An Effective Memory Optimization for Virtual Machine-Based Systems

Duy Le, *Student Member*, *IEEE*, and Haining Wang, *Senior Member*, *IEEE*

Abstract—Utilizing the popular virtualization technology (VT), users can benefit from server consolidation on high-end systems and flexible programming interfaces on low-end systems. In these virtualization environments, the intensive memory multiplexing for I/O of Virtual Machines (VMs) significantly degrades system performance. In this paper, we present a new technique, called Batmem, to effectively reduce the memory multiplexing overhead of VMs and emulated devices by optimizing the operations of the conventional emulated Memory Mapped I/O in Virtual Machine Monitor (VMM)/hypervisor. To demonstrate the feasibility of Batmem, we conduct a detailed taxonomy of the memory optimization on selected virtual devices. We evaluate the effectiveness of Batmem in Windows and Linux systems. Our experimental results show that 1) for high-end systems, Batmem operates as a component of the hypervisor and significantly improves the performance of the virtual environment, and 2) for low-end systems, Batmem could be exploited as a component of the VM-based malware/rootkit (VMBR) and cloak malicious activities from users' awareness.

Index Terms—Memory management, virtual machine, security.

1 Introduction

As a platform-virtualization software solution, VMM/hypervisor has been widely used for supporting a diverse set of hardware devices and monitoring information between a host machine and multiple guest operating systems (OSes). For high-end systems, VMMs are attractive for server consolidation due to their strong resource and fault isolation guarantees. On a high-end server, the main memory is shared between VMs and is monitored by a VMM [1], [2]. The bottleneck of the system lies in VM multiplexing, which is dependent on system capacity features, including memory slot availability, additional power consumption, and the memory-sharing mechanism of VMM. Here the memory-sharing mechanism is known as page sharing, memory compression, or memory I/O multiplexing. Thus, the memory usage inside VMs and the memory-sharing mechanism in VM multiplexing are critical to a server's performance. As a result, commodity VMMs require an effective memory-sharing mechanism between VMs and their host, such as optimizing frequent paging and memory that is mapped for virtual I/O devices.

For low-end systems, such as mobile netbooks, laptops, or client desktops, a VMM provides a high-level OS interface for application programming via traditional real-time APIs, and it also provides the ability to run programs on different OS platforms. However, due to the small capacity of a low-end system, multiple VMs cannot be installed on a single host. Thus, in terms of performance, memory sharing is not a critical issue for low-end systems. Nevertheless, in terms of security, malware may exploit virtualization techniques including memory sharing to completely control VMs on low-end systems. SubVirt [3] and BluePill [4] are typical examples of VMBR that attempt to append a thin VMM as a middleware between a running OS and hardware devices. The success of VMBR relies on two factors: compromising devices and hiding malicious behaviors. More specifically, VMBR requires virtual devices to intercept the I/O operations of a victim OS, and then VMBR must cloak its malicious behaviors, which could include system modification violation or performance degradation.

Therefore, reducing the overhead in memory multiplexing of VMs will not only improve the performance of high-end systems, but also help us understand the possibility of cloaking malicious VMBR behaviors in low-end systems. In this paper, we present Batmem, an effective technique to improve the performance of the Memory Mapped I/O (MMIO)—a conventional memory exchange mechanism—by reducing the overhead and redundant memory regions during the multiplexing of VMs. The key component of Batmem is a dynamic circular buffer that coalesces memory partitions to be written into the reserved memory areas of virtual devices. We also employ a compression algorithm to reduce the allocated memory regions used in such I/O writing. For either high-end or low-end systems, Batmem is applied on virtual devices, such as the Video Graphics Array (VGA), Network Interface Controller (NIC), and Universal Host Controller Interface (UHCI). In particular, for high-end systems, we use selective micro benchmarks to evaluate system performance at the device level. For low-end systems, Batmem functions as a VMBR component. To validate its effectiveness in concealing VMBR activities from users' observations, we evaluate the performance of selected user applications while maintaining two malicious services: keylogger and data transmission.

With Intel Virtualization Technology (VT) support, we implement Batmem and conduct experiments on emulated devices for both Windows and Linux systems. Our

• *The authors are with the Department of Computer Science, The College of William and Mary, PO Box 8795, Williamsburg, VA 23185. E-mail: {duy, hnw}@cs.wm.edu.*

Manuscript received 13 July 2010; revised 11 Oct. 2010; accepted 12 Oct. 2010; published online 18 Jan. 2011.
Recommended for acceptance by A. Zomaya.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-07-0414.
Digital Object Identifier no. 10.1109/TPDS.2011.37.

experimental results show that Batmem effectively reduces the overhead of the I/O multiplexing in virtual devices while eliminating the redundant memory regions used in a memory exchange. For high-end systems, Batmem significantly improves the performance of the virtual environment, but for low-end systems, Batmem could be exploited by VMBR to cloak its malicious activities.

The implementation of Batmem is based on Kernel Virtual Machine (KVM) [5], instead of other open source hypervisors such as Xen [6] or Lguest [7]. Even though Xen supports fault containment and performance isolation by partitioning physical memory among multiple VMs and allows unmodified guest OSes to run on a VT-x supported host, its particular domain-based architecture makes it impossible to compare with other light-weight hypervisor architectures, such as KVM or lguest, in terms of driver domain model and performance. In contrast, KVM inherits lguest's flexibility and turns a Linux kernel into an in-kernel hypervisor, in which OSes can directly run on the hardware and take advantage of VT-x.

The remainder of this paper is organized as follows: in Section 2, we survey related work. In Section 3, we detail the design and implementation of Batmem. In Section 4, we evaluate the performance of Batmem at both low and high system levels (i.e., device and user levels). In Section 5, we discuss the relevant security aspects of Batmem from both offense and defense sides. Finally, we conclude in Section 6.

2 Related Work

We briefly describe the previous works that are related to ours from four different perspectives: MMIO and memory compression, VMBR behaviors, malicious devices and drivers, and end-user behaviors.

2.1 MMIO and Memory Compression

Optimizing MMIO and memory compression has been studied before. McKenney [8] discussed the possibility of improving functionalities of MMIO in different symmetric multiprocessing (SMP) architectures. Huggahalli et al. [9] implemented a framework to indirectly improve MMIO by delivering inbound I/O data directly into processor caches. Xia et al. [10] recently proposed an intermediate technique to improve the performance in passthrough I/O. However, these optimizations only target regular NICs, not other PCI devices. Considering a regular architecture of the MMIO circular buffer, we improve its performance by dynamically monitoring the reserved memory regions of devices.

For the memory compression, the WKdm algorithm proposed by Wilson et al. [11] is an effective hybrid method that utilizes both statistical and dictionary-based techniques. Gupta et al. [12] recently implemented a hypervisor extension that supports multiple in-memory compression algorithms. Their work significantly reduces memory consumption not only in running virtual applications, but also across different VMs. Instead of using such algorithms, we apply Run-Length Encoding (RLE) [13] to compress memory partitions due to its simplicity in encoding with the threshold run, achieving the reduced time complexity in comparison to other algorithms.

2.2 VMBR Behaviors

The previous studies of VMBR primarily focus on how a computer system can be infected by VMBR [3]. Recent research targets the malicious functionalities of VMBR on VM-based systems, such as memory shadowing [14] or fingerprinting methods [15]. In addition, the approach to analyzing malware behaviors through VM-based systems has been studied, such as a tainting technique [16] and an active monitoring library [17]. However, since these previous works are offline analyses, they cannot be deployed on end-user systems for online detection. Instead of revisiting the VMBR problem, our work analyzes the potential VMBR challenges and verifies the possibilities of VMBR to hijack computer systems by adapting its behaviors.

2.3 Malicious Devices and Drivers

With respect to devices and drivers, a few malware models have been developed on the exploitation of particular hardware and software components. King et al. [18] presented a substantial design space in malicious circuitry to build a flexible and hard-to-detect malicious processor. To hijack a system, Embleton et al. [19] proposed a System Management mode Based Rootkit (SMBR) that exploits an obscure mode on Intel processors. In terms of modifying the kernel driver model, the released KVM patches also use an MMIO coalescent technique [20] in one KVM run session. However, like the virtual passthrough I/O of Xia et al. [10], these models neither use the dynamic batching method nor cover the UHCI device. In contrast, we focus on drivers that can be maliciously modified for malware's purposes and conduct experiments on the emulated E1000, instead of the Realtek NIC.

Regarding hardware optimized for virtualization, recent work focuses on building efficient virtual awareness devices and improving the hardware performance by modifying the network subsystem [21]. However, such improvements require the modification of the Xen's driver domain model. Although our prototype is developed on KVM with supported VT-x, Batmem can be applied in other virtual context switching models to narrow the performance gap of the virtualization and verify the security holes of virtual devices.

2.4 End-User Behaviors

In accordance with user-perceived performance, we classify user behaviors into three different groups: screen-based, net-based, and filesystem-based. Note that the performance metrics are chosen according to their acceptable validity and reliability in previous studies [22], [23].

Screen-based behavior is a variation of the end-user screen interaction, which is described by an instantly estimated frame-per-second (FPS) metric. The higher the FPS values, the closer the matching between a real VGA and an emulated VGA.

Net-based behavior is a variation of the end-user network activity. We use performance metrics such as virtual capacity and packet delay of a NIC to quantify the variation. Here virtual capacity is defined as the maximum data transfer rate over the virtual NIC, specifically between the guest OS and outside networks.

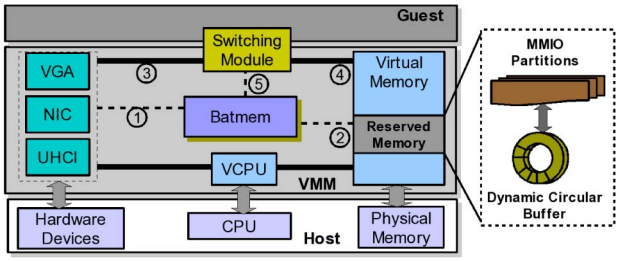

Fig. 1. Virtual environment overview with Batmem.

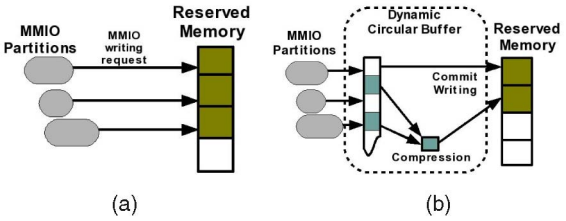

Fig. 2. (a) Regular MMIO partitions written into reserved memory. (b) Batmem enhances writing speed by using dynamic circular buffer with compression.

Filesystem-based behavior affects user interactions on virtual filesystems, such as emulated USB storage. The I/O activities of end-users will be affected by the bandwidth of filesystems.

3 System Design and Implementation

In this section, we first detail the system design of Batmem. In particular, Batmem improves the speed of MMIO write by using 1) a dynamic circular buffer to group write requests, and 2) a compression to minimize written memory partitions into the reserved memory. Then, we describe its implementation and related malicious services on KVM.

3.1 System Design

In our design, Batmem participates in writing MMIO with other virtual components, such as the main memory, devices, and context switch. As shown in Fig. 1, a virtual CPU (VCPU) takes control of reading data from the device and writing MMIO data to memory. Batmem intercepts such an exchange by monitoring virtual device status ① and I/O device operations ② to control the MMIO writing on the reserved memory area. The context switch, known as a switching module, controls I/O requests sent from guest OS device drivers to virtual devices ③. With VCPU, the switching module conducts the MMIO writing to virtual memory ④. To allow Batmem to properly participate in the MMIO writing, we need to establish a connection between Batmem and the switching module ⑤. This connection registers Batmem into a contact list of the switching module, making Batmem capable of monitoring I/O requests. These I/O requests are issued from the guest OS device driver controller, passing through the context switch in each MMIO session.

In each MMIO session, the issued MMIO requests are executed by I/O instructions at the switching module and VCPU. Such requests hold the address and size of the shared page of memory for each MMIO writing. When the Batmem/switching module connection is established, Batmem exchanges with the switching module to obtain information about the MMIO session, including the addresses belonging to MMIO page and reserved memory partitions. Each partition holds information about its size, address, and status (registered or unregistered). The registered partition introduces an occupied memory area, which is allocated and used by Batmem. The unregistered partition presents an extensible unoccupied memory area. The unregistered partitions are extended and used when current allocated memory partitions for Batmem are overrun by numerous arrivals of writing requests. Through Batmem, these partitions are registered or unregistered with the switching module. Note that such partitions are associated with another assigned memory area known as a batched buffer. To increase the MMIO writing speed from the reserved memory for a device to the main memory, we create the batched buffer as a dynamic circular buffer to store all MMIO partitions as a batch for each writing session.

The dynamic circular buffer structure is built on an ordinary circular buffer to prevent buffer underruns when devices perform numerous write-backs to the main memory. As shown in Fig. 2a, the dynamic circular buffer is a list of memory regions, where each element can be freed or ready to be filled upon receiving a writing request. To batch MMIO partitions, each buffer element needs to record the physical address and size of the mapped memory. Upon receiving notification from the switching module at session completion, Batmem informs the dynamic circular buffer to group all current partitions in the buffer. Instead of sequentially writing into the reserved memory, which is time consuming and may slow other devices' I/Os on the main memory, Batmem simply completes an MMIO by copying the available buffer to the reserved memory. Since the buffer to be copied is in a mapped memory area that lies on the same main memory area, this copying is obviously much less expensive than the regular sequential MMIO writing. In order to eliminate buffer underruns when the requests of other devices fall behind, Batmem adjusts the size of the dynamic circular buffer by appending a number of free elements. Note that although a circular buffer has been widely used in sharing memory mechanisms, our improvement goes beyond the design of data structure by dynamically associating its functionalities with a memory compression in each writing session.

To reduce the memory footprint in the reserved memory, as shown in Fig. 2b, Batmem employs RLE to compress the batched memory regions in the dynamic circular buffer. Because such a compression is useful only when the compression ratio is high, we need to determine the regions to compress. At the beginning, instead of compressing an entire region, we just compress the first half of a region. Batmem defines a threshold to compare with the compression ratio of the first half region's. If the measured compression ratio is higher than the given threshold, the compression is effective. The rest of the region is compressed and then is written into the reserved memory. Otherwise, the entire uncompressed batched region is committed to writing into the reserved memory as usual.

In the reserved memory, Batmem marks the compressed regions to differentiate them from others. When a read request from the VM accesses a compressed region,

Batmem automatically decompresses and returns the region. Note that a compressed region and its mark remain intact until it is discarded or overwritten by new regions.

Batmem groups and compresses MMIO partitions without affecting the MMIO session. After the partitions to be grouped are successfully registered, the switching module notifies both Batmem and the devices to activate the MMIO batching. Using the device status provided by the switching module, Batmem can differentiate the device and its registered memory partitions from other devices that are not actively executed. Note that when VMM is initialized, to monitor the virtualized main memory, VCPU needs to map all first pages of device memory structures to the main memory. Since the regular size of the mapped memory is given and specified by VMM, Batmem maps the offset of this dynamic circular buffer to the first page of the main memory to easily locate the buffer in each MMIO session.

3.2 Implementation

We implement Batmem on KVM, an open-source-based VMM/hypervisor, which operates as a subsystem leveraging the virtualization extension. The recent KVM version works as a Linux kernel module running under a VT-x supported host. As a benefit of the Linux kernel architecture, KVM can perform or schedule the OS as a Linux process. For high-end systems, Batmem is added into KVM as a module that maintains a capability to monitor multiple running VMs. For low-end systems, Batmem is implemented as a VMBR component that attempts to conceal the presence of running malicious services from the end-user. The implementation on KVM includes two main parts: Batmem and malicious services.

3.2.1 Batmem

Batmem takes advantage of the standard ioctl() functions under the Linux kernel to allocate, register, and unregister memory partitions. Such functions are immediately initialized with the KVM core module when the host is started. To protect allocated partitions from other processes that do not involve the MMIO writing session, the KVM core must be secure before and after each use. To secure the KVM core and schedule legitimate processes, we use a semaphore. We monitor a memory partition by using flag and side values that represent the results and side effects of the current batching process. The results consist of the registered partition information, including device identifications, the reserved memory size, and the circular dynamic buffer address. The side effects are considered to be either memory allocation latencies or buffer overrun circumstances. For the dynamic circular buffer, we start with a default size of 100 elements. If the number of partitions being used reaches the current buffer size, the buffer size is incremented by 10 elements. We choose these numbers to strike a balance between system memory usage and buffer overrun circumstances.

We need to minimize the overhead produced by compression/decompression operations. The overhead is measured in terms of the execution times of various functions involved in MMIO/Batmem, where we enable each function in isolation and evaluate its execution time. Fig. 3 shows the average of total overhead imposed by major Batmem/MMIO operations, corresponding to the different compression ratio thresholds. As expected, the overhead grows with the increase of the compression threshold value. The overhead growth is primarily due to the increased number of batched regions that are available for compression. More specifically, we conduct multiple experiments using different compression ratio values as integers in a given range with an estimated error of the standard deviation. In each experiment, we maintain consistent batched memory regions as input for the compression. As a result, the ratio threshold of 5 percent is selected as the default value, with which the compression/decompression module only adds 1-1.5 percent overhead to the entire system.

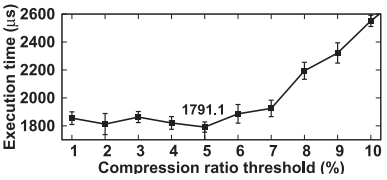

Fig. 3. Average of total MMIO functions' overhead.

For VGA and Rtl8193 NIC, we intercept their original registration functions to monitor both device information and writing processes. The interception directly points original registrations to our new registration routines. Therefore, we can perform batching on the mapped memory partitions of these devices upon receipt of their statuses. Since Batmem conducts the batching, this interception makes our new routines transparent to the guest OS device drivers.

For UHCI, we modify its registration and create its reserved memory for MMIO. To have the KVM UHCI function as a regular PCI device, we modify UHCI registration based on the core registration of a standard PCI device. However, on the KVM, UHCI is emulated without a reserved memory area. We create a reserved memory for UHCI on the main memory and add it to the contact list of the switching module. To allow UHCI to operate MMIO, we modify the UHCI initialization by directly assigning the destinations of its I/O operations to the new reserved memory. Note that such modifications do not affect the fundamental UHCI architecture.

3.2.2 Malicious Services

We implement two malicious services as parts of VMBR: keylogger and data transmission between malware.

First, using the kernel keylogger concept [24], we implement the keylogger to compromise both the data buffers and I/O functions of the emulated keyboard controller. Since the emulated keyboard controller is operated as a kernel module within KVM, we need to recompile KVM with the keylogger to activate the service. To hijack a keystroke data buffer, the keylogger first checks the buffer availability, then performs its own read/write functions to copy the keystroke data to its buffer. The checking is executed via generated interrupts at an emulated serial port of KVM. We implement a small module to store the copied keystroke data as readable log files under the host. Although the keylogger is not fully functional, such as encrypting keystroke data or sending it

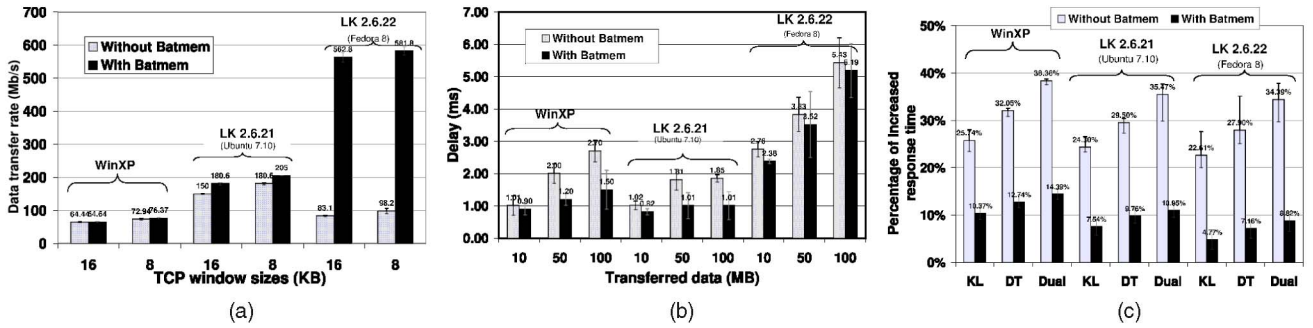

Fig. 4. (a) NIC virtual capacity (larger is better). (b) UDP packet delay (smaller is better). (c) Web browser response times with malicious services (smaller is better).

out to networks, we believe that its interception precisely represents a regular VMBR's keylogger service.

Second, to illustrate a data exchange between two pieces of malware, we implement a data transmission service for exchanging data between the user level of the guest OS and the kernel level of the host. The exchanged data are guest OS sensitive information, such as a Windows registry structure or a Linux filesystem map. An inside-the-guest malware functions at the user level of the guest OS. Another out-of-the-box malware operates at the kernel level of the host, more specifically, inside KVM. These two pieces of malware attempt to periodically send and receive data to each other by using implicit communication methods, such as interrupts, ports, or devices. The inside-the-guest malware cannot modify device drivers, and using interrupt-based or port-based communications is more challenging than a device-based method. We implement a simple protocol based on TCP/IP that allows both malware to send and receive data packets via an emulated NIC. The emulated NIC is initialized when the system is started with activated network services. VMBR can immediately perform this data transmission service afterwards.

4 Experimentation

We use benchmarks and sample payloads to evaluate the effectiveness of Batmem. First, for high-end systems, we conduct experiments with KVM/Batmem for three types of guest OSes: Windows XP, Ubuntu 7.10 Linux Kernel (LK) 2.6.21, and Fedora 8 LK 2.6.22. Each type has two guest OSes, for a total of six guest OSes. We run these guest OSes on a Tank GT20 server that includes a quad-core 2.0 GHz Intel Xeon processor and 4 GB RAM. Each guest OS uses 512 MB shared memory. We use benchmarks to examine the operations of intercepted devices. Second, for low-end systems, our evaluation includes two parts: analyzing the modules of Batmem based on their sizes and complexities, and measuring the varied runtime application behaviors when malicious services are activated. The running host consists of Intel 2.0 GHz and 1 GB memory, in which a shared 512 MB is for a guest OS.

4.1 High-End Systems

We use selected device level benchmarks to verify the effectiveness of Batmem on NIC, VGA, and UHCI. Our experiments are conducted in two scenarios: with and without Batmem. Each experimental result is an average of eight independent measurements along with an error estimate specified by the sample standard deviation. Due to the different running services involved and differences between UDP and TCP in terms of reliability and weight, we use IPerf [25] to measure the two parameters of virtual NIC, i.e., virtual capacity and UDP packet delay, which correspond to the net-based behaviors. The measurements are conducted under different benchmark configurations. We cluster the performance results into different groups based on the running OS.

Fig. 4a shows that Batmem works more effectively in Linux than in Windows. The results show that Batmem increases the virtual capacity of Windows by only 0.05 percent. However, these virtual capacity values are varied in Linux. In the LK 2.6.22, the virtual capacity is significantly improved by 490 percent, but only by 16.5 percent in the LK 2.6.21. These improvements are due to MMIO partitions, which belong to MMIO requests of the virtual NIC and are completely grouped by Batmem. Such grouping increases the data written into the main memory, and thus increases the virtual capacity. Note that without Batmem, the virtual capacity of the vanilla LK 2.6.22 is even less than that of the LK 2.6.21. The reason is that the vanilla LK 2.6.22 system applies some modifications on the TCP congestion control of the LK 2.6.21. On one hand, the beneficial modifications consist of merging sampling RTT, recomputing RTT updates, and resizing option fields with flag bits. In particular, the TCP socket buffer is required to consider invalid zero timestamps in communication with the RTT sampler upon the ACKed TCP retransmission request, and hence slightly affects its data transfer rate. On the other hand, these modifications increase the number of MMIO requests and reduce the amount of data written into the main memory for each request. The reduced amount of written data lowers the virtual capacity. Therefore, we believe that these modifications of the TCP congestion control significantly improve the virtual capacity in the LK 2.6.22 when Batmem is active.

Fig. 4b shows the effectiveness of Batmem in reducing UDP packet delays. The UDP packets are transmitted between the guest OS and the host through the virtual NIC. We conduct the experiments with different amounts of transferred data. Our experimental results demonstrate that Batmem helps Windows reduce the UDP packet delay up to 83 percent. In Linux systems, we observe that Batmem also reduces the UDP packet delay in the LK 2.6.21 by 45 percent,

TABLE 1
VGA Bandwidth (Frames/s, Larger Is Better)

	Mem cache ON	Mem cache OFF
WinXP	37.07 ± 7.7	33.41 ± 3.5
WinXP+Batmem	45.17 ± 6.15	44.81 ± 8.65

TABLE 2
Win UHCI Bandwidth (KB/s, ntfs, Larger Is Better)

	Write		Read	
	Sequential	Random	Sequential	Random
WinXP	156 ± 5.2	126 ± 0.5	122 ± 3.2	170 ± 0.6
WinXP+Batmem	480 ± 6.1	351 ± 0.4	391 ± 4.6	242 ± 0.7

but by just 13 percent in the LK 2.6.22. As expected, in all the systems, Batmem works less effectively with the increase of transferred data because the NIC device driver progressively issues MMIO requests under such an increase. More specifically, the more MMIO requests are issued, the more partitions are reallocated. Consequently, Batmem induces more overhead to group the partitions.

We use 3DBench [26] to measure the VGA memory bandwidth in Windows, which corresponds to the screen-based behavior. The benchmark intensively executes 3D routines that require aggressive I/O data exchanges on the VGA card. These data exchanges depend on three major factors, including processor speed, VGA bus size, and memory cache. Since the processor speed and the VGA bus size cannot be changed, to observe the variations of the VGA bandwidth, we conduct experiments in two cases, with and without memory cache. As shown in Table 1, for 60-100 seconds, Batmem helps the Windows system increase the actual VGA bandwidth, represented by FPS, in both cases by 20-33 percent.

We use the SiSoftware [27] in Windows and Bonnie++ [28] in Linux to measure the UHCI I/O performance, which corresponds to the filesystem-based behavior. To avoid a sensitivity of the filesystem workload that may affect the overall performance of I/O, we consistently maintain a data file for such measurements. Table 2 shows the improvement of read/write in Windows when Batmem is active. In particular, Batmem increases the I/O speed up to 220 percent in the sequential mode and 170 percent in the random mode. In Linux, to measure the operations of read, seek, and delete, we create a 128 MB file, clear the cache, and assign a 4 KB chunk for each operation. For writes, we create an empty file and keep writing 4 KB data chunks to the file until the file size reaches 128 MB. As shown in Table 3, Batmem takes advantage of the asynchronous write in the ext3 filesystem when the number of MMIO sessions is increased, and thus increases the amount of exchanged data for a period of time.

TABLE 4
VMBR Module Size (Bytes)

	Batmem	Malicious Services		Installation procedure
		Keylogger	Data transmission	
Code	12,038	7,415	3,624	2,617
Module	13,332	9,194	8,936	-

Moreover, Batmem increases the I/O speeds from 7 to 15 percent in the LK 2.6.21 and from 95 to 164 percent in the LK 2.6.22, respectively. The accelerations for the LK 2.6.22 are significant when Batmem is active. This is because patches are applied on the LK 2.6.22 to optimize inode read/write functions of the ext3 filesystem, thereby increasing the speed of I/O requests. In fact, the improvement of the I/O request speed enhances the host/virtual I/O context switch. Consequently, Batmem can accelerate the MMIO writing and increase the UHCI I/O bandwidth.

We also conduct experiments to evaluate memory saving on the system. Our results show that the compression component can save up to 5 percent of memory. Since the saving is not significant, we plan to employ more effective compression algorithms for greater improvement in the future.

4.2 Low-End Systems

4.2.1 Module Examinations

We examine our implemented modules on VMBR, including Batmem, malicious services, and the VMBR installation procedure, in terms of their size and complexity features. These modules must limit their sizes and complexities to hide themselves on systems. Since our VMBR is built on KVM, whose original size is given, our focus is on these new modules.

First, the Batmem module includes 1) vector structures, which form the dynamic circular buffer, 2) shared libraries, which consist of memory interactions with devices, and 3) a compression buffer, which supports memory compression. Even without applying source code optimization methods, as shown in Table 4, we observe that the module size of Batmem remains almost the same after the compilation (12 KB of source code and 13 KB of binary code). The slight difference between the two numbers is due to the use of the KVM shared memory library.

Second, of the malicious services, the keylogger implementation is more complex than the data transmission service. As a Linux kernel module, the keylogger uses low level kernel I/O functions to lock, read, and write the keyboard data. For data transmission, the inside-the-guest module benefits from high level functions to maintain its communication, while the out-of-the-box module uses

TABLE 3
Linux UHCI Bandwidth (KB/s, File Size = 128 MB, Chunk Size = 4 KB, ext3, Larger Is Better)

	Sequential per character		Random	Sequential create			Random create		
	Output	Input	Seeks	Create	Read	Delete	Create	Read	Delete
LK2.6.21	3155± 4.3	2007± 2.8	132.5± 1.1	232± 2.4	477± 6.3	3101± 3.3	348± 1	475± 2	804± 2.0
LK2.6.21+Batmem	3560± 5.6	2309± 3.1	150.9± 2.0	252± 2.0	508± 7.2	3448± 8.4	374± 7.2	505± 4.1	864± 3.1
LK2.6.22	4010± 4.1	3832± 7.2	177.1± 6.5	438± 1.4	1055± 5.4	5480± 9.5	717± 7.2	1041± 6.6	1731± 5.8
LK2.6.22+Batmem	9309± 4.5	7924± 6.5	469.2± 4.2	885± 2.0	2143± 9.2	10724± 12.2	1453± 7	2141± 5.1	3470± 4.4

primitive kernel read/write functions. Therefore, the binary size of the data transmission module is significantly expanded compared to the keylogger module. This comparison clearly shows the advantage of using low level library functions for malicious module implementations.

Third, we consider the VMBR installation as a procedure, instead of a part of the malicious module. This procedure functions as a script, which includes essential initializations on KVM and devices, to instantly invoke KVM when the host is started. More specifically, since we only insert and activate this procedure script at the end of the boot sequence, the fundamental structure of the host boot sequence is not changed. In some cases, users may recognize a variation of the guest OS screen resolution because the emulated VGA is not automatically detected. However, this gap can be resolved if attackers retrieve accurate hardware VGA device information to properly configure the guest OS resolution.

4.2.2 User-Level Experimentation

We run the selected user-level applications on guest OSes under two different conditions: with and without malicious services. The performance metric we used is application response time. As one of the most popular Internet applications, web browsers are sensitive to response time. Our selected applications include Internet Explorer in Windows and Firefox in Linux. We differentiate the response times in two cases, with and without Batmem, on compromised systems running malicious services. Note that we do not change the configuration of the web browsers during the experiments, and all web browser caches are cleared before each test to avoid possible side effects.

In our experiments with malicious services, malware is executed either separately as a single service or simultaneously as a dual one (**Dual**). For the keylogger (**KL**), we use AutoHotkey [29] and Autokey [30] to generate keystroke patterns. For data transmission (**DT**), a connection is automatically established to exchange files between two malicious components. We conduct these experiments in two scenarios, without and with Batmem. Each experimental result is compared with the response time of a vanilla system (i.e., the base value $\mathcal{T}$). The lowest $\mathcal{T}$ is 18.44 seconds in LK 2.6.21 and the largest $\mathcal{T}$ is 23.62 seconds in Windows.

Without Batmem, through web browsers, we access a local website, download, and store a given data file into a USB drive. As shown in Fig. 4c, running malicious services significantly increases the user-perceived response times. For example, with a dual service, compared to the corresponding $\mathcal{T}$, the response time is increased by 38.47 percent in Windows and 35.76 percent in Linux.

With Batmem, we repeat the previous tests. As expected, Batmem effectively reduces VMBR overhead in both Windows and Linux. Thus, the user-perceived response times of the web browsers are greatly decreased, which is evidently shown in Fig. 4c. For Windows systems, the reduction is around 60 percent, while for Linux systems, the reduction is up to 80 percent.

Overall, our results clearly demonstrate the capability of Batmem in concealing VMBR's activities from user awareness. The overhead reduction by Batmem in Windows is not as much as that in Linux systems. We believe that this is due to mainly non-optimization of device context switches and I/O system calls in Windows systems.

5 Discussion

In this section, we discuss the reliability and security issues related to Batmem in the virtual environment.

5.1 Reliability

The dynamic circular buffer and memory compression techniques of Batmem can be applied to other hypervisors because they do not depend on a particular hypervisor architecture. Batmem only attempts to improve the speed of MMIO write by monitoring I/O functions on selected devices. More specifically, while the context switch and reserved memory areas are two primary components of the hypervisor, Batmem only optimizes their memory I/O exchanges and does not modify their fundamental operations. Therefore, the operations of these original components are not affected by Batmem.

For memory compression, the actual benefit is determined by a trade-off between its overhead and compression ratio. While the chosen compression ratio threshold of 5 percent is not reasonably high, we believe that it is appropriate because the total system overhead is only increased by 1-1.5 percent. As expected, the compression behavior highly depends on the chosen algorithm. Although the applied RLE is less effective than WKdm and/or Lempel-Ziv in terms of compression ratio [11], we also believe that the prototype of Batmem shows the potential of using such a simple technique to reduce memory redundancy in a virtual support system.

5.2 Security

In high-end systems, Batmem is embedded into the hypervisor without violating the security design of the hypervisor. System administrators can protect Batmem from other malicious accesses inside VMs by placing Batmem as a read-only component within a protected memory area of the host. Such a technique follows the similar approach of memory shadowing proposed by Riley et al. [14]. As a result, Batmem is protected in high-end systems.

For low-end systems, we discuss the challenges to protect a low-end system from an installation of VMBR, as well as to detect its presence, in the rest of this section.

5.2.1 Preventing VMBR Installation

To protect the host boot sequence from malicious modifications of VMBR, we can employ software or hardware solutions. Software solutions secure BIOS or boot processes by using encryption or out-of-the-box verification. Attackers need to retrieve the BIOS information to properly configure virtual devices when the host is started. Encryption methods prevent this retrieval by encrypting/decrypting the BIOS information upon its exchange among legitimate system components. Out-of-the-box methods use the checkpoint verification technique, which compares system snapshots between suspicious and legitimate boot sequences to discover the malicious modifications. In general, hardware solutions can be built on a tainting technique that monitors exchanged data among legitimate

system components. Those suspicious uses of tainted data will be considered as illegitimate. However, for a low-end system, both software and hardware solutions are difficult to apply because they either need to reboot the system for the snapshot comparison or degrade the system performance by their aggressive verifications of primitive data.

5.2.2 Detecting VMBR/Batmem

Since Batmem is operated as an embedded component within VMBR, detecting its presence is challenging. However, as we mentioned in Section 3, to easily locate the buffer of each MMIO session, Batmem maps the offset of the dynamic circular buffer to the first page of the main memory. This design may motivate defenders to scan and compare the content of the device memory and the first page of main memory to determine grouped regions, and hence, detect the presence of Batmem. Unfortunately, aggressively checking memory partitions is very expensive, leading to significant performance degradation.

As an alternative, we can check the local time source to detect VMBR [31]. However, this method is not very robust since attackers can evade detection by using other similar approaches as Batmem to cloak their malicious activities on virtual components. In contrast, Garfinkel et al. [32] show a possibility of detecting VMBR without using time-based techniques. Nevertheless, they target highly resource constrained VMBRs [3], and the flexible and small ones like Bluepill [4] are not considered.

We can also exploit a vulnerability of KVM by checking shutdown conditions of the VCPU triple faults at the user level [33]. The effectiveness of this technique highly depends on possibilities to conceal such shutdown conditions of attackers.

6 Conclusion

We have presented the design and implementation of Batmem, a technique that significantly reduces the overhead of the conventional memory exchange mechanism MMIO. To demonstrate its feasibility, we build Batmem in KVM and conduct experimentation in both high-end and low-end systems. For the high-end systems, we evaluate the performance improvement of virtual devices. For the low-end systems, Batmem functions as a VMBR component. Our experimental results on Windows and Linux show significant performance improvements with the use of Batmem in device-level benchmarks and user-level applications.

We believe that Batmem will help researchers to better understand the critical issues of memory sharing and VMBR in both high-end and low-end virtual support systems. We hope that our work will also motivate system designers to carefully evaluate security gaps at the real/virtual boundary in designing devices for virtual environments and to pay more attention to the threats posed by the adaptive behaviors of VMBR.

References

[1] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K.G. Shin, "Performance Evaluation of Virtualization Technologies for Server Consolidation," Technical Report HPL-2007-59, HP Labs, 2007.

[2] M.R. Marty and M.D. Hill, "Virtual Hierarchies to Support Server Consolidation," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007.

[3] S.T. King, P.M. Chen, Y.-M. Wang, C. Verbowski, H.J. Wang, and J.R. Lorch, "SubVirt: Implementing Malware with Virtual Machines," *Proc. IEEE Symp. Security and Privacy (SP '06)*, 2006.

[4] J. Rutkowska, "Introducing Blue Pill," June 2006, http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html, Oct. 2010.

[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: The Linux Virtual Machine Monitor," *Proc. Linux Symp.*, 2007.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03)*, 2003.

[7] "Lguest: The Simple x86 Hypervisor," http://lguest.ozlabs.org, Oct. 2010.

[8] P.E. McKenney, "Memory Ordering in Modern Microprocessors, Part II," *Linux J.*, vol. 2005, no. 137, p. 5, 2005.

[9] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, 2005.

[10] L. Xia, J. Lange, P. Dinda, and C. Bae, "Investigating Virtual Passthrough I/O on Commodity Devices," *ACM SIGOPS Operating Systems Rev.*, vol. 43, no. 3, pp. 83-94, 2009.

[11] P.R. Wilson, S.F. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," *Proc. Ann. Conf. USENIX Ann. Technical Conf. (ATEC '99)*, 1999.

[12] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, and A. Vahdat, "Difference Engine: Harnessing Memory Redundancy in Virtual Machines," *Proc. Eighth USENIX Symp. Operating System Design and Implementation (OSDI)*, 2008.

[13] "Run-Length Encoding Algorithm," www.data-compression.info/Algorithms/RLE, Oct. 2010.

[14] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," *Proc. 11th Int'l Symp. Recent Advances in Intrusion Detection (RAID '08)*, 2008.

[15] X. Chen, J. Andersen, Z.M. Mao, M. Bailey, J. Nazario, and F.J. Zhang, "Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware," *Proc. IEEE Int'l Conf. Dependable Systems and Networks with FTCS and DCC (DSN '08)*, 2008.

[16] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis," *Proc. 14th ACM Conf. Computer and Comm. Security (CCS '07)*, 2007.

[17] B.D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *Proc. IEEE Symp. Security and Privacy (SP '08)*, 2008.

[18] S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and Implementing Malicious Hardware," *Proc. First USENIX Workshop Large-Scale Exploits and Emergent Threats (LEET '08)*, 2008.

[19] S. Embleton, S. Sparks, and C. Zou, "Smm Rootkits: A New Breed of OS Independent Malware," *Proc. Fourth Int'l Conf. Security and Privacy in Comm. Networks (SecureComm '08)*, 2008.

[20] "kvm: Coalescent Writes to MMIO," www.mail-archive.com/kvm@vger.kernel.org/msg00296.html, Oct. 2010.

[21] J.R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the Gap between Software and Hardware Techniques for I/O Virtualization," *Proc. USENIX Ann. Technical Conf. (ATC '08)*, June 2008.

[22] Y. Endo, Z. Wang, J.B. Chen, and M. Seltzer, "Using Latency to Evaluate Interactive System Performance," *Proc. Second USENIX Symp. Operating Systems Design and Implementation (OSDI '96)*, 1996.

[23] J.W. Palmer, "Web Site Usability, Design, and Performance Metrics," *Information Systems Research*, vol. 13, no. 2, pp. 151-167, 2002.

[24] "Vlogger at the Hacker's Choice," www.thc.org, Oct. 2010.

[25] "NLANR/DAST: Iperf—The TCP/UDP Bandwidth Measurement Tool," http://sourceforge.net/projects/iperf/, Oct. 2010.

[26] "Superscape 3D VGA Benchmark," www.bookcase.com/library/software/msdos.util.screen.vga.html, Oct. 2010.

[27] "SiSoftware Sandra—Windows System Analyser," www.sisoftware.co.uk, Oct. 2010.

[28] "Bonnie++: File System Benchmarks," www.coker.com.au/bonnie++, Oct. 2010.
[29] "AutoHotkey: Program with Hotkeys and AutoText," www.autohotkey.com, Oct. 2010.
[30] "Autokey: Text Replacement Tool for Linux," http://autokey.sourceforge.net, Oct. 2010.
[31] P. Ferrie, "Attacks on Virtual Machine Emulators," Dec. 2006.
[32] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility Is Not Transparency: VMM Detection Myths and Realities," *Proc. 11th USENIX Workshop Hot Topics in Operating Systems (HOTOS '07)*, 2007.
[33] "Kernel TRAP—KVM: Detect if VCPU Triple Faults," http://kerneltrap.org/mailarchive/git-commits-head/2008/4/27/1622284, Oct. 2010.

Duy Le received the BS degree from Hanoi University of Technology, and the MS degree from Francophone Institute of Computer Science, Hanoi. He is working toward the PhD degree in the Department of Computer Science, The College of William and Mary. His current research interests include computer system security, operating systems, virtual machine, energy efficiency, and anomaly detection. He is a student member of the IEEE.

Haining Wang received the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2003. He is an associate professor of computer science at the College of William and Mary, Williamsburg, Virginia. His research interests include area of networking systems, computer security, cloud computing, and wireless and mobile computing. He is a senior member of the IEEE and the IEEE Computer Society.