A Virtualization Based Monitoring System for Mini-intrusive Live Forensics

Xianming Zhong[1], Chengcheng Xiang[1], Miao Yu[2]
Zhengwei Qi[1], and Haibing Guan[1]

[1] Shanghai Jiao Tong University
{zhongxianming, xiangchengcheng, qizhwei, hbguan} @sjtu.edu.cn
[2] Carnegie Mellon University
superymk@cmu.edu

Abstract. Digital evidences hold great significance for governing cybercrime. Unfortunately, previous acquisition tools were troubled by either the shortage of suspending the target system's running or the security of the acquisition tools themselves, thus the correctness and accuracy of their obtained evidences cannot be guaranteed. In this paper, we propose VAIL, a novel virtualization based monitoring system for mini-intrusive live forensics, which employs hardware assisted virtualization technique to gather integrated information from the native computer system. Meanwhile, the execution of the target system will not be interrupted and VAIL keeps immune to attacks from the target system. We have implemented a proof-of-concept prototype that has been validated with a Windows guest system. The experimental results show that VAIL can obtain comprehensive digital evidences from the target system as designed, including the CPU state, the physical memory content, and the I/O activities. And on average, VAIL only introduces 4.21% performance overhead to the target system, which proves that VAIL is practical in real commercial environments.

1 Introduction

With the rapid development of the Internet, the attendant cybercrime also brings a shocking scale of time and financial losses. According to the report from Norton [1], in 2011, the total cost of cybercrime in the USA was about 139.6 billion US dollars, among which 32 billion was the direct cash cost and 107.6 billion was victims' value of time lost.

Consequently, computer forensics technique becomes an emerging research area, which has attracted many researchers to explore. The complete forensics process consists of three distinct phases: acquisition, analysis, and presentation [2]. The acquisition phase is the foundation of computer forensics, which focuses on obtaining the necessary information about the target system state as digital evidences. The gathered evidences should be truthful and accurate, otherwise wrong conclusions will be made in the following analysis phase and presentation phase.

Depending on whether investigators keep the target system running or not, computer forensics can be divided into two main groups: static forensics and live forensics. Considering criminal evidences being stored on the permanent I/O device only [3], most static forensics tools, like Encase [4] and FTK [5], clone the disk offline to accurately obtain digital evidences. Static forensics can get a lot of information about the target system, like web pages viewed, installed programs, log files and so on. But the volatile data that only existed in memory is totally beyond the scope of static forensics. Moreover, servers of online retailers usually require 24/7 availability and the shutdown of these machines may bring unacceptable money losses. Luckily, live forensics remedies the disadvantages of static forensics. It extends the information gathering range of forensic examiners without blocking the running of the target system. These data involve processes information [6], the process list [7], kernel objects [8], and the raw memory content [9,10], which may be leveraged to record and reproduce the crime scene.

The trivial software solution for live acquisition is to query from the Operating System (OS) directly. Iain et al. [11] list several practical tools for different scenarios, including Win32dd [12], KnTTools [13], and Fport [14]. Memoryze [15] is another popular user process forensic tool of this type. However, if the OS is compromised by skillful criminals, it may return fake and incorrect system status to cheat the acquisition tools.

Virtualization technology wraps the OS and user applications in a Virtual Machine (VM), which is monitored and managed by the Virtual Machine Monitor (VMM) like Xen [16]. The researchers add the acquisition module in the VMM to exploit the higher level of privileges, so that the malware in kernel-level cannot tamper the forensic progress easily. This kind of acquisition tools has been developed rapidly in recent years, especially after the hardware extensions for the x86 architecture [17, 18] simplified the development of the VMM. The representatives of this type are VIX tools [7], Ruo's work [9], Srinivas's work [19], and BodySnatcher [20].

In fact, current forensic tools still face significant challenges and risks. Ayers discussed limitations of existing forensic tools and proposed a set of requirements for the next generation tools [21]. Garfinkel also summarized current forensic research directions and argued that where future forensics works should focus on [22]. We think the first main challenge is that many previous approaches [7, 9, 19] need to set up the VMM level prior to the launching of the OS, which alters the target system running environment significantly, especially for the non-virtualized host. In the extreme case, rebooting or reinstalling the whole system is required, thus sharing the same problem with static forensics. The second challenge arises from the fact that the VMM has a broad attack surface and is not immune to serious security vulnerabilities [23]. The larger code size the VMM has, the more vulnerabilities it may suffer from. So designers have tried to remove unnecessary parts of the VMM, achieving a minimal code size and external interface, like SecVisor [24] and BitVisor [25]. But they suffer from the first challenge, which alters the execution environment drastically. Furthermore, few of researches address the acquisition of I/O devices. Memory dump files can

only provide snapshots of the system memory at a certain time, which leads to lack of understanding of the target system's activities during a continuous period of time.

In this paper, we propose VAIL, a novel virtualization based monitoring system for mini-intrusive live forensics, which considers the previous challenges simultaneously. Benefiting from *Silent-Virtualization* approach, VAIL is simply loaded as an OS driver to build the VMM layer, then requires no binary modifications to legacy applications and the guest OS. It also does not interrupt the execution of the target system. VAIL can obtain comprehensive digital evidences from the target system as designed, including the CPU state, the physical memory content, and the I/O activities. It has such a small code size that it is suitable for formal verification to make sure serious vulnerabilities have been excluded. In addition, VAIL hides and protects itself sufficiently by *Memory-Hiding* mechanism, avoiding being detected and compromised by malicious software or the compromised guest OS. To validate our approach, we have implemented a prototype and conducted a series of evaluations. The experimental results show that VAIL can protect itself appropriately and incurs a slight performance overhead to existing applications, proving that VAIL is applicative for realistic environments and improves live forensics technique significantly.

The rest of the paper is organized as follows. Section 2 presents VAIL's design assumptions and model. Section 3 provides implementation details and related discussions, while Section 4 evaluates VAIL through a series of experiments. We survey related work in Section 5, then illustrate the conclusion and the future work in Section 6.

2 Design

2.1 Goals and Assumptions

In order to address the existing issues that computer forensics suffer, we have following main design goals. First, the proposed techniques should influence the execution of the target system as little as possible, which means tending to achieve live forensics perfectly. Second, the trueness and correctness of the acquisition results should be guaranteed, even under the situation that the OS has been compromised by some malware. Third, comprehensive digital evidences should be gathered, including the CPU state, the physical memory content, and the I/O activities.

Since we leverage hardware assisted virtualization, the Hardware Enabled Virtualization (HEV) should be available on the processors of the target system. We assume that the target system is booted from bare metal, then the lower hypervisor layer can be built successfully. Theoretically, VAIL works normally in nested virtualization environments, but we do not consider the recursive virtualization situation due to each CPU core can run at most one VMM at any time in the current circumstance. VAIL is supposed to be loaded as an OS driver. If a malicious kernel has been forbidden to insert drivers, there are already some

ways to break this limitation, like the page-file attack [26]. Finally, hardware based attacks are out of the scope of this paper, such as TLB cache [27], SMM exploitation [28], and malicious DMA [29].

2.2 Silent-Virtualization Approach

Inspired by the NewBluePill Project [30], we propose Silent-Virtualization technique to build the hypervisor layer silently and dynamically with the help of hardware assisted virtualization. Even an OS is running, we can load VAIL at any supposed time, and do not disturb the execution of the target system. In other words, for normal users, nothing special has happened and they will not notice this huge change of the system architecture. Both Intel and AMD have proposed their own hardware assisted virtualization technology [17, 18]. Though each technique has its own characteristics, their main design goals and principles are the same. We implement our VAIL prototype based on Intel VT-x technology [31].

Intel VT-x technique separates the CPU execution into two modes: *VMX root mode* and *VMX non-root mode*. It fixes the so-called virtualization hole problem in x86 architecture, which indicates that every sensitive instruction is only permitted to be executed in VMX root mode. Then the hypervisor owns a full control of the VMs' running. As the codes of the guest OS are executed in VMX non-root mode, if any sensitive instruction is included, the CPU will switch to VMX root mode and active the hypervisor handler. After the hypervisor handler has finished dealing with the exception, the CPU resumes to execute the guest OS. The hypervisor maintains a Virtual-Machine Control Structure (VMCS) for each VM to record the guest system's execution environment status. We can also register some interested events in VMCS, like page fault exception or I/O instructions, to trap into the hypervisor when it is necessary.

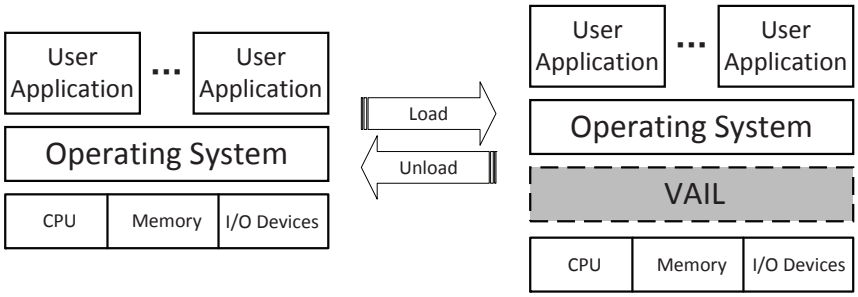


Fig. 1. The architecture overview of Silent-Virtualization Approach.

A typical VMM launching with Intel VT-x technique consists of three steps. First, the HEV is enabled via BIOS. Second, the CPU is configured to execute the VMM in VMX root mode. Third, the guest OS is booted in VMX non-root

mode. The launching of hypervisor can occur at any time, even there has an OS running. We just need to continue running the virtualized system in the third step appropriately.

Figure 1 shows how the system architecture is changed when loading/unloading VAIL. VAIL is loaded as an OS driver, which creates a process and acquires memory through standard kernel APIs. The essential virtualization environment is initialized in the loading procedure and the OS is wrapped into a VM silently. It shares the same usage model with many existing forensic tools listed in [11]. We think little modification during VAIL installation is tolerable. After all, no zero invasive solution for a *posteriori* forensic analysis exists [32]. In addition, VAIL employs Memory-Hiding mechanism, which will be introduced in the next section, to clear the memory footprints and hide itself. The unloading procedure is much similar to the loading procedure, which is also mini-incursive to the guest OS. Memory and other resources occupied by VAIL are released and the guest OS regains the power of controlling.

2.3 Memory-Hiding Mechanism

Besides the loading/unloading procedure, it is also very important for VAIL to hide itself when working under the target system. The main threat is that the malicious software may compromise the OS kernel, thus the existence of VAIL may be exposed and the safety of VAIL cannot be ensured. When having detected the existence of VAIL, the compromised kernel can either disturb the forensic processing or attack VAIL directly. In order to protect VAIL from being wrecked, two aspects of integrity need to be taken into consideration: static integrity and runtime integrity [33].

Static integrity can be guaranteed by read-only media without installing into the hard disk, i.e., Live DVD/USB [2], and VAIL's code can be attested before loading by employing the Trusted Platform Module (TPM) [34]. But the runtime integrity is more difficult to be achieved. Unlike general *VMM-based* forensic systems that start earlier than the guest OS, VAIL is loaded as an OS driver and will leave memory footprints before it owns the full control of the whole system. Although the OS isolates and protects the memory of different processes by virtual address translation mechanism, so that malicious processes cannot access or modify other processes' memory easily. The compromised kernel can still corrupt the memory of VAIL and stops VAIL from working normally.

VAIL resists attacks from the guest OS by Memory-Hiding Mechanism. Since the guest OS is not worthy to be trusted any longer, VAIL needs to have its own memory management system that is independent of the guest OS. As Figure 2 illustrates, VAIL travels the original page table of the guest OS to find the kernel used physical memory pages, and maps them into a private page table at the same virtual memory address. In particular, the virtual memory address that stores the guest OS page table should point to the private page table now. And after VAIL has been loaded, it will change the entries of its memory in the guest OS page table to spare physical memory pages with fake contents.

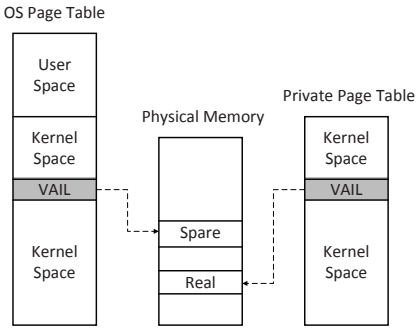


Fig. 2. The design model of Memory-Hiding Mechanism.

Here have two main purposes. First, VAIL becomes autonomous of memory management because the private page table has replaced the position of the guest OS page table. The guest OS will be unable to access or modify the real physical memory of VAIL, which enhances the reliability of VAIL. Second, kernel objects and OS APIs are available for VAIL, so that VAIL can take advantage of them to manage its own memory more conveniently. We should track the modifications to the guest OS page table, that means once they happen, the corresponding entries of the private page table should also be updated. The synchronization between the OS page table and the private page table is maintained by EPT, which we will introduce in detail in Section 3.

Much similar to the scheduling of processes in the traditional OS, the private page table is activated when the CPU loads its physical memory address in the CR3 register. When the VM starts and the guest OS resumes to be executed, the CR3 register will switch back the physical memory address of the original OS page table. The frequency depends on how often the sensitive events happen. According to our experimental results presented in Section 4, the performance overhead is very slight.

3 Implementation

Currently, we have implemented a prototype of VAIL leveraging Intel VT-x technique and succeed to apply it for live forensics of a Windows system. The activities of the whole target system can be divided into three parts: CPU, memory, and I/O. In this section, we introduce how VAIL can monitor them in turn.

3.1 CPU

Because VAIL only supports a single guest OS, we can ignore the scheduling and context switches among different guests and focus on context switches between the guest OS and the hypervisor. Based on Intel VT-x, each guest OS associates

with a region in memory called VMCS region, which saves the necessary context information and controls Virtual-Machine Extensions (VMX) non-root operation and VMX transitions [31]. During the execution of the guest OS, if certain events occur, it will result in a VMEXIT event and transfer control to the hypervisor. Depending on the fields in VMCS, these special events include execution of sensitive instructions, hardware or software interrupts, and other exceptions. After taking appropriate actions, the hypervisor can then return to the guest OS by generating a VMENTRY event.

Exploiting the feature of acquiring the CPU state, one useful appliance is detecting hidden processes. The attackers usually hide their malicious processes by tampering some aspect of the OS, in order to avoid touching off a defender. For example, modifying program binaries like *ps*, hooking into the call path between a user application and the kernel, or manipulating kernel data structures directly by DKOM are three mainstream means for hiding processes [35]. But each running user process corresponds to a different CR3 value. When loading or storing control registers by the MOV instruction, it is a privileged operation, thus VAIL can intercept all loaded CR3 values. If we compare them with the processes list that the guest OS provides, any intercepted CR3 that does not exist in the list is suspicious.

3.2 Memory

In order to dominate the memory management, VMMs add an extra memory translation layer after the translation of the OS virtual address to physical address. Traditional VMMs employ Shadow Page Table (SPT) to maintain the address updating of this additional layer in a software way. But the updating of memory page table entries may be very frequent, so the performance overhead caused by context switch is tremendous. With the HEV technique, this additional memory address translation is handled by the hardware, which simplifies the implementation of the VMM. In Intel VT-x technique, it is called Extended Page Table (EPT), while AMD-V names it as Nested Page Table (NPT) with the much similar mechanism.

With EPT mechanism employed, after translating the Guest Virtual Address (GVA) into the Guest Physical Address (GPA), the guest OS is no longer able to use the GPA to access the physical memory directly, though it will not notice this fact. Instead, the real physical memory address, referred as the Machine Physical Address (MPA), is produced by traversing a set of EPT paging structures. By EPT, VAIL saves much code to do the memory addresses translation. EPT paging structures are similar to those used in the guest OS, consisting of the address field and permissions field. If try to apply unsupported operations on an arbitrary page, the guest OS will be interrupted and trapped into the hypervisor. VAIL monitors the updating of the OS's page table by marking the corresponding entries in EPT as read-only. Then every modification leads to a page fault and VAIL will update the private page table appropriately. Popularizing this method to the whole memory space, Miao et al., [36] have implemented a prototype

to dump physical memory contents by marking EPT entries as read-only, and improved it with some optimizations [37].

3.3 I/O

I/O activities take place very frequently in normal computer systems and it is important to monitor I/O accesses for digital forensics. By default, VAIL allows the OS drivers to access devices directly, which frees VAIL from complicated device drivers and reduces risks of involving security vulnerabilities. When it is necessary, investigators can custom VAIL to intercept the I/O activities of the critical devices. There are three ways for hardware drivers to access I/O devices: Programmed I/O (PIO), Memory Mapped I/O (MMIO), and Direct Memory Access (DMA) [25]. We use two applications that VAIL supports contemporarily, keyboard logger and Network Interface Card (NIC) interception, as examples to describe the details of I/O acquisition.

Keyboard Logger As the keyboard is one of the most common input devices, logging and analyzing the keys typed will have great significance on comprehending the purposes and usage patterns of criminals. The keyboard driver is mainly implemented by PIO, which means using I/O instructions to read or write specified I/O port address. The hypervisor running on Intel VT-x processors can specify I/O addresses to be intercepted by configuring the I/O bitmaps in VMCS. And the I/O port addresses assigned to each device can be determined in advance and will not be overlapped, so that we can only intercept all PIOs of the target device and ignore others. To capture changes made by PCI configuration mechanism, the hypervisor should also intercept the PIO of it and update the I/O bitmaps.

The keyboard on the PC platform usually uses two I/O ports: a control port (0x64) and a data port (0x60). As the name indicates, the control port is used for sending commands or reading the status of the keyboard, and the data port is in charge of transmitting data between the device and the OS. Once a key has been pressed or released, the keyboard will generate a scan code and send it to the keyboard driver by the OS interrupt mechanism. In details, the keyboard driver receives scan codes by PIO. To implement the keyboard logger, VAIL needs to monitor the data port of the keyboard, so that we can intercept every scan code, then find out which key the user has pressed. Especially for the English continuous typing, it is easy to reorganize the content that the user is inputting.

NIC Interception The NIC is a classic PCI device, which involves MMIO and DMA simultaneously to work. Researchers usually monitor the network activities to provide better protections to the target system [38, 39]. MMIO means that in order to read or write the registers of the NIC as simple as memory access, the OS maps the register region into the kernel space by the page table. And DMA supports a handy way to transfer data between the NIC and the memory

directly without CPU intervention. Before introducing how to intercept them for digital forensics, we would like to present the basic workflow of the NIC.

Transmitting and receiving network packages are the main functions of the NIC. The NIC that our test machine owns, Intel 82578DM, uses two DMA descriptor rings, which are allocated by the NIC driver, to realize these functionalities respectively. As shown in Figure 3, the base address of the descriptor ring is stored in register Transmit Descriptor Base Address (TDBA)/Receive Descriptor Base Address (RDBA), while the total length is saved in register Transmit Descriptor Length (TDLEN)/Receive Descriptor Length (RDLEN). The descriptor ring also has a header pointer, Transmit Descriptor Head (TDH)/Receive Descriptor Head (RDH), and a tail pointer, Transmit Descriptor Tail (TDT)/Receive Descriptor Tail (RDT), to distinguish the used descriptors from the free ones. Each descriptor contains information about the package to be handled and the descriptor itself, and the detail format can be found in [40].

When the OS wants to send a package, it calls the NIC driver to fill the required descriptors with the necessary information, e.g., the buffer address and the size, then update the register TDT. The hardware will be notified and employ DMA to read the package from the memory to transmit. After the package has been processed, the relevant descriptor will be cleared and the register TDH will also be updated. The region between the header and the tail, which is marked as white in Figure 3, is descriptors that are waiting to be dealt with by the hardware. And other descriptors that are gray present those have been handled by the hardware and are free for the software. The receive procedure is much similar to the sending, excepting that the hardware writes received packages into memory, instead of reading packages from memory to the device.

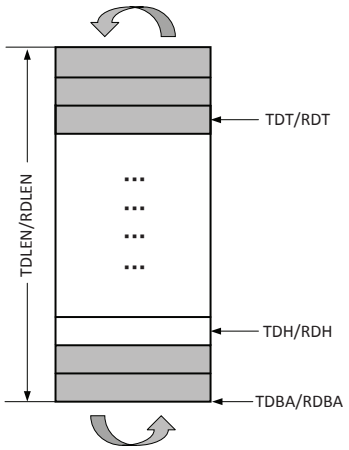


Fig. 3. The Descriptor Ring Structure.

We do not intercept the DMA descriptors directly, because each descriptor has many different fields and the patterns of writing or reading them are not confirmed. Frequent accessing may also bring numerous context switches between the guest OS and the hypervisor, which cause assignable performance overhead. Only monitoring the particular registers is a more clever way. Moreover, all the NIC registers have been mapped into the kernel space by MMIO, which can be intercepted by the EPT. VAIL monitors the updating of the register TDH and copies the content of the package secretly for digital forensics. But the receive process is not as simple as transmitting, since when the driver updates the register RDH, the package has not been really received. When the content arrives, the hardware writes the buffer by DMA, which is not catchable by the page table. To deal with this, instead of intercepting the NIC interrupts, we alternatively save the RDH and compare it with the previous value. Because the buffers that the DMA descriptors point to belong to the NIC driver expressly. Then the descriptors between them are the last received package.

Because the EPT is coarse-grained, which can only monitor a whole page, not a signal byte or word. Operations to other irrelevant registers on the same page may also cause an EPT page fault, which increases potential unnecessary performance overhead. However, this is the nature of handing MMIO by hypervisors and according to our experimental results that will be shown in Section 4, the performance overhead on the target system, either the overall impact or the network influence, is slight and acceptable.

4 Evaluation

This section will first analyze the security of VAIL and then show the experimental results. All experiments are carried out on a Dell Optiplex 980 MT host with a 3.2 GHz Intel i5-650 processor, 2 GB RAM and an Intel 82578DM Gigabit Ethernet NIC. The guest OS version is 32-bit Windows XP SP3.

4.1 Security Analysis

Compared with other VMMs, VAIL removes unnecessary components and only intercepts the critical devices. As shown in Table 1, the total code size of VAIL is only 6626 Source Lines of Code (SLOC), including 4773 SLOC for Silent-Virtualization, 664 SLOC for Memory-Hiding, and 1189 SLOC for I/O interception. However, even if we exclude the Dom0, the Xen hypervisor still has over 100,000 SLOC [41], which is far more than VAIL. The smaller code size contributes more stability of the hypervisor and enhances live forensics processing.

Based on the assumptions we made in Section 2, even there is an OS running on the target machine, leveraging hardware assisted virtualization technique, VAIL can be loaded as an OS driver and inserts the VMM layer dynamically. The attackers cannot damage the static integrity of VAIL because of the protection of

Table 1. The code size of VAIL.

Module Name	Source Lines of Code
Silent-Virtualization	4773
Memory-Hiding	664
I/O Interception	1189
Total	6626

read-only boot media. After the installation completes, VAIL takes full control of the whole machine and owns higher privileged level than the guest OS.

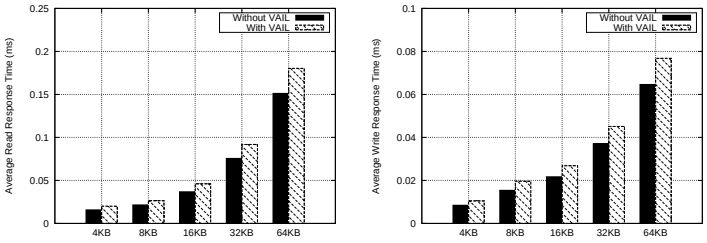
By using Memory-Hiding mechanism, VAIL is not visible in the guest OS. Processes list of Windows Task Manager cannot reflect the existence of VAIL. Because VAIL is loaded as an OS driver, not a user program. If the guest OS tries to scan physical memory, since the part that VAIL located is not accessible from the OS's page table, VAIL keeps safe enough. Even in the worst case, if the compromised OS kernel discovers VAIL, due to the protection of the private page table, it is unable to corrupt the memory of VAIL, neither forcing to unload VAIL nor to map VAIL's GVA to other GPA.

4.2 Performance Overhead

We used the SPEC CPU2000 suite [42] as CPU-intensive benchmarks to evaluate the overall performance overhead that VAIL leads into, and Iometer [43] for I/O-intensive testing. Specifically, as VAIL intercepts the NIC activities, the network latency and throughput were inspected. The general TCP and UDP influences were measured by JPerf [44]. And according to setting up an Apache web server [45], the circumstances with real applications were also considered.

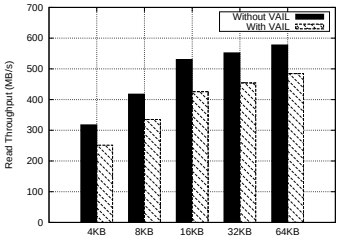
Overall Impact As shown in Figure 5, VAIL only incurred 4.21% performance overhead to the target system on average, which varied from 0.04% to 4.73%, excepting the MCF benchmark had a 23.71% increasing. After our investigation, the main result was that the poor space localization led to frequent Translation Look-aside Buffer (TLB) misses. And every TLB miss led to several times memory access overhead, including the walk-ups of the two-level traditional OS page table and the four-level EPT.

For I/O operations, each PIO results in at least a hypervisor trap and MMIO increases the count of the EPT page faults, which introduces many times of memory accesses as well as the MCF benchmark. VAIL has no direct influences on DMA events. The experimental results of Iometer are shown in Figure 4. We recorded the average response time and throughput of five situations with different sizes of blocks, from 4KB to 64KB, to be randomly read or written. With VAIL installed, the average read response time suffered an increment of 19.3%∼27.6%, while 18.8%∼27.6% for the average write response time. The

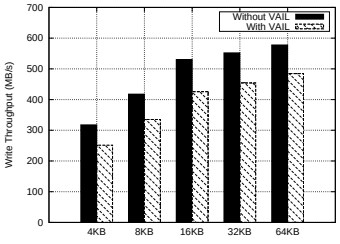


(a) Read Latency

(b) Write Latency



(c) Read Throughput



(d) Write Throughput

Fig. 4. VAIL performance impact - Iometer benchmarks

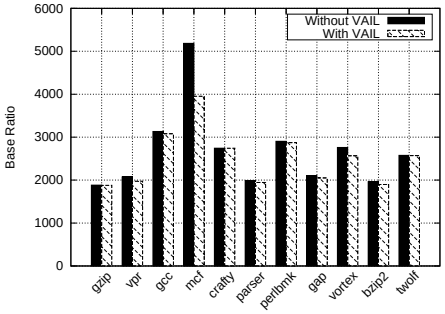


Fig. 5. VAIL performance impact - SPEC CPU 2000

Table 2. VAIL performance impact - Apache web server.

	Without VAIL	With VAIL
Total Time Taken (s)	126.89	127.17
Request Per Second	7.88	7.86
Time Per Request (ms)	126.89	127.17
Transfer Rate (KB/s)	2815.03	2809.01

read and write throughput both had a cut down of 16.1%∼20.7%, because we configured a half of reading and a half of writing in Iometer. An interesting phenomenon was that when the size of the block increased, the performance overhead reduced. The reason might be that, although the assessed memory addresses were different, they shared with the same high level page table, which brought the effect of TLB into playing more efficiently.

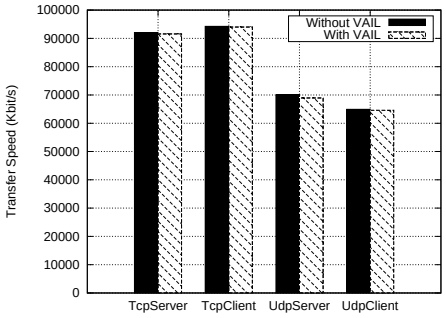


Fig. 6. VAIL performance impact - TCP and UDP

Network Influence In order to distinguish the different effects that VAIL brings on transmitting and receiving, we did not only set the target system as a client, but also as a server. Figure 6 presents the experimental results. Because the network itself has latency, it weakens the impact of VAIL. The throughput overhead of the TCP client case was 0.29%, while the TCP server was 0.52%. The UDP client was 0.46% and the UDP server was 1.52%. The UDP cases were both less than the TCP cases, because the UDP was not a reliable connection protocol, while the TCP had measures to deal with package loss. And the client cases were both better than the server ones slightly. Because the receiving interception needed additional comparison with the previously saved header register.

Finally, we set up an Apache web server to test the overhead on the real applications. From another machine, we tried to acquire the same file for 1000

times with the concurrent level to be 40. HTML transferred during the test was 365.499MB and the total transferred was 365.782MB. The performance overhead was close to the simple TCP/UDP, which was only 0.22%, while the exhaustive data are listed in Table 2. In summary, the additional network interception does not affect the target system very much. VAIL is suitable for the realistic web environments.

5 Related Work

After decades of digital forensics technique development, researchers have proposed many different ways to acquire the system running state. According to the running privileged level that forensics acquisition tools are located, we can divide them into three groups: *App-based*, *Kernel-based*, and *VMM-based*.

Primitive *App-based* acquisition tools existed in user mode only, which behaved as simple applications and relied on the OS completely to get the volatile digital evidences, such as Memoryze [15] and GNU dd [46]. Although being compact and easy to be installed, these *App-based* tools had obvious drawbacks: neither the reliability nor accuracy of obtained evidences can be guaranteed. Once compromised by malicious software, the OS can refuse to return any information or fool the acquisition tools with some incorrect results.

So as to grasp the initiative, *Kernel-based* acquisition tools went deep into the OS kernel, which were loaded as a kernel module. Win32dd [12] was a commonly used memory dump tool that belonged to this type. Investigators can send specific commands from the user level client console to process digital forensics. And modern commercial OSs themselves also provided some convenience for digital forensics. For example, when hardware or software failures occur, the OS will freeze the system state and generate a crash dump with extra debugging information. Kdump [47] achieved this functionality by loading a crash dump specific kernel on the Linux system. *Kernel-based* forensic tools seemed to be more independent and trustworthy, because they shared the same privileged level with the OS. However, the fundamental problem is still not resolved. The OS has a border attack surface that may include a lot of potential vulnerabilities. The cunning criminals can benefit from this and then ruffle the normal digital forensics process.

Finally, by leveraging virtualization technology, *VMM-based* acquisition tools gained higher privileged level than the OS so that they retained full control of the whole computer system. Garfinkel et al., first presented a VMM to inspect the state of the target OS [48]. Subsequently VMScope [49] elaborated this idea to construct the honeypot monitoring system. Based on Xen, the VIX tools suite [7] and Srinivas's work [19] resided the digital forensics examination out of the target system. By running in the special VM, Dom0, both of them had no modification to the guest system during acquisition theoretically. Xen developers even proposed an on-going project with Copy-on-Write technique to obtain the state of VMs [50]. Ruo et al., [9] proposed an asynchronous memory snapshot and forensics using split kernel module, which involved additional environmental

impact on the target system because it needs to modify the guest OS. VMWare Workstation [51] employed process based virtual machine to take a snapshot of the VM's state. But previous *VMM-based* acquisition tools have to either reinstall the target OS or reboot the physical machine, which is inapplicable when performing strict live acquisition in commercial environments.

Both BodySnatcher [20] and HyperSleuth [32] had tried to load an OS driver to insert the hypervisor dynamically, which was quite similar to our VAIL. BodySnatcher used a built-in acquisition OS to obtain the target system's volatile memory. Unfortunately, it needed to suspend the target system during acquisition, thus being intolerable if the target OS had a requirement of 24/7 availability. In addition to this, BodySnatcher's current implementation had highly constrained nature, which was only able to work with Windows 2000 on the single CPU VMWare host, not on real hardware. HyperSleuth utilized hardware assisted virtualization technique to perform sophisticated runtime analysis. The only drawbacks were that some small changes in the memory of the target were induced by the installation procedure and that a powerful attacker in the same network of the target could interfere with the packets containing the memory dump [52].

Hardware acquisition methods are an alternative method in live acquisition. As introduced in Section 3, DMA provides a convenient way for devices to access the memory of the target system. Carrier's work [53] disabled the CPU and performed DMA operations by an acquisition specific PCI card to obtain the volatile memory content. Also, Boileau's work [54] and Martin's work [55] employed firewire protocol to perform DMA operations. While these approaches look likely to finish the acquisition smoothly, Rutkowska's work [56] proved that it is possible to present a different memory view to DMA based acquisition devices through configuring MMIO features on emerging chip sets. Hence, obtained volatile memory content may be quite different from the real content that presented to the CPU. VAIL has not to worry about this because the hardware assisted virtualization technology ensures that the VMM has a broader view than the guest OS. Thus, VAIL can access all the target system content within its own context.

6 Conclusion and Future Work

We have presented VAIL, a virtualization based monitoring system for mini-intrusive live forensics that is able to gather different information of the whole target system, including the CPU state, the memory content, and the continuous activities of I/O devices. By Silent-Virtualization approach, VAIL is loaded as an OS driver. It inserts the hypervisor layer dynamically and resides itself at a higher privileged level than the guest OS, avoiding disturbing the execution of the target system. The safety of VAIL is guaranteed by Memory-Hiding mechanism. The memory that VAIL uses during installation has been hidden and protected by the private page table. We have implemented a proof-of-concept prototype that has been validated on a Windows guest system. The experimental results prove

that VAIL can stably provide the acquisition results with slight performance overhead.

But there are still some limitations existed in VAIL, which need to be improved in the future. The first one is that VAIL only focuses on monitoring the target system, which ignores how to export the acquired evidences appropriately. For example, we can integrate VAIL with an encryption and transmission module, so that the obtained results can be outputted to the removable media or remote computer systems. The second one is that VAIL is loaded as a Windows driver, which is not portable for Linux systems. If we abstract the core idea and make VAIL become completely OS independent, VAIL will have a more promising prospect. The third one is that the current version of VAIL is validated on native systems, though VAIL can work well theoretically in nested virtualization environments. Future work can adapt VAIL to be applicable in cloud environments.

Acknowledgements

This work is supported by the Program for PCSIRT and NCET of MOE, NSFC (No. 61073151, 61272101), the key program (No. 313035) of MOE, and International Cooperation Program (No. 11530700500, 2011DFA10850), and Singapore NRF CREATE S2E2 Program.

References

1. Symantec Corporation: Norton Cybercrime Report. `http://http://now-static.norton.com/` (2012)
2. Yen, P.H., Yang, C.H., Ahn, T.N.: Design and implementation of a live-analysis digital forensic system. In: Proceedings of the 2009 International Conference on Hybrid Information Technology. ICHIT '09, New York, NY, USA, ACM (2009) 239–243
3. Carrier, B.D.: File System Forensic Analysis. Addison-Wesley Professional (2005)
4. Guidance Software, Inc.: EnCase. `http://www.guidancesoftware.com/` (2001)
5. AccessData Group: FTK. `http://www.accessdata.com/` (2003)
6. Buchholz, F.: Pervasive Binding of Labels to System Processes. PhD thesis, Purdue University (2005)
7. Hay, B., Nance, K.: Forensics examination of volatile system data using virtual introspection. SIGOPS Oper. Syst. Rev. **42** (2008) 74–82
8. Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J.T.: Robust signatures for kernel data structures. In Al-Shaer, E., Jha, S., Keromytis, A.D., eds.: ACM Conference on Computer and Communications Security, ACM (2009) 566–577
9. Ando, R., Kadobayashi, Y., Shinoda, Y.: Asynchronous pseudo physical memory snapshot and forensics on paravirtualized vmm using split kernel module. In Nam, K.H., Rhee, G., eds.: ICISC. Volume 4817 of Lecture Notes in Computer Science., Springer (2007) 131–143
10. Savoldi, A., Gubian, P.: Towards the virtual memory space reconstruction for windows live forensic purposes. In: SADFE, IEEE Computer Society (2008) 15–22

11. Sutherland, I., Evans, J., Tryfonas, T., Blyth, A.: Acquiring volatile operating system data tools and techniques. SIGOPS Oper. Syst. Rev. **42** (2008) 65–73
12. MoonSols: Win32dd. `http://moonsols.com/blog/2-blog/9-moonsols-windows-memory-toolkit` (2008)
13. GMG Systems, Inc.: KnTTools. `http://gmgsystemsinc.com/knttools/` (2005)
14. McAfee, Inc.: Fport. `http://www.scanwith.com/download/Fport.htm` (2005)
15. MANDIANT Corporation: Memoryze. `http://www.mandiant.com/products/free_software/memoryze/` (2008)
16. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T.L., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In Scott, M.L., Peterson, L.L., eds.: SOSP, ACM (2003) 164–177
17. AMD Corporation: AMD Virtualization. `www.amd.com/virtualization/` (2005)
18. Intel Corporation: Intel Virtualization Technology. `http://www.intel.com/technology/virtualization/` (2005)
19. Krishnan, S., Snow, K.Z., Monrose, F.: Trail of bytes: efficient support for forensic analysis. In Al-Shaer, E., Keromytis, A.D., Shmatikov, V., eds.: ACM Conference on Computer and Communications Security, ACM (2010) 50–60
20. Schatz, B.: Bodysnatcher: Towards reliable volatile memory acquisition by software. Digital Investigation **4** (2007) 126 – 134
21. Ayers, D.: A second generation computer forensic analysis system. In: Proceedings of the 9th annual digital forensic research workshop. DFRWS (2009)
22. Garfinkel, S.: Digital forensics research: The next 10 years. In: Proceedings of the 10th annual digital forensic research workshop. DFRWS (2010)
23. Wang, Z., Wu, C., Grace, M., Jiang, X.: Isolating commodity hosted hypervisors with hyperlock. In: Proceedings of the 7th ACM european conference on Computer Systems. EuroSys '12, New York, NY, USA, ACM (2012) 127–140
24. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: ACM SIGOPS Operating Systems Review. Volume 41., ACM (2007) 335–350
25. Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., Kato, K.: Bitvisor: a thin hypervisor for enforcing i/o device security. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. VEE '09, New York, NY, USA, ACM (2009) 121–130
26. Rutkowska, J.: Subverting vistatm kernel for fun and profit. Black Hat Briefings (2006)
27. Wojtczuk, R., Rutkowska, J.: Attacking smm memory via intel cpu cache poisoning. Invisible Things Lab (2009)
28. Wojtczuk, R., Rutkowska, J.: Attacking intel trusted execution technology. Black Hat DC (2009)
29. Wojtczuk, R., Rutkowska, J., Tereshkin, A.: Xen 0wning trilogy. Invisible Things Lab (2008)
30. Invisible Things Lab: NewBluePill. `http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html` (2006)
31. Intel, I.: Intel 64 and IA-32 Architectures Software Developer's Manuals. (2007)
32. Martignoni, L., Fattori, A., Paleari, R., Cavallaro, L.: Live and trustworthy forensic analysis of commodity production systems. In: Proceedings of the 13th international conference on Recent advances in intrusion detection. RAID'10 (2010) 297–316
33. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: Security and Privacy (SP), 2010 IEEE Symposium on, IEEE (2010) 380–395

34. Trusted Computing Group: Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module (2011)
35. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Vmm-based hidden process detection and identification using lycosid. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. VEE '08, New York, NY, USA, ACM (2008) 91–100
36. Yu, M., Lin, Q., Li, B., Qi, Z., Guan, H.: Vis: virtualization enhanced live acquisition for native system. In: Proceedings of the Second Asia-Pacific Workshop on Systems, ACM (2011) 13
37. Yu, M., Qi, Z., Lin, Q., Zhong, X., Li, B., Guan, H.: Vis: Virtualization enhanced live forensics acquisition for native system. Digital Investigation (2012)
38. Zhou, Q., Yu, J., Yu, F.: A trust-based defensive system model for cloud computing. In: Network and Parallel Computing. Springer (2011) 146–159
39. Cheng, B.C., Liao, G.T., Lin, C.K., Hsu, S.C., Hsu, P.H., Park, J.H.: Mib-itrace-cp: An improvement of icmp-based traceback efficiency in network forensic analysis. In: Network and Parallel Computing. Springer (2012) 101–109
40. Intel, I.: Intel 82575EB Gigabit Ethernet Controller Software Developer Manual and EEPROM Guide. (2011)
41. Murray, D., Milos, G., Hand, S.: Improving xen security through disaggregation. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM (2008) 151–160
42. Standard Performance Evaluation Corporation: SPEC CPU2000. http://www.spec.org/cpu2000/ (2000)
43. Intel Corporation: Iometer. http://www.iometer.org/ (1998)
44. Free Development software: JPerf. http://sourceforge.net/projects/jperf/ (2011)
45. The Apache Software Foundation: The Apache web server. http://www.apache.org/ (1999)
46. P. Rubin, D.M., Kemp, S.: Gnu dd. http://www.gnu.org/software/coreutils/ (2005)
47. Goyal, V., Biederman, E.W., Nellitheertha, H.: Kdump, a kexec based kernel crash dumping mechanism. In: Linux Symposium. (2005)
48. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS, The Internet Society (2003)
49. Jiang, X., Wang, X.: out-of-the-box monitoring of vm-based high-interaction honeypots. In: Recent Advances in Intrusion Detection, Springer (2007) 198–218
50. Colp, P., Matthews, C., Aiello, B., Warfield, A.: Vm snapshots. Xen Summit, North America (2009)
51. VMware, Inc.: VMware Workstation. http://www.vmware.com/products/workstation/ (1999)
52. Reina, A., Fattori, A., Pagani, F., Cavallaro, L., Bruschi, D.: When hardware meets software: A bulletproof solution to forensic memory acquisition. (2012)
53. Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digital Investigation **1** (2004) 50–60
54. Boileau, A.: Hit by a bus: Physical access attacks with firewire. In: Ruxcon. (2006)
55. Martin, A.: Firewire memory dump of a windows xp computer: A forensic approach. (2007)
56. Rutkowska, J.: Beyond the cpu: Defeating hardware based ram acquisition. In: Blackhat. (2007)