# Secure and Flexible Monitoring of Virtual Machines

Bryan D. Payne    Martim D. P. de A. Carbone    Wenke Lee
*Georgia Institute of Technology*
{bdpayne,mcarbone,wenke}@cc.gatech.edu

## Abstract

*The monitoring of virtual machines has many applications in areas such as security and systems management. A monitoring technique known as introspection has received significant discussion in the research literature, but these prior works have focused on the applications of introspection rather than how to properly build a monitoring architecture. In this paper we propose a set of requirements that should guide the development of virtual machine monitoring solutions. To illustrate the viability of these requirements, we describe the design of XenAccess, a monitoring library for operating systems running on Xen. XenAccess incorporates virtual memory introspection and virtual disk monitoring capabilities, allowing monitor applications to safely and efficiently access the memory state and disk activity of a target operating system. XenAccess' efficiency and functionality are illustrated through a series of performance tests and practical examples.*

## 1 Introduction

On the wave of its renewed popularity, researchers are identifying many new applications that leverage the abstractions and isolation provided by virtualization. One area that has received significant attention is security. Security applications benefit from virtualization by running in isolated virtual machines (VMs) and building smaller trusted computing bases (TCBs). This technique has been used in a variety of research projects [21, 9, 10, 18, 7, 15, 2].

Among the security research that uses virtualization, an increasingly common practice is to setup a secure VM that is used to monitor the other VMs running on the same virtual machine monitor (VMM)[1]. These monitors are used in intrusion detection systems (IDS), integrity checking, honeypot systems and forensic analysis, among others. We believe that this idea is sound, yet many previous efforts in this space have focused more on the applications of introspection than building a proper architecture to support the technique.

In this paper, we present the XenAccess monitoring library and our lessons learned from the implementation process. XenAccess provides virtual memory introspection and virtual disk monitoring capabilities. Our experience designing and implementing this library has shown that implementing introspection in a secure and efficient manner is non-trivial. However, our architecture demonstrates how one can achieve these goals without losing monitoring functionality. Monitoring with XenAccess requires no changes to the VMM or to the VM being monitored. In addition, no changes are required to the OS being monitored, so XenAccess is not restricted to monitoring open source OSes. While our current implementation focuses on monitoring XenoLinux the XenAccess Library can be extended to monitor any OS that runs on the VMM. XenAccess incurs a negligible performance penalty for typical monitor applications.

We designed the XenAccess architecture based on six high-level requirements. In a general sense, these requirements can be seen as typical good programming guidelines, or good security guidelines. For example, some of our requirements could be seen as specialized versions of Saltzer and Schroeder's classic security design principles [28]. This is intentional, as our goal was to leverage known design principles in order to build a robust monitoring architecture. With this in mind, we identify the following six requirements for monitoring VMs:

1. **No superfluous modifications to the VMM.** The VMM should remain as small and simple as possible since it is part of the TCB. If a VMM includes the necessary primitives to support the monitoring architecture, then it should not be modified. If a VMM lacks the necessary primitives, then the modifications made should be what is minimally required to support the monitoring architecture.

2. **No modifications to the VM or the target OS.** Modifications to the target OS (i.e., the OS being monitored), are problematic. The target OS can tamper with

[1]Throughout this paper, the term VMM refers to the virtual machine monitor on top of which VM's run, whereas the term *monitor* refers to the actual VM monitoring applications that we focus on in this work.

this code, and changes to the target OS may require access to the target OS source code, which is not always available. One of the key reasons why virtualization is attractive for monitoring is the isolation between VMs. Placing monitoring code within the same OS that is being monitored bypasses this isolation, negating this key benefit. Therefore, this requirement encourages all monitoring code to remain in an isolated VM unless such a restriction makes it impossible for a monitor to gather the necessary information.

3. **Small performance impact.** An excessive performance impact can render a monitoring architecture worthless. This requirement ensures that the monitoring architecture does not prevent the target OS from performing its intended functions. The performance impact is measured as any reduction in performance of an application caused by the monitoring software. Ideally this impact is both small and consistent, but some initialization costs may be required.

4. **Rapid development of new monitors.** New monitors may be needed to address new types of attacks. Furthermore, it is advantageous to keep the monitor code simple to limit the opportunity for introducing errors into the monitors. The monitoring architecture should provide APIs that are used to develop new monitors. Therefore, satisfaction of this requirement means that the APIs should be designed in a way that simplifies the job of the monitor developer.

5. **Ability to monitor any data on target OS.** Monitors should have a full view into the target OS. The monitoring architecture should not be limited to providing information about a small part of the target OS. For example, an ideal memory monitor should be able to view all memory on the target OS. Likewise, an ideal disk monitor should be able to view all data going to and from the disk device. While this ideal may not always be possible, the more information a monitor can view, the harder it is for an attacker to evade detection.

6. **Target OS cannot tamper with monitors.** If the target OS can tamper with the monitors, then the possibility exists for malicious code to tamper with the monitors. For this reason, all of the monitors should be isolated or protected from the target OS. This is related to requirement (2), above. However, here we require that all monitor code, regardless of its location, be protected from attack. If all monitor code is in an isolated VM, then this is not difficult. If some monitor code must be placed outside of the TCB, then additional measures must be taken to protect that code. The extent of these measures will depend on the nature of the code being protected.

Our main contribution is the XenAccess monitoring architecture that satisfies the above requirements. It is important to emphasize that this paper addresses *an architecture for security* and not actual security techniques (such as IDS algorithms). These are topics for future papers. The remainder of this paper focuses on the XenAccess architecture, its implementation, and some example applications that demonstrate the performance and flexibility of XenAccess. Section 2 discusses the related work. Section 3 provides background information on the components in Xen used to build XenAccess. Section 4 presents the architecture and implementation details for XenAccess. Section 5 shows the results of our performance testing along with some example applications. Section 6 discusses future directions in this research space and we conclude with Section 7.

## 2 Related Work

VMMs first came into use over 35 years ago [11]. While Madnick and Donovan identified the security benefits of VMMs in the early 70s [20], research that explicitly leveraged these benefits did not take place until nearly 20 years later [17, 16]. More recently, virtualization is being used in different ways to address a variety of systems management, and security problems. In the security space, we have seen innovative work in intrusion detection systems [10, 18, 15], workload isolation [21, 9], attack investigation and debugging [7], and system monitoring [13, 2]. Each of these applications have one thing in common: they each require the ability to monitor data from a target OS. However, the mechanics of how to properly do such monitoring have not been adequately addressed in the literature. Through the details provided in this paper, and by making XenAccess an open source project, we are exposing these mechanics for the benefit of future work in this space.

The technique of virtual memory introspection was introduced by Garfinkel and Rosenblum [10]. While this work laid out how introspection could be used to build an intrusion detection system, the underlying mechanics of introspection were not discussed. Joshi et al presented a system called IntroVirt [15] that uses introspection and replay to test if a system was previously attacked through a known vulnerability. Similar to the first effort, only limited details were given regarding the introspection mechanism. More recently, several projects have provided details about their introspection techniques, only to reveal suboptimal security decisions in their architecture. The Hyperspector project [18] is a virtual distributed monitoring environment used for intrusion detection. The Hyperspector approach to introspection is to provide access to a few specific pieces of information (processes, sockets, etc). This limited view into the target OS violates property (5) of our requirements for a robust monitoring solution, and Hyperspector also violates property (1) by extensively modifying the VMM, and (6) by sharing OS kernels between VMs. Asrigo et al presented a system for monitoring honeypots [2], but they violate property (2) by requiring hooks in the target OS kernel, property (3) by causing a substantial performance impact, and property (4) by incorporating kernel code in new monitor hooks. Finally, the Antfarm system [13] tracks only OS-level processes, violating property (5), and performs the monitoring from within the VMM, violating property (1). Each of these virtual memory introspection systems were built to provide monitoring capabilities for a security system. However, none of these systems meet our six requirements for a monitoring solution, making it much more likely for an intruder to compromise, evade or disable the monitors.

Monitoring in a virtualized environment is not the only approach. Petroni et al developed Copilot [23], a secure coprocessor used to monitor the memory of a host. In practice, this approach is very similar to virtual memory introspection from a VM, but it requires extra hardware and cannot be generalized to monitoring other data such as disk I/O. Looking into the commercial world, many monitoring applications sold today simply run within the target OS. For example, anti-virus software typically runs in the same OS that it is protecting. However, this architecture is flawed because malicious software can simply disable the anti-virus software [3].

Monitoring at the disk level has traditionally taken place as part of a research trend focused on creating smarter, more semantically-aware devices. This has applications in both systems optimization and security. Sivathanu et al has shown how smart disks can employ gray-boxing techniques [1] to infer the semantics of the underlying filesystem and use this knowledge to enable various performance improvements and features like secure file deletion [29]. Researchers at Carnegie-Mellon University have leveraged the physical isolation of such systems to enable intrusion detection [22, 12] and recovery capabilities [30]. These systems are able to perform their functions in a tamper-resistant manner, regardless of an OS compromise. This approach, however, has the obvious downside of requiring additional hardware support and the need for a special infrastructure for communication between the management tools inside the OS and the disk IDS. XenAccess leverages virtualization to provide the same level of monitoring functionality without either of these limitations.

More recently, disk monitoring has started to receive attention in the context of virtual machines. HyperSpector's approach is to mount a shadow version of the monitored filesystem and execute integrity checkers (e.g., `tripwire`). Not only does this require significant modifications to the VMM, violating property (1) and increasing the chances of a VMM compromise; it also limits access to the disk data by providing an exclusively static and high-level view of it, violating property (5) and making it very easy to evade the monitor. Elango et al [8] and Jones et al [14] have applied some of the principles of semantically-smart disk systems and gray-boxing [29, 1] to the performance improvement of Xen virtual machines. Their results show how monitoring and active control of virtual machines can have a wide variety of applications outside the security area.

XenAccess is designed to work with Xen [4], but the ideas of virtual memory introspection and disk monitoring are not unique to Xen. Our architecture could be ported to any of today's virtualization solutions. In the past, many researchers choose to work with User Mode Linux (UML) [6], a virtualization solution that allows you to boot a Linux kernel as a process in a running version of Linux. The earliest work with introspection used VMWare [31], a full featured commercial virtualization product. Looking forward, interest is now growing in the kernel-based virtualization driver (KVM) [24] that is built into the Linux kernel starting with version 2.6.20. While our techniques are viable on any of these platforms, a virtualization solution designed as an independent and lightweight software layer running directly on the hardware, such as Xen, offers a solid foundation to a security-oriented solution.

## 3 Xen Hypervisor Background

The XenAccess monitoring library is based on Xen [4], a popular open-source virtual machine monitor (VMM). This section gives an overview of Xen's architecture, followed by a discussion of its memory management and block device I/O subsystems. The discussion of these subsystems is central to the understanding of XenAccess' monitoring components.

### 3.1 Overview

Xen has traditionally used a paravirtualized approach to implement virtualization. This technique consists of altering the guest OSes by replacing sensitive instructions that cannot be virtualized with special hypercalls, that is, calls that are made directly to the VMM. This approach has the advantage of providing good performance, since no trapping is done, and also allowing virtual machines to run on top of non-virtualizable architectures (such as x86) [25]. Nevertheless, one drawback of paravirtualization is that the guest OSes must be modified. Recent versions of Xen have the capability to run unmodified OSes by using the new Intel VT-x and AMD-V technologies. XenAccess uses paravirtualized domains without violating property (2) because the changes required for paravirtualization are not strictly part of the XenAccess architecture and do not make it any easier for the target OS to tamper with the monitoring code.

Xen uses a split domain architecture, meaning that reg-

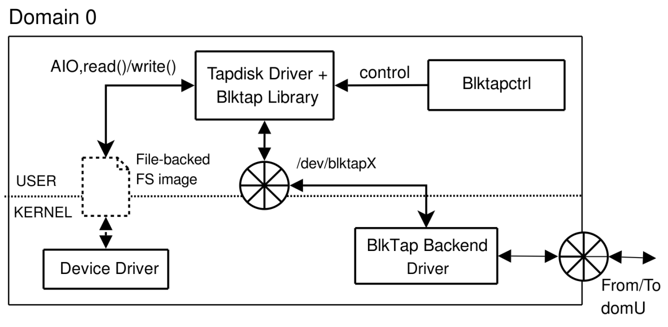


**Figure 1:** Blktap disk I/O architecture.

ular guest OSes are kept in unprivileged domains (domU), whereas a single administrative domain exists as Domain 0 (dom0). Dom0 can be seen as a domain-level extension of Xen in which all of the management functionalities are located. It has complete access rights to all virtual machines being run and also works as a device driver proxy for domU's virtual devices.

The VMM itself is a simple and thin software layer whose main job is to guarantee proper isolation between virtual machines, performing minimal resource management. This isolation is quite robust, since Xen relies directly on hardware-level protection mechanisms and has a much narrower interface than a standard operation system (e.g., Linux).

### 3.2 Memory Management

One of the key tasks for a VMM is to partition the system memory between each VM. Xen achieves this using three levels of memory: machine, physical, and virtual addresses. The machine addresses are the actual addresses used by the hardware and managed by the VMM. The physical addresses are what each paravirtualized OS uses. This first abstraction allows the VMM to assign non-contiguous memory regions to a paravirtualized VM. As far as the VM is concerned, the physical addresses are the addresses used by the hardware. In reality, these addresses are translated by a lookup table in the VMM into machine addresses. The third type of address is a virtual address, which is used the same way in the paravirtualized OS as traditionally used in OSes.

Both machine and physical addresses are often referred to in terms of a machine frame number (MFN) and a physical frame number (PFN). These numbers refer to a single page of memory, which are 4k bytes each[2]. A complete address is given as both an MFN or PFN and an offset into that page of memory. Using this scheme, Xen provides tables to convert from MFN to PFN and PFN to MFN. These tables are called M2P and P2M, respectively. Similarly, the running OSes use a page table (PT) to convert between virtual addresses and machine addresses. Xen protects these PTs in order to ensure the memory isolation between VMs. A paravirtualized OS must invoke a hypercall to modify its PT.

[2]Here we refer to the x86 architecture where memory pages are usually 4k, but can also be 4M.

### 3.3 Device I/O

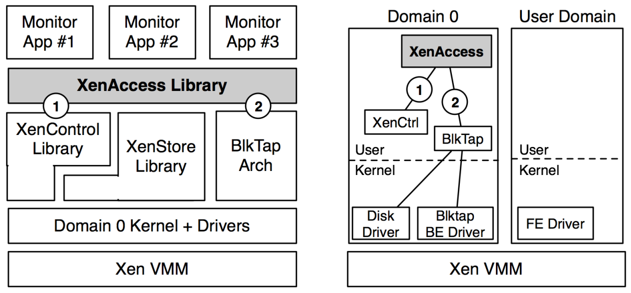
Xen's device I/O architecture is based on a split driver model. In this model, there is a frontend driver inside domU's kernel that communicates with a backend driver inside dom0's kernel. Inter-domain communication relies on shared asynchronous I/O rings, shared memory pages and a control framework called XenBus. This architecture is currently used for block and network devices.

For block devices, system calls are issued by domU applications, which are translated into block-level operation requests by the kernel. Traditionally, the backend driver in dom0's kernel receives the request from the frontend, and sends them directly to the disk. Xen 3.0.3 introduced a new architecture for block device I/O, illustrated in Figure 1, with some interesting new properties. A block tap was introduced, allowing disk drivers to be implemented as userspace applications. These userspace drivers tap into the backend driver (blktap) and can directly manage disk activity with relatively small performance costs [32]. This architectural change added substantial flexibility to disk driver development, greatly simplifying it and at the same time allowing more powerful functionalities to be implemented. The fact that tapdisk drivers are regular userspace applications allows virtual disk I/O to be implemented with simple file-manipulation system calls and/or library calls.

## 4 XenAccess Monitoring Library

### 4.1 Architecture

The primary goal for the XenAccess architecture is to satisfy the six requirements stated in the Introduction. We chose Xen as a virtualization solution because it is a Type I VMM; it runs directly on the hardware, allowing for a solid foundation to the TCB. It also already includes an infrastructure suitable to satisfy our monitoring needs, so that changes to the VMM are unnecessary (property (1)). Likewise, by building on top of Xen's infrastructure, we were able to design the monitoring architecture to work without changes to the target OS, allowing us to satisfy property (2). To prevent the target OS from tampering with the monitors and satisfy property (6), we place the monitors in a different VM than the target OS. Xen provides sufficient isolation between VMs for this to be a viable solution. And, for



(1) XenAccess maps memory pages using memory access functions from libxc (2) XenAccess intercepts disk I/O through the blktap capability, which is built into Xen

**Figure 2:** The XenAccess architecture leverages existing capabilities in Xen to reduce complexity and improve overall performance.

environments that require more explicit isolation, mandatory access control is built into the Xen VMM [27]. Finally, we desire an architecture that can monitor any data on the target OS in order to satisfy property (5). XenAccess currently provides monitoring capabilities for both memory and disk I/O. However, the architecture can easily be extended to monitor additional information such as network traffic, CPU context, and static disk contents. We examine XenAccess' adherence to properties (3) and (4) in Section 5.

Figure 2 shows the overall software architecture from two perspectives. On the right, we show the location of the critical components in relation to the VMM and VMs. Currently, XenAccess runs in Domain 0 as this simplifies access to the Blktap Architecture and the XenControl Library. XenAccess could also run in a user domain after sufficient privileges are given for that domain to perform the monitoring. The left side of Figure 2 shows how XenAccess fits into the Xen software stack. Here we emphasize that XenAccess is a library intended for use by monitor applications. Some simple example applications are discussed in Section 5.2.

Virtual memory introspection requires accessing the memory of one VM from another. Xen provides a function in the XenControl Library that is used for this purpose and this functionality could be added to other VMMs using a small amount of additional code. In Xen, the function `xc_map_foreign_range()`, maps the memory from one VM into another. After the memory is mapped, it can be treated as local memory, providing for fast monitoring capabilities. In order to convert a XenAccess API call into a call to `xc_map_foreign_range()`, XenAccess must perform several memory address translations. This requires additional information about the target OS which can be obtained from the XenStore, a database of information about each VM, and interpreted using some knowledge of the target operating system's implementation. The steps needed to convert a kernel symbol or virtual address into a memory mapped page are discussed in Section 4.2.1.

Whereas virtual memory introspection monitors the current state of memory pages, the virtual disk monitoring captures data traveling to and from the disk. This data is captured by placing code that directly intercepts the data path between the target OS and the hard drive it uses for data storage. We chose the Blktap Architecture for this capability due to its good performance [32] and its ability to deliver low-level information to user-space software such as the XenAccess Library.

The XenAccess architecture utilizes functionality included with Xen in order to reduce the implementation overhead and adhere to property (1). At this point it is important to emphasize that while we acknowledge that XenAccess' functionality and its adherence to the principles established in the Introduction are significantly based on the infrastructure already provided by Xen, the XenAccess architecture and the principles supporting it could be implemented on other VMMs as well. The core functionality needed in the VMM includes mapping memory between virtual machines, viewing VM-specific metadata (e.g., running kernel version), and tapping into the data between a device (e.g., the hard disk drive) and the associated device driver. This functionality could be added to any modern virtualization environment, if it is not already there, allowing for support of the XenAccess monitoring architecture. The only caution that must be taken is to implement at the VMM level only that which is strictly necessary, and do it very carefully, so as to minimize the probability of introducing bugs in the TCB. All the remaining functionality should be implemented in a special security or management domain (such as Xen's dom0) taking the appropriate performance considerations.

## 4.2 Implementation

XenAccess is implemented in C as a shared library with 1935 source lines of code (SLOC). XenAccess makes use of `libxc`, `libxenstore`, and the Blktap Architecture. The current version is built to monitor a paravirtualized version of Linux 2.6.16 running on Xen 3.0.4_1, however the techniques here can be extended to work with other OSes. The two primary monitoring functionalities in XenAccess are virtual memory introspection and virtual disk monitoring. Implementation details for each of these techniques are discussed in the sections below.

### 4.2.1 Virtual Memory Introspection

XenAccess uses the `xc_map_foreign_range()` function, provided through the XenControl Library (`libxc`), to view the memory of another VM. Using this function eliminates the need to modify the VMM or the target OS, satisfying properties (1) and (2). This function can be used to map a memory page from the target OS using its MFN. XenAccess uses this function for raw memory access and then builds up from there using address translation tables in the VMM and the target OS. For example, to convert a PFN to a MFN, XenAccess uses lookup tables that are provided by Xen. Similarly, to convert a virtual address to a MFN, XenAccess uses the PTs.

## Memory Introspection API

- **`xa_init()`**: Initializes access to a specific domU given a domain ID. This function takes a domain ID and returns a structure that holds cached information related to accessing that domain. All calls to `xa_init()` must eventually call `xa_destroy()`.
- **`xa_destroy()`**: Destroys an instance by freeing memory and closing any open handles.
- **`xa_access_kernel_symbol()`**: Memory maps one page from domU to a local address range. The memory to be mapped is specified with a kernel symbol (e.g., from `System.map`). This memory must be unmapped manually with `munmap()`.
- **`xa_access_virtual_address()`**: Memory maps one page from domU to a local address range. The memory to be mapped is specified with a kernel virtual address. This memory must be unmapped manually with `munmap()`.
- **`xa_access_user_virtual_address()`**: Memory maps one page from domU to a local address range. The memory to be mapped is specified with a virtual address inside a process address range. This function also requires a process ID. This memory must be unmapped manually with `munmap()`.

We provide an overview of the implementation of each of these functions.

All users of the introspection library must begin with a call to `xa_init()`. This function initializes the `xa_instance struct` which holds information that is used throughout the introspection process. Any work that can be done "up front" and cached is held in this structure. This includes locating the address of the kernel page directory, initializing a handle to `libxc`, initializing a pointer to a PFN to MFN lookup table, determining if the domain is paravirtualized or fully virtualized, and more. Once a user is done with the library, a call should be made to `xa_destroy()` to free any memory associated with the `xa_instance struct`.

After initializing the `xa_instance struct`, one can use any of the three access functions listed above. Starting with the simplest, the `xa_access_virtual_address()` takes a kernel virtual address and returns a pointer to the memory page holding that address along with the offset to the specified address within the memory page. This address translation requires a PT lookup, which requires XenAccess to load three memory pages. First, the page directory is loaded to find the location of the PT. Next, the PT is loaded to find the location of the address. Finally, the memory page holding the address is loaded and this page, along with an offset to the address, is returned to the user. Returning a shared memory page contributes to the good inter-VM memory copy performance shown in Section 5.1, which is a requirement for property (3).

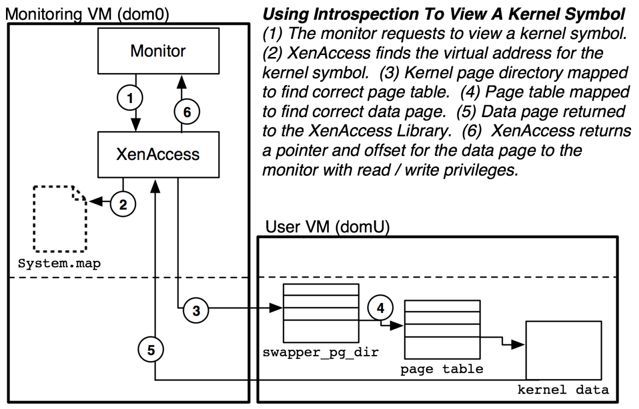

Monitoring VM (dom0)
Monitor
XenAccess
System.map
User VM (domU)
swapper_pg_dir
page table
kernel data
1
2
3
4
5
6
Using Introspection To View A Kernel Symbol
(1) The monitor requests to view a kernel symbol. (2) XenAccess finds the virtual address for the kernel symbol. (3) Kernel page directory mapped to find correct page table. (4) Page table mapped to find correct data page. (5) Data page returned to the XenAccess Library. (6) XenAccess returns a pointer and offset for the data page to the monitor with read / write privileges.

**Figure 3:** The steps needed to map a kernel memory page based on a kernel symbol using virtual memory introspection.

The `xa_access_kernel_symbol()` function, shown in Figure 3, requires one extra step beyond the virtual address translation described above. This step is to convert a kernel symbol to a virtual address. XenAccess performs this conversion using the `System.map` file associated with the kernel from domU. If this file is not available, then this function will fail. The `System.map` file is essentially a large table of symbols and addresses. XenAccess scans this file until it finds the symbol provided. It then proceeds with a virtual address access using the address associated with the kernel symbol. Since this operation requires performing a lookup from a file on disk, it is considerably slower than the `xa_access_virtual_address()` function, but the results are cached so the average case is fast as discussed in Section 5.1. Further performance improvements could be achieved by memory mapping the file, moving the costly file read operations into the `xa_init` function. However, most monitoring applications will repeatedly view the same memory location, using the cached information.

The final function in the virtual memory introspection API is `xa_access_user_virtual_address()`. This function provides access to user space memory. Page table lookups for a virtual address in user space are essentially the same as kernel space. The main difference is that we must lookup the location of the page directory associated with the process. Recall that for kernel space, the location of the page directory is cached during library initialization, but the page directory locations for each process can change as processes come and go. To lookup the page directory for a process, XenAccess scans the kernel task list looking for a process with the given process ID. Upon finding a match, the page directory can be obtained from the `task_struct` in kernel memory. Using this page directory, the remainder of the virtual address translation is the same as previously described for the kernel.

**HVM Support** XenAccess has preliminary support to perform memory introspection on fully virtualized (HVM) VMs. In HVM VMs, physical addresses and machine addresses are the same. Therefore, XenAccess will automatically detect HVM domains and not attempt to perform this translation in those cases. In practice, the P2M translation is a simple table lookup, so omitting this step does not measurably improve performance. Since memory introspection support for HVM VMs is in its early stages, there is some reduced functionality. This reduced functionality is the reason why there is no HVM performance data available for the user address function in Figure 5.

**Improving Performance** Since XenAccess must use memory from the target OS and the VMM to perform address translations, these operations can be costly. Therefore, XenAccess uses a least recently used (LRU) cache to store the results of the address translations. This is similar to a translation lookaside buffer (TLB). However, in the case of XenAccess, we also cache kernel symbol names since disk access is always a slow operation. This caching is critical to achieving acceptable performance and satisfying property (3), as discussed in Section 5.1.

**Use of OS-specific Information** A virtual memory address can be converted to a MFN without any knowledge of the OS in domU. This is because the address conversion is specific to the processor architecture and not to the OS. A PT lookup, which is required to perform this address conversion, starts by obtaining the address of the page directory. This information is stored in one of the control registers, `CR3`, of the domU CPU context. Starting with the page directory, one can complete a PT lookup and, therefore, find the MFN associated with any virtual address on a host. However, it can be difficult to determine what virtual address to access.

Identifying virtual addresses that are interesting requires some knowledge about the OS. One artifact of compiling a Linux kernel is the `System.map` file. This file is a listing of symbols exported from the kernel along with the virtual address of each symbol. Using this file, combined with the ability to access arbitrary virtual addresses, one can view and modify data such as the system call table, interrupt descriptor table, Linux kernel module (LKM) list, task list, and more. In Microsoft Windows, exported symbols are available in debugging libraries and in `ntdll.dll`. Of course, making use of these data structures requires knowledge of the data layout inside each structure. In Linux, this is determined by inspecting the source code and using technical references such as the kernel books by Bovet and Cesati [5] or Love [19]. In Windows, much of this information is available in technical references as well [26].

The memory introspection implementation in XenAccess provides logical separations between the OS-specific code and the general code that is OS-neutral. This is done to permit rapid integration of new target OSes. For example, the current code is only designed to monitor a Linux 2.6 OS, but extending XenAccess to monitor Windows or FreeBSD would only require adding the specific knowledge for each OS.

### 4.2.2 Virtual Disk Monitoring

XenAccess introspects into low-level disk traffic, just as it is able to map raw memory pages. It therefore satisfies property (5) by providing full and complete access to data. XenAccess also includes an inference engine which is able to dynamically infer the high-level filesystem operations executed inside a domain based on the intercepted low-level disk traffic. To this end, we have decided to leverage the Blktap architecture described earlier since it simplifies the implementation of the interception mechanism and avoids making modifications to the VMM, which is encouraged by property (1). The biggest challenge, however, is faced by the inference engine which must somehow overcome the semantic gap between the low-level view and the desired higher-level, filesystem-oriented view that will be given as output. It does this combining pre-programmed filesystem structure knowledge with dynamic inference techniques.

Whereas the interception mechanism (which is roughly equivalent to the introspection memory-mapping) is independent of the current OS and filesystem by only providing raw access to disk traffic, the inference engine is dependant on knowledge of the filesystem in use. So far, knowledge has been included in the inference engine to be able to determine only file/directory creation/removal operations under the ext2 filesystem, although knowledge about other filesystems can be incorporated.

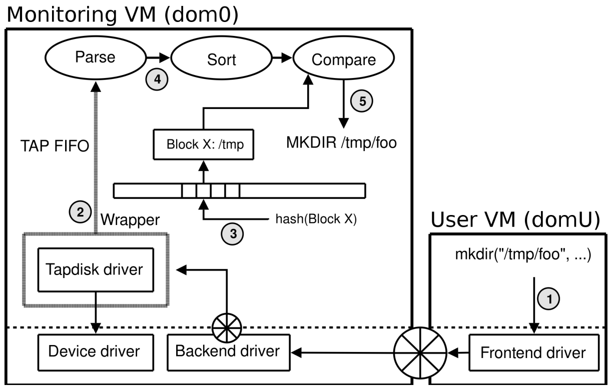


**Figure 4:** The XenAccess disk monitoring functionality. (1) A `mkdir()` system call is executed by an application inside the user VM. This call is translated into low-level operations by the kernel and sent to the backend driver at the monitoring VM. (2) The block data flow is intercepted by the wrapper driver before it is processed by the tapdisk driver and sent through the tap FIFO. (3) Blocks are read from the FIFO and hashed into the table. (4) If a record is found for a block X (which holds part of `/tmp` contents), it is parsed and sorted. If not, it is discarded. (5) The new and old version of block X are compared, its differences are translated into a directory creation.

## Disk Monitoring API

- **`xadisk_init()`**: Opens the FS image file and initializes the library's main data structures;
- **`xadisk_destroy()`**: Closes the image file and deallocates all global structures;
- **`xadisk_set_watch()`**: Sets a watch-point in one of the filesystem's directories;
- **`xadisk_unset_watch()`**: Removes the watch associated with a directory;
- **`xadisk_activate()`**: Creates a thread that runs the main monitoring and inference engine;
- **`xadisk_deactivate()`**: Finishes the monitoring thread associated with a monitoring instance.

**Wrapper drivers** As illustrated in Figure 4, wrapper drivers intercept the disk data flow before it is processed by the tapdisk driver, sending all the data received to the inference engine. Their functionality is simple: disk block data and metadata received from kernel space are marshalled into a buffer and sent to the inference engine through a FIFO. This architecture allows the inference routines to execute asynchronously with the actual disk reads and writes, which reduces performance impact.

**Initialization and Watchpoints** The disk monitoring library is initialized by the `xadisk_init()` function. It creates and initializes all relevant data structures, and opens all FIFO and file descriptors that will be used. It also reads and parses important filesystem metadata contained in the filesystems's superblock and the group descriptors. It gathers all the essential information to bootstrap the monitoring instance, allowing watchpoints to be set and the inference engine to start. This information is stored in a `xadisk_t struct`, which is returned by the function. The `xadisk_destroy()` function ends a monitoring instance, closing the image file descriptor and deallocating all associated data structures.

Watchpoints are set by the `xadisk_set_watch()` function. It receives as an argument the full path of a directory that is to be monitored for file/directory creation/removal. This function initially performs a recursive search through the filesystems's structures (inodes and directory entries) with the goal of finding the disk block(s) which store the directory's content. The corresponding blocks are then read from disk, parsed, and have their entries sorted. Finally, they are inserted into a global hash table. Watchpoints are erased by removing the corresponding block records from the hash table. This is done by the `xadisk_unset_watch()` function.

**Inference Engine** The inference engine constitutes the core of the virtual disk monitoring library and its operation is illustrated in Figure 4. It is activated by the API function `xadisk_activate()`, which subsequently creates a new thread running the engine itself. At each iteration, the engine reads a new record from the tap FIFO, sent by the wrapper driver. Next, the record's block number is hashed into the global hash table and the existence of an older version is checked. If it is found, file/directory creations/removals are inferred by determining the differences between the old and new versions of the block. The name and type of the object created or removed can be determined by looking at the type field in the corresponding entry. Results are written to a per-application FIFO that is accessible by a program using XenAccess. The function `xadisk_deactivate()` stops the inference engine by killing the thread executing it and closing all associated descriptors.

**Limitations** The current implementation has some limitations. Although it has access to all data sent to the disk, for now the inference engine is only able to infer file/directory creation/deletion. It is certainly possible, nevertheless, to implement the inference of more intricate operations like file read/write, object renaming and file truncating. More elaborate algorithms will be required, but the same monitoring architecture can be used.

Since we directly rely on the Blktap Architecture which is exclusive to paravirtualized environments, disk monitoring of fully-virtualized guests (and therefore, OSes other than Linux) is not supported for now. However, we feel that implementing this support would not require major effort, since the inference engine would not require any structural changes. It would be a matter of finding out at which point of the disk data flow a tap must be introduced and a new wrapper driver implemented. And of course, the inclusion of specific knowledge of the filesystem being used is also necessary.

Similarly, although the current implementation of the inference engine is dependant on detailed knowledge of the ext2 filesystem, its architecture could be used without changes to monitor other filesystems that adhere to the same general design principles as ext2 (such as the use of directory structures). Besides, we were careful enough throughout its design and implementation to compartmentalize almost all filesystem-specific implementation, making it easy for the integration of new ones.

## 5 Experimental Results

This section focuses on the performance and qualitative evaluation of our prototype. We evaluate the performance of each technique separately through a series of micro-benchmarks. This is followed by example applications of each monitoring technique, showing their usefulness in monitoring an OS's internal structures and disk activity.

### 5.1 Performance Results

The performance figures show that XenAccess introduces minimal overhead. We performed the testing on Xen 3.0.4_1 running Fedora Core 6 in both dom0 and domU. This software was run on a 2.33 GHz Intel Core Duo processor with 2 MB L2 cache, 2 GB RAM, and an 80 GB 7200 RPM disk. Dom0 was assigned 2 processor cores and domU was assigned one processor core.

#### 5.1.1 Virtual Memory Introspection

Each of the performance measurements shown in this section were done using the `gettimeofday()` function, which has a micro-second granularity. Times were measured by recording the time immediately before and after the function being measured. The difference between the two times was recorded. This measurement was repeated for 1000 times for each test. We choose 1000 measurements because this was sufficient to minimize the standard deviation for a given set of measurements under this setup. Additional measurements did not improve the precision.

The data in Figure 5 show the average time to complete the specified function call. The cache hit columns represent the results with the LRU cache enabled. The cache miss columns represent the results with LRU cache disabled. The simplest case is shown on the left of this graph. The `xa_access_virtual_address()` function must map three memory pages on a cache miss and one on a cache hit. This difference explains the improvement seen with the LRU cache. The time for `xa_access_kernel_symbol()` is dominated by the operation to lookup the kernel symbol. This operation is a lookup inside a file on disk, which is costly. With a cache hit, the symbol to machine address mapping is stored in the cache, making the performance similar to `xa_access_virtual_address()`. The last access function is `xa_access_user_virtual_address()`. This function must traverse the task list in the domU kernel to locate the page directory for the process virtual address. This explains the slower performance for the cache miss. On a cache hit, this traversal is not needed, performance is essentially the same as `xa_access_virtual_address()`.

After the memory is accessed, the next step is to read from or write to that memory. As seen in Figure 6, this operation is fast compared to mapping the memory. These performance results show the time required to `memcpy()` data

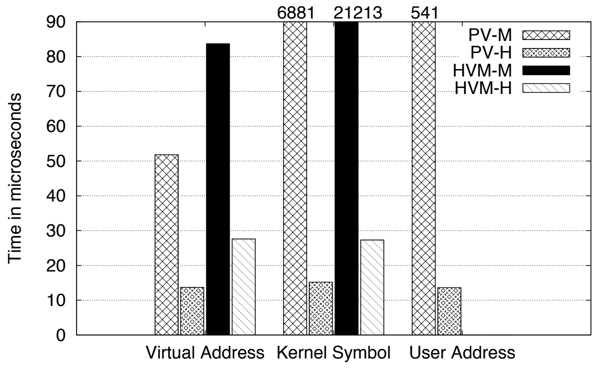


**Figure 5:** Performance of the three memory access functions in XenAccess for a paravirtualized target domain.

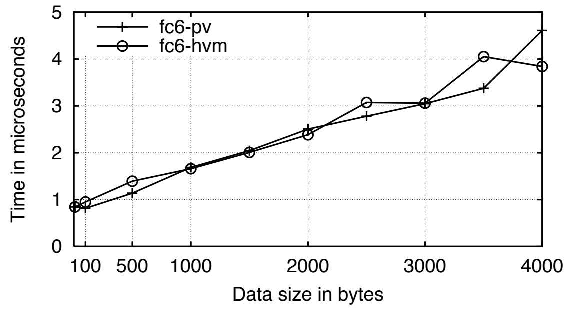


**Figure 6:** Time for monitor to read memory through introspection.

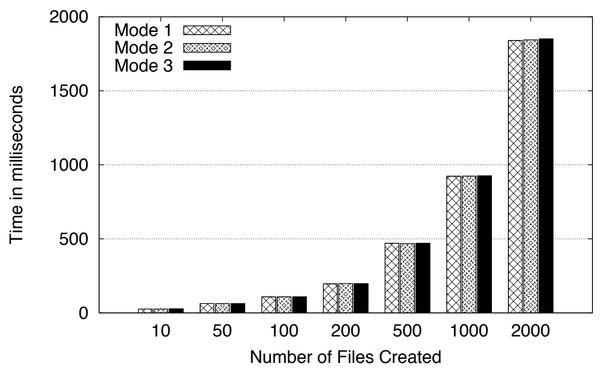


**Figure 7:** Performance of three different scenarios running a disk benchmark. Mode 1: Disk monitoring enabled and watchpoints set on each benchmarking directory; Mode 2: Disk monitoring enabled and no watchpoints set on the benchmarking directories; Mode 3: Standard `block-aio` tapdisk driver being used (no monitoring).

from kernel memory in the target OS. In general, we found that data is copied into a data monitor at a rate of approximately 1kB / $\mu$sec. Figure 6 shows that `memcpy()` performance for PV and HVM VMs is essentially the same. The variance in these measurements can be attributed to experimental noise given the precision of our timing mechanism and the small measurement times. Looking at the cache hit values in Figure 5 and the memory copy performance, the memory introspection capabilities in XenAccess perform well enough to have a negligible impact on the overall system performance, satisfying property (3) from the Introduction.

#### 5.1.2 Virtual Disk Monitoring

XenAccess' disk monitoring performance was evaluated through the execution of a benchmarking shell script. This script tests exhaustively one of the operations monitored by XenAccess: file creation. It works by measuring the time it takes to create a variable number of files inside ten different directories. The total number of files created is equally distributed throughout the ten directories. The timing measurements include both the execution of the file creation commands and the manual flushing of the changes to the disk (through the `sync` command). Manual flushing was necessary so that the actual performance impact would not be hidden by the operating system's buffer cache.

The script was executed in three different modes and the results were compared. A sample size of 50 was used for each mode, which was enough to minimize the standard deviation. Measurements were made using the Linux `time` command, which has millisecond precision. A description of the modes is included in Figure 7 caption.

Figure 7 shows that the difference between the different timing measurements for each test case is negligible. Even in the case where 2000 files were created, the differences between the measurements were not statistically significant. The conclusion is that the performance overhead added by XenAccess' disk monitoring capabilities are minimal, and therefore obeys property (3) as defined in the Introduction.

The explanation for this negligible overhead lies in the design of the Blktap architecture, as well as XenAccess' own design. Most of XenAccess' disk monitoring engine code is executed asynchronously with regard to the actual disk I/O. The wrapper driver is basically the only extra code added by XenAccess to the disk I/O critical path. It does a simple data marshaling followed by a memory copy operation to the tap FIFO. These are not expensive operations. The asynchronism created by the use of a FIFO allows the inference engine, which is the most performance-intensive component of the architecture, to execute in parallel with the actual disk operations. In addition, we ran our benchmarks on a dual-core platform, enabling real parallelization of the tasks.

### 5.2 Example Applications

XenAccess is straight forward to use, allowing for rapid development of new monitors to satisfy property (4) from the Introduction. In this section we show several example applications to demonstrate our monitoring capabilities. While reading through these examples, keep in mind that we are deliberately showing simple use cases as an introduction to the library. However, XenAccess provides complete read/write access to a VM's memory space and has complete access to the disk I/O. With this level of power, the potential applications are only limited by the user's imagination.

#### 5.2.1 Virtual Memory Introspection

Using introspection, XenAccess can view and modify data in memory of a running OS. The example below shows how to use XenAccess to view the LKMs. Additional examples in the open source release show how to list running processes and view memory pages of a particular process on the target OS. These examples utilize information directly from a running Linux kernel.

**List Linux Kernel Modules** This example uses the `xa_access_kernel_symbol()` function to list the LKMs installed into the domU kernel and is 44 SLOC. Program 5.1 shows the code for this example. The code follows a linked list in the domU kernel memory using introspection. It starts by loading the memory page containing the head of the list, which is found using the `modules` kernel symbol. This address points to a `module struct`. This structure contains a circular doubly linked list that points to the rest of the modules. Therefore, the code proceeds by loading the memory page addressed by the next pointer all the way down the list. For each structure, the module name is accessed by creating a pointer to its offset, and then it is printed to `stdout`. Since the linked list is circular, the code ends when it finds a pointer back to the head of the list.

**Program 5.1** Source code for an example that lists all running LKMs in the domU kernel. All error checking code has been removed for clarity.

```
xa_init(dom, &xai);
memory = xa_access_kernel_symbol(&xai,
    "modules", &offset);
memcpy(&next_module, memory + offset, 4);
list_head = next_module;
munmap(memory, XA_PAGE_SIZE);
while (1){
    memory = xa_access_virtual_address(&xai,
        next_module, &offset);
    memcpy(&next_module, memory + offset, 4);
    if (list_head == next_module){
        break;
    }
    name = (char *) (memory + offset + 8);
    printf("%s\n", name);
    munmap(memory, XA_PAGE_SIZE);
}
xa_destroy(&xai);
if (memory) munmap(memory, XA_PAGE_SIZE);
```

Since this example is accessing and displaying OS-specific information, it requires OS-specific knowledge. In this case, the knowledge falls into two categories. First, we must know that the `modules` symbol points to the beginning of a linked list that will provide the information that we need. Second, we must know the offsets within the `module struct` needed to access information such as the `next` pointer and the module name. Requiring this type of information is common for introspection applications. For this example, the information needed was available in both the Linux source code, and Bovet and Cesati's kernel book [5].

**Additional Examples** The example above is straightforward and provides a quick understanding of XenAccess's introspection capabilities in operation. Other monitors are not much more complex. For example, we developed an application that monitors for changes in the system call table (110 SLOC) and an application that monitors the integrity of an installed LKM (172 SLOC). The security applications of these types of monitoring are clear in areas like intrusion detection and integrity checking, and have been well explored in literature. XenAccess makes these types of applications possible by providing memory access at the proper levels of abstraction. Compared to other virtualization monitoring architectures, such as the work by Asrigo et al [2], XenAccess allows for rapid monitor development since monitors are small user-space applications rather than kernel hooks. Based on our experience building XenAccess introspection monitors, we feel that our architecture satisfies property (4).

### 5.2.2 Virtual Disk Monitoring

In this example, three file/directory creation/deletion commands are executed inside domU followed by the `sync` command, which flushes the changes to disk. In the management VM, a monitoring program is run which catches changes to the `/root` directory in domU for 30 seconds using XenAccess disk monitoring capabilities. This is done by first initializing the engine and setting a watchpoint in `/root` by using the `xadisk_set_watch()` function. Next, the engine is activated and its output is directed to the standard output. The monitor's source code is shown in Program 5.2 and a sample execution is shown below for both domU (left) and dom0 (right).

```
domU ~ # mkdir foo       dom0 ~ # ./monitor /root
domU ~ # touch dummy     MKDIR: /root/foo
domU ~ # rm bar          MKFILE: /root/dummy
domU ~ # sync            RMFILE: /root/bar
```

**Program 5.2** This disk monitoring application outputs to the standard output all file/directory creation/removals happening in domU's /root directory as soon as they are committed to disk.

```
xadisk_t *x;
xadisk_obj_t *obj;
x = xadisk_init(1, FILE_IMAGE);
obj = xadisk_set_watch(x, argv[1]);
xadisk_activate(x, "/dev/xen/tapfifo0\0");
dup2(1, x->fifo_fd);
sleep(30);
xadisk_unset_watch(x, obj);
xadisk_destroy(x);
```

The simplicity of this example shows how XenAccess can be used to enable rapid development of similar monitoring applications, satisfying property (4).

From a security perspective, one application of this engine is a disk-based intrusion detection system [22, 12] whose goal is to detect suspicious file/directory creation/deletion commonly done by rootkits. In this case, the watchpoints would most likely be set in privileged system directories such as `/bin` and `/usr/bin`. An IDS deployed in this fashion has the obvious advantage of not requiring additional hardware in the hard drive, as done by traditional disk-based IDSes to achieve isolation. In our case, the isolation is provided by the VMM.

The need to manually perform a disk flush through the `sync` command in the example above illustrates one of the problems involved in this type of monitoring: the fact that changes made to the filesystem are not immediately committed to the disk in modern OSes. This can have serious implications for real-time disk-based IDSes, as it opens a window for evasion attacks. One way of addressing this problem is to use disk monitoring together with memory introspection: disk flushes could be externally enforced by actively manipulating the flushing timers in the guest OS memory through introspection. This approach is currently being investigated, along with others in which memory introspection and disk monitoring can be associated.

## 6 Discussion and Future Work

Stepping back to look at the six requirements given for a robust monitoring solution, we note that XenAccess satisfies each of these requirements. (1) The XenAccess Library uses an unmodified version of Xen as a VMM platform. (2) Using the capabilities provided by Xen, no special code needs to be inserted into the target OS. This is especially useful as it allows XenAccess monitors to work with both open and closed source target OSes. (3) Our performance testing shows that our address translation, memory copying, and disk I/O monitoring functions have small overheads, making these capabilities effective for a variety of monitoring applications. (4) Our example applications show that developing monitors with XenAccess is straightforward, with a minimal learning curve. (5) While our existing library implementation can view memory and disk I/O, the XenAccess architecture is easily extensible to collect any type of data from the target OS. (6) Finally, leveraging the protections provided by the VMM, XenAccess is sufficiently isolated from the target OS and any possibility of tampering by malicious software.

XenAccess currently provides a solid foundation for monitoring in a virtualized environment. Yet, our experiences working with virtual memory introspection and virtual disk monitoring highlighted some areas that would benefit from additional research. Introspection requires use of OS-specific information, as discussed in Section 4.2.1. This means that it is possible for an OS upgrade, hotfix, or patch to break the monitors. Ideally, XenAccess should provide an abstraction layer that dynamically adapts to these changes and provides a consistent interface to monitor applications. Finding techniques to enable this approach is still an open research problem.

For reasons of backwards compatibility, changes in filesystem structure and layout are very rare. So disk monitoring is not prone to the types of problems discussed above for introspection. Instead, we envision the future work in this space to focus on scalability, functionality, and HVM support. The current issues with scalability and functionality were discussed in Section 4.2.2. HVM support will require changes to tap into the QEMU device driver mechanism used by Xen.

## 7 Conclusion

This paper described XenAccess, a monitoring library for Xen virtual machines. XenAccess' development was guided by a set of design principles aimed at providing a solid foundation for secure and flexible virtual machine monitoring. XenAccess implements virtual memory introspection and virtual disk monitoring capabilities by leveraging Xen's existing infrastructure. By using it to access the target VM's raw memory pages and disk I/O, XenAccess is able to infer OS data structures and filesystem operations at a useful abstraction level.

Our evaluation revealed that XenAccess imposes a minimal performance overhead to the target OS memory and disk operation. We also showed practical examples of the type of information that memory introspection and disk monitoring can gather, illustrating the potential of each technique.

## Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0627430. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th Symposium on Operating System Principles*, 2001.

[2] K. Asrigo, L. Litty, and D. Lie. Using VMM-based sensors to monitor honeypots. In *Proceedings of the 2nd ACM/USENIX International Conference on Virtual Execution Environments*, 2006.

[3] P. Barford and V. Yegneswaran. An inside look at botnets. In *Advances in Information Security*. Springer, 2006.

[4] 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. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[5] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 3rd edition, 2005.

[6] J. Dike. *User Mode Linux*. Prentice Hall, 1st edition, April 2006.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, December 2002.

[8] P. Elango, S. Krishnakumaran, and R. H. Arpaci-Dusseau. Design choices for utilizing the disk idleness in a virtual machine environment. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2006.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted comput-

ing. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Network and Distributed System Symposium*, 2003.

[11] R. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34 – 45, June 1974.

[12] J. L. Griffin, A. Pennington, J. S. Bucy, D. Choundappan, N. Muralidharan, and G. R. Ganger. On the feasibility of intrusion detection inside workstation disks. Technical Report CMU-PDL-03-106, Carnegie Mellon University, 2003.

[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of the USENIX Annual Technical Conference*, June 2006.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, San Jose, CA, October 2006.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Oct 2005.

[16] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.

[17] N. L. Kelem and R. J. Feiertag. A separation model for virtual machine monitors. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 78 – 86, 1991.

[18] K. Kourai and S. Chiba. Hyperspector: Virtual distributed monitoring environments for secure intrusion detection. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

[19] R. Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.

[20] S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 210 – 224, March 1973.

[21] R. Meushaw and D. Simard. Nettop: A network on your desktop. *Tech Trend Notes (NSA)*, 9(4):3 – 11, Fall 2000.

[22] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[23] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[24] Qumranet, Inc. Kvm: Kernel-based virtualization driver.

[25] J. S. Robin and C. E. Irvine. Analysis of the Intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[26] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, 4th edition, 2004.

[27] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, December 2005.

[28] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Communications of the ACM*, 17(7), July 1974.

[29] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of FAST: 2nd USENIX Conference on File and Storage Technologies*, 2003.

[30] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[31] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, volume 36, pages 181 – 194, 2002.

[32] A. Warfield. Virtually persistent data. In *Xen Developer's Summit (Fall 2006)*, 2006.