	13.01	31.2	239.82%
`RegOpenKeyExW`	32.15	39.0	121.31%
`RegSetValueExA`	23	31.2	135.65%
`RegSetValueExW`	32.15	38.5	119.75%
`RegCreateKeyExA`	61.08	31.2	51.08%
`RegCreateKeyExW`	47.11	31.2	66.23%
`RegDeleteKeyA`	61.16	65.6	107.26%
`RegDeleteKeyW`	12.07	31.2	258.49%
`CopyFileExW`	1201.2	93.6	7.79%
`CopyFileExA`	1404	109.2	7.78%
`CreateFileA`	249.6	93.6	37.50%
`CreateFileW`	312	108.4	34.74%
`DeleteFileW`	296.4	31.2	10.53%
`DeleteFileA`	202.8	62.4	30.77%

used to measure the elapsed time to invoke each hooked API with the profiling system. Table IV shows the mean of the original elapsed time and the latency of 1000 runs of hooked API invocations. The time latency ranges from 30 to 100 μs. The time latency incurred by `LoadLibrary` API invocations is less than 1μs and is not shown in the table. The APIs for file operations take the most time on average (76.59 μs). Some registry APIs take significant performance hits.

We believe that the latency incurred by the profiling code (P-code) is very low because P-code consists only of around 70 instructions on average. As a result, we anticipate that continuously accessing the Windows registry may incur heavy system I/O and thus many log buffer accesses in our system, degrading performance. However, this can be improved by optimizing the buffering mechanism. This likely explains the unexpected overhead.

D. A Real-World Example

In Windows, when a user attempts to logon, the `LogonUser()` function is called. The user inputs the username and a plaintext password. If the function succeeds, the user receives an access token that specifies the credentials of the specified user account or, in most cases, creates a process that runs in the context of the specified user. In this experiment, we hook this function to record the username and password input of users. We implement a logon program with `LogonUser()` as shown in Fig. 12. When the user inputs the username and password (test/test), the logon information is recorded correctly in the log buffer by our profiling code.

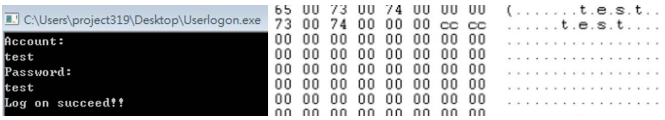

Fig. 12. Hooking `LogonUser()`.

VII. DISCUSSION

A. System Portability

The proposed VMI monitoring mechanism is hypervisor-portable due to the fact that all implementation requirements are KVM-independent. Users can migrate the monitoring mechanism from KVM to different hypervisors which meet the following requirements:

- *Modify page table entry to change address translation.* The page table structure and the principle of address translation are defined by Intel VT-x. As a result, any hypervisor which is compatible with the Intel VT-x extension can use the proposed mechanism.
- *Create shadow leaf page. All hypervisors can allocate memory space with write/executable access.* Hence, creating shadow leaf pages is not a problem for other hypervisors.
- *Knowing guest OS kernel structure. In Windows, we use the guest kernel structure, EPROCESS, to construct an active process list.* If the related structure in different OSes (e.g., the `struct_t` structure in Linux) is known, our monitoring mechanism can be applied on different guest OSes.

In sum, all of the requirements above can be achieved if the hypervisors are compatible with x86/64 hardware-assisted virtualization. Hence, the proposed monitoring mechanism is largely portable.

B. System Scalability

In this subsection, we discuss system scalability from several different aspects: multiple-process (or -thread) monitoring, multiple-VM monitoring, and shadow leaf page usage.

- Multiple-process (or -thread) monitoring. Since the guest process CR3 (gCR3) is unique to a process in a VM, we can determine which process invokes the hooked API. By doing so, our system can monitor multiple processes at the same time. Furthermore, if the target process forks new processes, our system can keep tracking its child processes (because we hook the process creation APIs, i.e., `CreateProcessA`, `CreateProcessW`, and `WinExec`, to obtain the process forking information). Our system generates separate profiles for different target processes. For multi-thread monitoring, when a process creates a new child thread to run malicious code, our system records the call invocations of the main thread and the child thread at the same time using the thread ID.
- Multiple-VM monitoring. To monitor multiple VMs, we can use the extended page table pointer (EPTP), which is stored in the VMCS and is unique to a VM, and can differentiate multiple VMs for simultaneous multiple-VM monitoring.
- Shadow leaf page usage. Generally, to hook one API, our system must allocate one shadow leaf page and one profiling buffer. However, in some cases, we can reduce the number of allocated pages by using a shared shadow leaf page and profiling buffer. For example, if the difference between the address of two APIs is less than 4096 (e.g., `RegSetValueExA` and `RegSetValueExW`), they can be put into the same shadow leaf page (4 KB page). Currently, our profiling system hooks 22 APIs and uses only 17 shadow leaf pages.

C. Threat Model

Since the proposed method hooks the memory of a guest machine dynamically, the system can inspect a) shared libraries stored in the userspace memory (such as the Windows API library and the C library) of a guest, b) system calls invoked by a guest process, and c) guest kernel functions. These function calls, used by malware, are also used by research [1], [2], [36] to analyze or classify malware. When malware uses these libraries to perform malicious tasks, the proposed method hooks them and records the usage of the functions for further analysis. Thus, we view our design as threat-agnostic, as the purposed system inspects any type of malware. The fundamental assumption is that the underlying hardware (including RAM, MMU, ETP, etc.) and hypervisor (i.e., KVM) are trusted.

However, if sophisticated malware performs malicious tasks without invoking any dynamic libraries or system calls (i.e., it includes all basic libraries statically and also refrains from invoking system calls), it can evade inspection of the proposed design. However, such a program is unlikely to be desired, because a) the program would be huge, b) the malicious tasks performed by such malware would be limited, and c) the memory used by the malware would be exorbitant. Such an implementation strategy would be unusual and would also raise additional suspicions about the program.

For anti-profiling mechanisms, we anticipate that two major approaches could be used by the attacker: time latency and memory integrity checking. First, stealthy malware might be able to detect the presence of our system by measuring the time latency incurred by the system (although it is very low). Thereafter, it would evade our monitoring by performing no malicious tasks. A possible solution to counter such a technique would be to hook and manipulate the time-related functions. However, we anticipate that it is difficult for a guest process to measure time latency in a virtual machine [31] due to the complexity of hypervisors (compared with that of a host machine). The incurred time latency can be viewed as the overhead of a normal hypervisor. Hence, as long as the latency is low or negligible, a guest process cannot differentiate if a function is hooked or not.

Second, since the original code is replaced by the shadow leaf page and the profiling code (see Fig. 6), the checksum of the code changes as well. One way to solve this problem is to swap the original frame back to the memory when a guest process performs memory integrity checking. However, detecting such checking at the runtime while profiling requires a sophisticated method. Thus, another possible solution is to adopt a two-phase monitoring mechanism. In the first phase, we record every function invoked by the guest process, including the integrity checking function. We deliberately trigger the memory integrity checking function and record the return value. In the second phase, we then manipulate the return value of the hooked memory integrity checking function to evade such checking. However, these two solutions are beyond the scope of this paper and necessitate an automatic code analysis tool to detect the presence of a memory integrity checking mechanism and manipulate the result of such a function dynamically.

Another possible attack is the attacker perturbing the memory of the target process after our system deploys the P-code. To deal with this attack, we can further make a code integrity check module in KVM to check the integrity of the deployed P-code when KVM is activated to make sure the profiling mechanism works correctly.

D. Potential Applications

The major application of in-guest monitoring and API profiling is to generate profiles for malware analysis, detection, and classification [1]–[3], [5], [8]. The API call invocation sequence collected by the proposed system can be viewed as the DNA of a process. Therefore, an analyst can develop algorithms to detect the API call sequence of malicious software [2] for misuse detection or to detect abnormal call sequences for anomaly detection [32]. Moreover, the API usage can be used to classify malware into different malware

families [5], [33]. Nevertheless, the proposed MMU redirection mechanism is not limited to process profiling.

In addition to building a process profiling system, the proposed MMU redirection mechanism can be applied in applications such as service hot-patching, keylogging, debugging, and password crackers (as shown in the experiments). By replacing the code section of the service that is about to be updated, system managers can update the services without restarting the system. However, such a mechanism could also be used by malicious cloud service providers to develop malware like the password cracker by hooking the authentication function, leading to information theft or loss. We expect the proposed mechanism to not only inspire cloud service providers to develop related management applications to confront emerging threats on cloud computing, but also to alert security researchers to look out for mechanisms that can be used to compromise the guest VM.

VIII. Conclusion

Virtualization is the core component of cloud computing that provides isolation between the different hardware and software services of customers. While the current design of the hypervisor focuses on resource sharing and management between virtual machines, it lacks a proper mechanism for security assurance, as more businesses embrace virtualized environments. New threats emerge every day, and one of the latest is the virtual machine cache side-channel attack [34]. In this paper, we design and develop an MMU redirection mechanism which intercepts the execution of API invocation in a guest virtual machine to support runtime security monitoring and profiling. Based on the proposed MMU redirection, we implement a profiling system prototype on QEMU/KVM with three important properties: minimum VM transition overhead (in-guest profiling), transparency to guest VM (no modification in guest OS), and high-level semantic data logging (high-level Windows API invocation monitoring and profiling). The experimental results show that our profiling system incurs no more than 7.38% system overhead and that the latency of executing individual APIs does not exceed 100 μs. The proven performance of our implementation allows for Windows API invocation monitoring and profiling in real-time. Furthermore, without guest OS modifications we keep the transparency of the guest virtual machine. High-level Windows API invocation profiling traces provide high-level information which can be easily understood by analysts.

References

[1] M. Egele *et al.*, "A survey on automated dynamic malware analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, Feb. 2012, Art. no. 6, doi: 10.1145/2089125.2089126.

[2] S. Gupta, H. Sharma, and S. Kaur, "Malware characterization using windows API call sequences," *J. Cyber Secur. Mobility*, vol. 7, no. 4, pp. 363–378, Oct. 2018, doi: 10.13052/jcsm2245-1439.741.

[3] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2003, pp. 191–206.

[4] S. Sharwood. (Nov. 7, 2017). *AWS Adopts Home-Brewed KVM as New Hypervisor*. [Online]. Available: https://www.theregister.co.uk/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/

[5] S.-W. Hsiao *et al.*, "A cooperative botnet profiling and detection in virtualized environment," in *Proc. IEEE Conf. Commun. Netw. Secur.*, National Harbor, MD, USA, Oct. 2013, pp. 154–162.

[6] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, Anaheim, CA, USA, 2005, pp. 41–46.

[7] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proc. ACM Conf. Comput. Commun. Secur.*, Chicago, IL, USA, 2009, pp. 477–487.

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

[9] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *Advances in Information and Computer Security* (Lecture Notes in Computer Science), vol. 7038. Berlin, Germany: Springer, 2011, pp. 96–112, doi: 10.1007/978-3-642-25141-2_7.

[10] P. Barham *et al.*, "Xen and the art of virtualization," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 154–177, Oct. 2003.

[11] R. Uhlig *et al.*, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, May 2005, doi: 10.1109/mc.2005.163.

[12] G. Neiger *et al.*, "Intel virtualization technology: Hardware support for efficient processor virtualization," *Int. Technol. J.*, vol. 10, no. 3, pp. 167–177, 2006.

[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Ottawa Linux Symp. (OLS)*, Ottawa, ON, Canada, Jun. 2007, pp. 225–230.

[14] J. S. Lee, H. M. Ham, I. H. Kim, and J. S. Song, "POSTER: Page table manipulation attack," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2015, pp. 1644–1646.

[15] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," presented at the Black Hat, Las Vegas, VA, USA, Aug. 2015.

[16] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 361–372, Jun. 2014, doi: 10.1145/2678373.2665726.

[17] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *Proc. Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Toulouse, France, Jun./Jul. 2016, pp. 431–442.

[18] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Secur. Privacy*, vol. 5, no. 2, pp. 32–39, Mar./Apr. 2007, doi: 10.1109/MSP.2007.45.

[19] *Cuckoo Sandbox*. Accessed: Sep. 20, 2019. [Online]. Available: http://www.cuckoosandbox.org

[20] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, Oakland, CA, USA, May 2008, pp. 233–247.

[21] D. Zhan, L. Ye, B. Fang, X. Du, and Z. Xu, "CFWatcher: A novel target-based real-time approach to monitor critical files using VMI," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Kuala Lumpur, Malaysia, May 2016, pp. 1–6.

[22] A. More and S. Tapaswi, "Virtual machine introspection: Towards bridging the semantic gap," *J. Cloud Comput.*, vol. 3, no. 1, pp. 1–14, Dec. 2014, doi: 10.1186/s13677-014-0016-2.

[23] Y. Fu and Z. Lin, "Exterior: Using a dual-VM based external shell for guest-OS introspection, configuration, and recovery," *ACM SIGPLAN Notices*, vol. 48, no. 7, pp. 97–110, Mar. 2013, doi: 10.1145/2451512.2451534.

[24] S. Vogl and C. Eckert, "Using hardware performance events for instruction-level monitoring on the X86 architecture," in *Proc. Eur. Workshop Syst. Secur. (EuroSec)*, Bern, Switzerland, 2012, pp. 1–6.

[25] D. Song *et al.*, "BitBlaze: A new approach to computer security via binary analysis," in *Information Systems Security* (Lecture Notes in Computer Science), vol. 5352. Berlin, Germany: Springer, 2008, pp. 1–25, doi: 10.1007/978-3-540-89862-7_1.

[26] C. Willems, R. Hund, and T. Holz, "CXPInspector: Hypervisor-based, hardware-assisted system monitoring," Ruhr-Univ. Bochum, Bochum, Germany, Tech. Rep. TR-HGI-2012-002, Nov. 26, 2012.

[27] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, Santa Clara, CA, USA, Oct. 2016.

[28] VMware Inc. (2009). *Performance Evaluation of Intel EPT Hardware Assist*. [Online]. Available: https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

[29] P. Hosek and C. Cadar, "VARAN the unbelievable: An efficient N-version execution framework," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 339–353, May 2015, doi: 10.1145/2786763.2694390.
[30] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, Oakland, CA, USA, May 2015, pp. 55–69.
[31] B. Adamczyk and A. Chydzinski, "Achieving high resolution timer events in virtualized environment," *PLoS ONE*, vol. 10, no. 7, Jul. 2015, Art. no. e0130887, doi: 10.1371/journal.pone.0130887.
[32] S. Forrest *et al.*, "A sense of self for unix processes," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, Oakland, CA, USA, May 1996, pp. 120–128.
[33] S.-W. Hsiao, Y. S. Sun, and M. C. Chen, "Virtual machine introspection based malware behavior profiling and family grouping," May 2017, *arXiv:1705.01697*. [Online]. Available: https://arxiv.org/abs/1705.01697
[34] A. Shahzad and A. Litchfield, "Virtualization technology: Cross-VM cache side channel attacks make it vulnerable," in *Proc. Australas. Conf. Inf. Syst. (ACIS)*, Adelaide, South Australia, 2015, pp. 1–14.
[35] S. Schaik, K. Razav, B. Gras, H. Bos, and C. Giuffrida, "Reverse engineering hardware page table caches using side-channel attacks on the MMU," Vrije Univ. Amsterdam, Amsterdam, The Netherlands, Tech. Rep. IR-CS-51, 2017.
[36] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *J. Inf. Secur.*, vol. 5, pp. 56–64, Mar. 2014, doi: 10.4236/jis.2014.52006.
[37] *PassMark Software*. Accessed: May 1, 2018. [Online]. Available: https://www.passmark.com/products/performancetest/
[38] P. Mishra, E. S. Pilli, V. Varadharajan, and U. Tupakula, "Intrusion detection techniques in cloud environment: A survey," *J. Netw. Comput. Appl.*, vol. 77, pp. 18–47, Jan. 2017, doi: 10.1016/j.jnca.2016.10.015.
[39] B. D. Payne, "Simplifying virtual machine introspection using LibVMI," Sandia Nat. Lab., Albuquerque, NM, USA, Tech. Rep. SAND2012-7818, Sep. 2012.
[40] C. Benninger *et al.*, "Maitland: Lighter-weight VM introspection to support cyber-security in the cloud," in *Proc. IEEE Int. Conf. Cloud Comput.)*, Honolulu, HI, USA, Jun. 2012, pp. 471–478.
[41] S. Bharadwaja *et al.*, "Collabra: A xen hypervisor based collaborative intrusion detection system," in *Proc. Int. Conf. Inf. Technol., New Generat. (ITNG)*, Las Vegas, NV, USA, Apr. 2011, pp. 695–700.
[42] T. K. Lengyel *et al.*, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New York, NY, USA, Dec. 2014, pp. 386–395.
[43] J. Shi *et al.*, "SPEMS: A stealthy and practical execution monitoring system based on VMI," in *Cloud Computing and Security* (Lecture Notes in Computer Science), vol. 9483. Cham, Switzerland: Springer, 2015, pp. 380–389, doi: 10.1007/978-3-319-27051-7_32.

Shun-Wen Hsiao received the B.S. and Ph.D. degree from the Department of Information Management, National Taiwan University, in 2004 and 2012, respectively. From 2006 to 2008, he participated in the iCAST Collaborate Research Project with the CyLab, Carnegie Mellon University. In 2012, he joined the Institute of Information Science, Academia Sinica, Taiwan, where he held a Post-Doctoral Research Fellowship. Since 2017, he has been with the Department of Management Information Systems, National Chengchi University, where he is currently an Assistant Professor. His research interests are in the area of cybersecurity, malware behavior analysis, virtualization technology, and FinTech.

Yeali S. Sun received the B.S. degree in computer science from National Taiwan University, and the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles in 1984 and 1988, respectively. From 1988 to 1993, she was with Bell Communications Research Inc. (Bellcore), where she was involved in the area of planning and architecture design of information networking, broadband networks, and network and system management. In 1993, she joined the Department of Information Management, National Taiwan University, where she is currently a Professor. She served as the Department Head for National Taiwan University, from 2006 to 2008, where she was the Director of the Computer and Information Networking Center, from 2009 to 2014. Her research interests are in the areas of system and network security, quality of service, wireless mesh networks, multimedia content delivery, Internet pricing and network management, and performance modeling and evaluation.

Meng Chang Chen received the B.S. and M.S. degrees in computer science from National Chiao Tung University, Taiwan, and the Ph.D. degree in computer science from the University of California at Los Angeles, in 1989. He then joined AT&T Bell Labs, as a member of Technical Staff and worked as the technical leader of several projects in the area of data quality of distributed databases for mission critical systems. After that, he has been with Institute of Information Science, Academia Sinica, Taiwan, and assumed the responsibility of Deputy Director for 5 years, where he is currently a Research Fellow. His current research interests include wireless networks, network and system security, machine learning, and data and knowledge engineering.