Virtual Machine Introspection

Observation or Interference?

As virtualization becomes increasingly mainstream, virtual machine introspection techniques and tools are evolving to monitor VM behavior. A survey of existing approaches highlights key requirements, which are addressed by a new tool suite for the Xen VM monitoring system.

Kara Nance and Brian Hay
University of Alaska, Fairbanks

Matt Bishop
University of California, Davis

At one time, desktop computers were "one machine, one operating system, one application," forcing users to close one application to open another—and often to spend more time *waiting* than *doing* as a result. The advent of "one machine, one operating system, many applications" let users run multiple programs simultaneously and introduced a major step forward in computational evolution.

Today, virtualization lets users have "one machine, multiple operating systems, multiple applications" and switch between them at will. This not only lets developers easily test their programs on multiple OSs and enterprise users more effectively utilize hardware through server consolidation, it's also useful to computer users in general. When virtual machines are distributed with a set of preconfigured applications, users can easily utilize complex applications. Further, the isolation offered by VMs provides some security benefit, such as allowing general Web browsing while reducing the risk of compromise to the underlying physical system.

Although virtualization isn't new, the recent development of x86 virtualization products has revived interest in the virtualization market. This has led to the evolution of virtual machine introspection (VMI) techniques and tools to monitor VM behavior. VMI tools inspect a VM from the outside to assess what's happening on the inside.[1] This makes it possible for security tools—such as virus scanners and intrusion detection systems—to observe and respond to VM events from a "safe" location outside the monitored machine. Here, we survey and categorize the current crop of VMI technologies, then offer a detailed description of the Virtual Introspection for Xen (VIX) tool suite, which addresses key VMI requirements.

Virtualization overview

As Figure 1 shows, in a virtualized environment, a VM monitor provides the interface between each VM and the underlying physical hardware. The OS layer between a VMM and the physical hardware is optional, depending on which of the two major types of VM managers you choose.

In a type 1 system,[2] the VMM runs directly on the physical hardware, eliminating an abstraction layer and often improving efficiency as a result. Examples of type 1 systems include VMware ESX,[3] Xen (www.xensource.com/xen/xen/nfamily/virtualpc/default.mspx), and Microsoft Hyper-V (http://technet2.microsoft.com/windowsserver2008/en/servermanager/virtualization.mspx). In a type 2 system, the VMM uses an OS as an interface to the physical hardware. Type 2 systems include VMware Workstation, the QEMU open source process emulator (http://bellard.org/qemu/), KVM (http://kvm.qumranet.com/kvmwiki), Parallels (www.parallels.com), and Virtual PC/Server (www.microsoft.com/windows/products/wi). Type 2 systems rely on the underlying OS to provide hardware interaction and device drivers, and thus often have a wider range of physical hardware components to interact with.

To illustrate how virtualization works, we'll examine a simplified event sequence that occurs when a process attempts to access a memory address in its

 PUBLISHED BY THE IEEE COMPUTER SOCIETY ■ 1540-7993/08/$25.00 © 2008 IEEE ■ IEEE SECURITY & PRIVACY

virtual address space. From a process perspective, the request results in direct access to the memory address (see Figure 2a). However, as Figure 2b shows, while the OS layer has an active role in providing memory location access, it's actually abstracted from the process because it accesses the page table to map the logical memory address to a physical memory address. When the same request comes from a VM, it adds an additional level of complexity (see Figure 2c). To isolate the many VMs that might run on a single system, the VMM provides an abstraction layer between each VM OS's memory management and the underlying physical hardware. The VMM thus translates the VM-requested page frame number into a page frame number for the physical hardware, and thereby gives the VM access to that page.

Because of the VMM's active involvement in this process and its elevated privileges, it can also access memory pages assigned to each VM directly—without the VM actually requesting the page. The VMM can also make those pages accessible to other VMs on the system, which facilitates the VMI process.

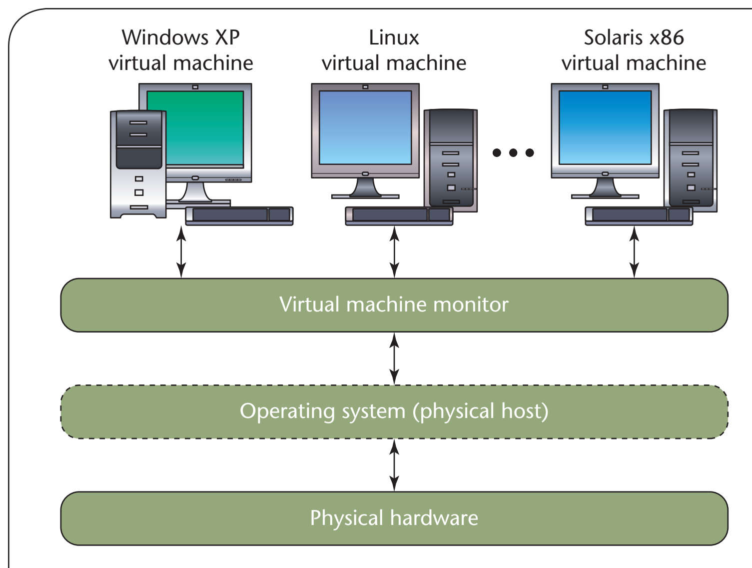

Figure 1. A generic system configuration for virtualization. The virtual machine monitor provides an interface between the underlying hardware and each VM. The operating system layer is optional, depending on the VM.

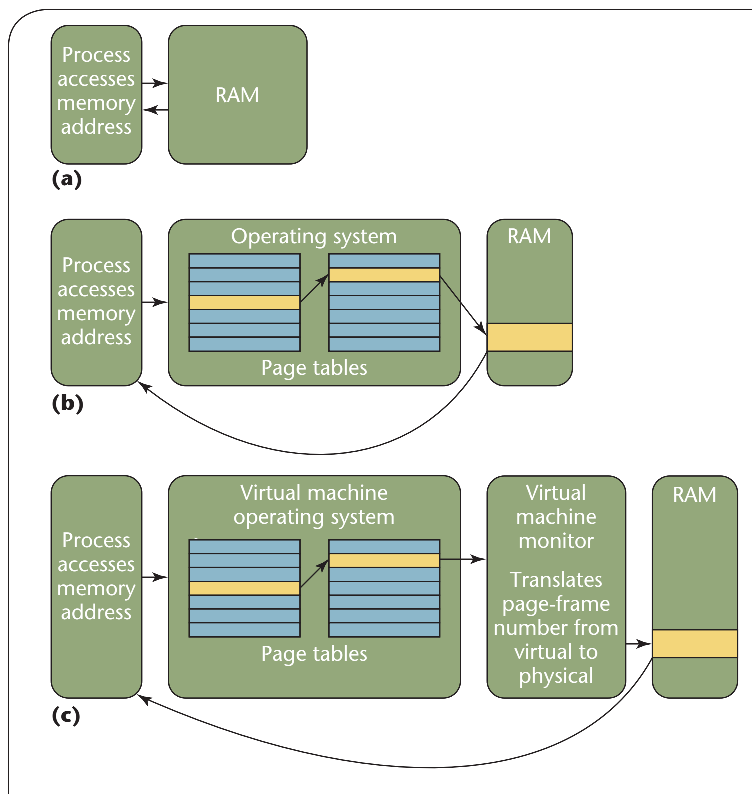

Figure 2. Memory mapping. The logical view from the perspective of (a) a process, (b) an operating system, and (c) a virtual machine monitor.

Virtual machine introspection

Many systems have implemented VMI. We classify these systems according to whether they interfere with a threat or simply monitor it; how much they know about the guest OS; and their ability to replay events.

Threat monitoring versus interfering

VMI systems fall into one of two categories: those that only monitor subject behavior and those that interfere with subject behavior.

For example, Livewire, an early host-based intrusion detection system, monitors VMs to gather information and detect attacks.[1] When it finds an attack, it merely reports it rather than interfering with it. In contrast, LycosID uses crossview validation techniques to compare running processes visible from high and low abstraction layers. The system then patches running code to enable reliable identification of hidden processes.[4] Manitou, a VMI designed to detect malware, compares known instruction-page hashes with memory-page hashes at runtime.[5] If no match is found, the instruction page is considered corrupted and marked as nonexecutable. Similarly, μDenali, a VMM, acts as a switch for network requests to a set of VMs; after a given time period, it can force a VM reboot.[6] Both LycosID and μDenali thus alter or interfere with the VM on the basis of an externally defined factor (the presence of a hidden process and time, respectively).

Our distinction between monitoring and interfering mirrors the security distinction between detection and response. A security mechanism using VMI to monitor a system can only detect and report problems, whereas one that can interfere can actually respond to a detected threat. It might, for example, terminate the relevant processes or VM, or

reduce the resources available to the VM to starve the attacker.

Semantic awareness

Our second axis of classification involves a VM's knowledge of its guest OS—that is, its *semantic awareness*. For example, Lares gives each VM an internal "hook" that activates an external monitoring control upon execution.[7] The monitor can then interrupt execution and pass control to a security mechanism. To achieve this, the hook is injected into the VM OS and the hypervisor write-protects both the hook and the code segment (or "trampoline") that transfers control to the security mechanism. Placing the hook so that it triggers at a meaningful system execution point requires an understanding of the OS's semantics. Thus, Lares must be semantically aware.

In contrast, AntFarm is specifically designed to monitor the VM's (virtual) memory management unit (MMU).[8] From that, it can construct the virtual-to-physical memory mapping and infer information about the machine's processes and OS. Hence, AntFarm is semantically unaware of the monitored system (although it builds up such an awareness over time). Some approaches, including IntroVirt,[9] attempt to bridge the "semantic gap" between the VMI application and the target VM by using functionality on the target VM itself to lend context to the acquired data. While this can be a useful approach in some cases, any such reliance runs the risk of deception by malware present in the target VM, just as would be possible if the VM were running as a process on the target itself.

This axis tells us whether the VMI can account for different guest OS characteristics and thus provide information that is more detailed. For example, a semantically aware VMI can parse kernel memory to build a process table map and hence process information. Semantically unaware VMI applications simply see memory as bits; most accumulate some knowledge of the guest OS and its processes over time, but they can't achieve the same familiarity as semantically aware VMI applications.

Event replay

The ability to replay, or log, events on a VM is useful not only for debugging OSs (which is why researchers introduced VMs in the late 1960s) but also for replaying compromises. ReVirt[10] is an example of a logging VMI; it serves as the basis for time-traveling VMs that allow replay from any previous VM state.[11] In contrast, Livewire and μDenali are logless, and instead analyze the current system state as it executes.

To allow replay, a VM must record enough information to reconstruct interesting portions of the system state. A logging VMI can replay events preceding unusual behavior until the cause is found, allowing deep analysis of security compromises. The penalty is that the VM or VMI must record extra information. The information's nature and amount varies depending on the replay's goals.

Security monitoring and VMI classifications

Our three categories capture the most important VMI properties for security monitoring.

- Threat monitoring versus interference captures the distinction between reading and writing.
- Semantic awareness captures the knowledge (or lack of knowledge) of context and environment that's critical to proper event interpretation.
- Event replay determines whether analysis must be performed in real time—as the target system executes—or at some later time under the analyst's control.

Using these three factors as a guide, you can select a VMI system that matches your security analysis requirements. All three classifications also take advantage of the VM's inability to interfere with the VMM's actions. Consider, for example, a terminate-and-stay-resident computer virus. If it loads before the antivirus program, the TSR can alter the intercept vectors so that they ignore it and other viruses. But a VM's malware can't alter VMM routines that check the VM pages containing the intercept vectors, and thus can't prevent the VMM antivirus mechanisms from detecting VM infection.

Digital forensic applications that use VMI differ from traditional digital forensic applications because they are covert; the data is thus untainted by the observer effect. Assume, for example, that your system has potentially been compromised and you want to apply digital forensics techniques to analyze the scenario. Traditionally, you'd shut down the machine, take an image of the disk, and forensically analyze it. In so doing, you would lose important RAM information, which likely contained forensically relevant information about the dynamic system state, such as which processes were running or which network connections were active

But, if you acquire evidence by reading the VM memory from a process external to the VM itself, the contents of memory and disk are available, and you avoid the need to attempt to reconstruct the system state solely from a static snapshot of the disk. Figure 2 shows the interaction between a process and its associated memory. For a VMI application to get to this step—and subsequently access memory associated with a particular process—it must identify the VM's individual processes. A VMI application might accomplish this by reconstructing the process list, then processing the data it contains to compute

the page table location for each process. From that, it can derive the individual page table entries. At this point, the VMI application can reconstruct the memory associated with each process. Given that and the process table's information on each process, it can determine exactly what each process was doing. This reconstruction can—indeed, should—be done with the VM paused, so that the VM's state can't change during reconstruction. This eliminates the observer effect because the VMI application doesn't execute in the VM's memory space and thus doesn't affect its contents.

Implementation

You can implement VMI applications in at least two system locations. One option is to embed the VMI application in the VMM itself. This requires you to modify the VMM code, and tends to make the VMI application highly dependent on the VMM version.

The second option is to place the VMI application outside the VMM. This is the option we chose using Xen, placing the VMI application in the privileged Dom0 VM. This makes the tools less likely to change as the VMM changes because they interact through a stable API. However, it might reduce the application's ability to perform inline processing (that is, to react to target VM requests in real time).

Virtual introspection for Xen

We developed and tested the Virtual Introspection for Xen (VIX) tool suite as a proof-of-concept VMI application.[12] We selected Xen for our project because it was open source, and thus let us modify or augment the VMM's functionality if necessary. Xen is also under active development; it's supported in several leading Linux distributions and has several mailing lists dedicated to its development and operation. However, the techniques we used in our project are also applicable to other virtualization platforms.

Xen overview

Xen is a type 1 VMM, so there's no underlying OS on the physical host. However, to provide a management interface for Xen—which the VMM itself doesn't provide—a special VM runs on the system at all times. In Xen, VMs are referred to as domains, and this special management domain is called Dom0 (see Figure 3). The VMM gives Dom0 system access to a control library, which lets the system administrator create, destroy, start, pause, stop, and allocate resources to VMs from Dom0. Dom0 also typically provides drivers for the host's physical hardware components, letting the other resident VMs—known collectively as DomU systems—utilize the hardware devices.

In addition to these common administrative functions, the Dom0 system can also request that memory

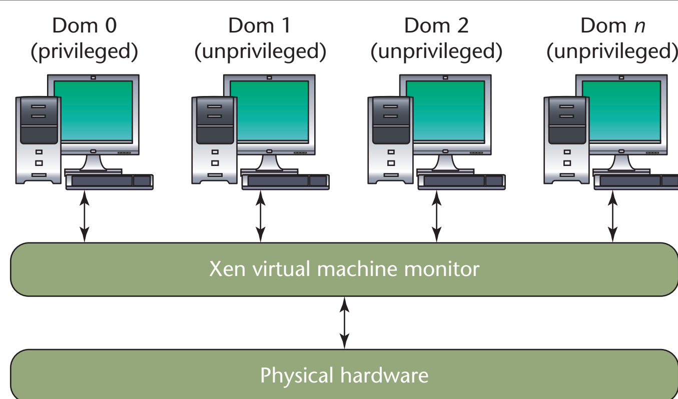

Figure 3. Xen system configuration. This configuration has *n* unprivileged virtual machines (domains) and a single privileged Dom0 VM, which provides a management interface.

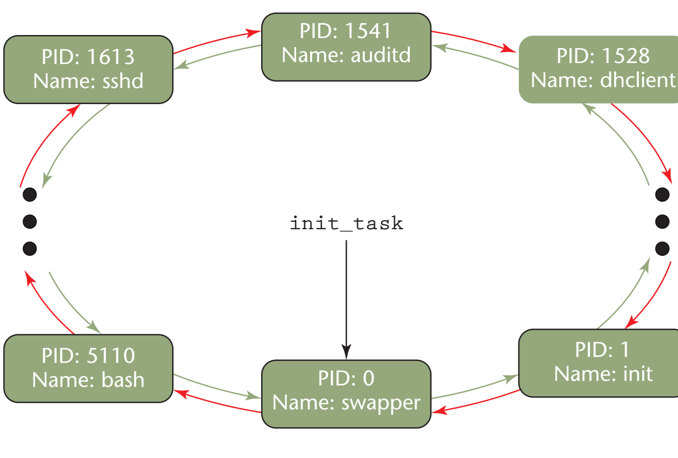

Figure 4. An example process list. A circular, double-linked list of *task_structs* in the Linux 2.6.x kernel.

pages allocated to unprivileged VMs be available to the Dom0 system. This allows a VMI application running within Dom0 to view the memory of any other VM on the system. Such functionality should be available only to the privileged Dom0 system, which should be reserved exclusively for management functionality. All other VMs should be restricted to accessing only the memory that the VMM has specifically allocated for their use.

How VIX works

Basically, VIX pauses operation of the target VM, maps some of its memory into the Dom0 system, acquires and decodes the memory pages' relevant data, and then resumes operation of the target VM. As an example, all current Linux system processes have an associated `task_struct` data structure that stores or links to information such as the process ID, pro-

cess name, memory map, and execution time. VIX can reference such data structures in many ways. Typically, it traverses the process list—a list of `task_structs` that the OS maintains. As Figure 4 shows,

An important outstanding question with respect to VMI is whether we can detect monitoring of the target VM—and if so, under what conditions and to what extent.

Linux stores this list as a circular double-linked list. Each `task_struct` contains the memory address of the previous and next `task_structs` in the list, and the end of the list links back to the start, forming a `task_structs` circle. Each kernel version has an associated memory address for the first process in the list, and from that address, VIX can easily traverse the entire list.

Accessing memory (such as a data structure) in a typical x86 application is a fairly trivial and fully automated task: the application requests a memory address within the process's address space, and the OS transparently translates the address into a page frame. For programmers writing virtual introspection applications, however, the process is more complex. Rather than having the OS map logical to physical addresses, the introspection program must manually traverse the page tables to convert the logical address to what the VM believes is a physical address but which is, in fact, simply another logical address to the underlying OS. Further, this provides a page frame only in the context of the VM, which believes it has contiguous physical RAM. To access the required data's actual physical memory page, the introspection program must perform a further manual translation between the VM page frame and the underlying physical host's page frames.

At this point, the VMI application has only obtained access to a memory page holding the requested data. The VMI application must still transform the raw data into useful information for the user. And, although Dom0 is observing the data structure, the structure is defined in the declaring DomU's system context. For example, the Dom0 and DomU system kernels might both define a `task_struct`, but might format it differently due to differences in kernel versions and configurations.

Furthermore, any of the observed data structure's memory references—such as pointers—are valid only in the context of the defined structure's address space on the VM being monitored. To dereference such pointers, the introspection program must (once again) manually traverse the entire page table of the VM-to-physical page frame translation.

In VIX, programs often have to carefully and repeatedly perform these operations during traversal of a linked list of Linux kernel data structures, such as the task list's `task_structs`. We define the `init_task` value for the VM's OS version; from this, we know the first `task_struct` data structure's memory location. From there, the VIX application `vix-ps` can traverse the entire task list. This approach lets VIX produce the same output as the `ps` command. It also allows the graphical system monitor to run within the VM itself, so that processes hidden to the VM user appear in the `vix-ps` listing. We can do this because VIX doesn't rely on any potentially compromised VM functionality in creating the process list. However, because VIX doesn't depend on any VM OS functionality for information, VMI applications can add other functionality.

Examples include running a sanity check for processes that aren't in the process list, but that appear in other kernel structures, such as the run queues. Such inconsistencies might indicate attempts to hide processes from the user, while still making them eligible for scheduling—a technique that rootkits use to ensure continued access to a compromised machine after the initial attack. We successfully implemented and demonstrated VIX's capability to detect such process inconsistencies that indicate malware presence.[12] We've since added several other tools to the VIX suite—including `vix-netstat`, `vix-lsof`, `vix-pstrings`, `vix-lsmod`, `vix-pmap`, and `vix-top`—that mimic the functionality of common non-VMI system tools.

Future investigations

An important outstanding question with respect to VMI is whether we can detect monitoring of the target VM—and if so, under what conditions and to what extent. It might seem that if the VMI application monitors the VM during the brief periods when the VM is not scheduled for execution and only reads data from the VM memory space, that it wouldn't modify the VM state, and thus, monitoring would go undetected from the perspective of a user (that is, an attacker) on the target VM. However, the attacker might be able to detect VMI using ancillary information. The VM could potentially detect unusual patterns in its scheduled execution frequency, or possibly question the page fault rate (where memory that was expected to be in RAM was paged out to disk, or vice versa). Detecting VM monitoring remains an open question, and one that deserves serious consideration if the results of VMI operations are to be used for security purposes. This is particularly important if organizations use VMI for digital forensics, for example, where the monitoring process results or effects can have real and serious legal consequences.

A second issue is whether it's possible for unprivileged VMs to compromise the VMM and thereby gain elevated access levels to the underlying physical host. Today, developers generally implement VMM as software, which means there might be bugs in the code that could leave the VMM vulnerable to compromise. This might result from an attacker carefully crafting input from the managed VMs, similar to the compromises possible in OSs today. As virtualization technology continues to develop, our hope is that developers will carefully craft VMMs with a view to simplicity, reliability, and sound security engineering practices. In contrast to many OS projects, where integrating new functionality often eclipses security and process-isolation needs, such high-assurance VMM development will let us apply VMI as reliable and unbiased reporters of VM activity.

While VMI is a relatively new research and development area, the Virtualization in Digital Forensics Research Agenda[13] recently identified it as one of the three target research areas within virtual environments analysis. Specifically, VDFRA identified the need for research on methods or mechanisms to monitor, filter, and analyze

- the interaction between the virtualized host and the underlying virtual or physical hardware it runs on; and
- the VM's internal state, including OS and process data structures.

Our own research team is continuing to address the technology's challenges, and our VIX tools suite offers a positive step forward in the advancement of VMI research. ☐

References

1. T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection-Based Architecture for Intrusion Detection," *Proc. 10th Symp. Network and Distributed System Security* (NDSS 03), Internet Society, 2003, pp. 191–206.
2. *IBM Systems Virtualization Version 2 Release 1*, IBM Corp., 2005; publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf.
3. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, white paper, VMware, 2007; www.vmware.com/files/pdf/VMware_paravirtualization.pdf.
4. S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "VMM-based Hidden Process Detection and Identification Using Lycosid," *Proc. ACM Int'l Conf. Virtual Execution Environments* (VEE 08), ACM Press, 2008, pp. 91–100.
5. L. Litty and D. Lie, "Manitou: A Layer-Below Approach to Fighting Malware," *Proc. Workshop Architectural and System Support for Improving Software Dependability* (ASID 06), ACM Press, 2006, pp. 6–11.
6. A. Whitaker et al., "Constructing Services with Interposable Virtual Hardware," *Proc. 1st Symp. Networked Systems Design and Implementation* (NSDI 04), Mar. 2004.
7. B. Payne et al., "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 2008, pp. 233–247.
8. S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "AntFarm: Tracking Processes in a Virtual Machine Environment," *Proc. Annual Usenix Tech. Conf.*, Usenix Assoc., 2008, pp. 1–14.
9. A. Joshi et al., "Detecting Past and Present Intrusions through Vulnerability-Specific Predicates," *Proc. Symp. Operating System Principles* (SOSP), 2005, pp. 91–104.
10. G.W. Dunlap et al., "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," *Proc. 2002 Symp. OS Design and Implementation* (OSDI 02), ACM Press, 2002, pp. 211–224.
11. S. King, G. Dunlap, and P. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," *Proc. Annual Usenix Tech. Conf.*, Usenix Assoc., 2005; www.usenix.org/events/usenix05/tech/general/king/king.pdf.
12. B. Hay and K. Nance, "Forensics Examination of Volatile System Data Using Virtual Introspection," *ACM Sigops OS Review*, vol. 42, no. 3, 2008, pp. 74–82.
13. M. Pollitt et al., "Virtualization and Digital Forensics: A Research and Education Agenda," *J. Digital Forensic Practice*, vol. 2, no. 2, 2008, pp. 62–73.

Kara Nance *is a professor and chair of the Department of Computer Science at the University of Alaska, Fairbanks, where she also directs the Advanced Systems Security Education, Research, and Training (ASSERT) Center. Her research interests include data systems and computer security. Nance has a PhD in computer science from the University of Oklahoma. Contact her at ffkln@uaf.edu.*

Matt Bishop *is a professor in the Department of Computer Science at the University of California, Davis. His research interests include vulnerabilities analysis and security policy modeling. Bishop has a PhD in computer science from Purdue University. His textbook,* Computer Security: Art and Science *(Addison-Wesley, 2002), is widely used in graduate and advanced undergraduate classes on computer security. Contact him at bishop@cs.ucdavis.edu.*

Brian Hay *is an assistant professor in the Department of Computer Science at the University of Alaska, Fairbanks, where he directs the Advanced System Security Education, Research, and Training (ASSERT) Lab. Hay has a PhD in computer science from Montana State University. Contact him at brian.hay@uaf.edu.*