IMPLICIT OPERATING SYSTEM AWARENESS IN A VIRTUAL MACHINE MONITOR

by

Stephen T. Jones

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Computer Sciences

at the

UNIVERSITY OF WISCONSIN–MADISON

2007

For my family

ACKNOWLEDGMENTS

So many people have contributed to the success of this endeavor. I am indebted to them all. I will mention just a few by name here.

First and most importantly I want to thank my wife and best friend Alyson whose persistent faith, love, encouragement, and support have been essential to all of my successes and have helped me learn from each of my failures. She got me started on this path and has seen me through to yet another beginning. My children Emma, Abbi, and Ian provide me with constant inspiration and perspective, often when I expect it least and need it most.

My advisors Andrea and Remzi gently and deftly shepherded me through the sometimes uncomfortable realization of what high quality research actually is. Their curiosity and enthusiasm about computer systems and, most importantly to me, their humanity form the core of what I take away from Wisconsin.

I thank my Parents Ben and Shirley Jones for giving me life, teaching me the value of work, and nurturing a love of discovery and learning. I also appreciate the diverse examples of all my wonderful siblings who continue to teach me whenever we are together. My brother Benjamin contributed especially to my love of design and elegant technology by generously sharing his time and ideas with me when I was young.

I want to gratefully acknowledge the financial support provided to me by Sandia National Laboratories and the collective friendship and wisdom of all of my Sandia colleagues with whom I am privileged to work. My manager Roxana Jansma deserves special mention. She has repeatedly stepped up to provide me both material and moral support for which I am deeply grateful. The amazing technical prowess of my friend and mentor Doug Ghormley originally motivated me to return to school and I have asymmetrically benefited from his example and advice.

While in Madison I have had the good fortune to interact with a fantastic group of people. Randy Smith has been a great friend with whom I have enjoyed sharing the travails and triumphs of graduate student life. My great research group colleagues Nitin Agrawal, Lakshmi Bairavasundaram, John Bent, Nathan Burnett, Tim Denehy, Haryadi Gunawi, Swetha Krishnan, Vijayan Prabhakaran, Florentina Popovici, and Muthian Sivathanu have taught me more than I ever cared to know about disk drives and storage systems.

Finally, I would like to thank my many friends in the Madison first ward who have made my time in Madison a spiritual as well as an intellectual journey.

Contents

List of Tables

List of Figures

ABSTRACT

Commodity server and desktop computer systems have become powerful enough in recent years to profitably make use of system virtualization technology. System software vendors are enthusiastically embracing system virtualization to address some of the key issues facing today's enterprises like manageability, rapid service deployment, and disaster recovery.

Widespread adoption of virtualization has a disruptive influence on system organization. In a virtualized environment, the virtual machine monitor (VMM) supplants the operating system as the primary resource manager. When a virtualization layer is present, certain system features like resource scheduling, cache management, and security monitoring can often be implemented most naturally within the VMM.

While a VMM understands and controls system hardware resources, it currently knows very little about the high-level software abstractions implemented within guest operating systems, a fact referred to as the "semantic gap". Information pertaining to OS constructs like processes, threads, users, and caches is often useful, however, when implementing services at the VMM layer. Hence, researchers have invented ways of directly exporting relevant information from the operating system to an underlying VMM. This direct approach, while effective, has some important drawbacks. For example, it leads to close coupling between VMM-layer services and specific OS vendors and versions, reducing the applicability of services and complicating deployment and management.

We have invented and implemented techniques that can be used by a VMM to infer useful information about selected operating system abstractions and achieve a level of implicit operating system awareness. Our approach uses observation of architectural events and the fact that modern operating systems share many basic features and responsibilities. This dissertation describes our techniques in detail and presents the results of a careful experimental evaluation of them. Using case studies, we show that implicit operating system awareness within a VMM can be used to implement a variety of useful applications like sophisticated I/O scheduling, flexible memory management, efficient caching, and reliable security monitoring that significantly enhance the value of the virtualization layer.

Chapter 1

Introduction

System virtualization technology has arrived on every major server and desktop computing platform [3, 4, 46, 93]. High quality virtual machine implementations for servers [4, 101] and desktop PCs [27, 91] allow system managers to consolidate servers [103], support multiple operating systems [36], provision resources on-demand [99], perform security isolation, monitoring, and authentication [34, 56], provide fault tolerance [11], and optimize for specialized architectures [12]. As both software [27] and hardware [37, 46] support for near zero overhead virtualization develops, and as virtualization is included in dominant commercial operating systems [8], it appears that virtualized computing environments will become ubiquitous.

As virtualization becomes prevalent, the *virtual machine monitor* (VMM), naturally supplants the operating system as the primary resource manager for a machine. In a virtualized environment, all physical system resources like CPUs, memory, and I/O peripherals are owned and managed by a VMM. Today, the operating system is the main target for innovation in system services. In a world where virtualized environments are the norm, one should consider how to implement some traditional operating system services like resource allocation, scheduling, and security monitoring within a VMM [15].

The transition of some functionality from the operating system into the VMM has many potential benefits. For example, VMM-based services can be portable across different operating systems. By implementing a feature a single time within a VMM, it is logically available to all operating systems running above. Further, the VMM may be the only place where new features can be introduced into a system, because the operating system is legacy or closed-source or both. The VMM is also the only locale in a virtualized system that has total control over system resources and can likely make the most informed resource management decisions.

1.1 Overcoming the Semantic Gap

However, pushing functionality down one layer, from the OS into the VMM, has its drawbacks as well. One significant problem is the lack of higher-level knowledge within the VMM, sometimes referred to as a *semantic gap* [15]. The semantic gap is a result of the narrow interface provided by the virtual architecture. The virtual machine interface isolates the VMM from guest operating systems and hides a guest's internal state from the VMM. For example, a VMM is not inherently privy to the semantics of a guest operating system's basic abstractions (*e.g.*, processes, threads, and users), its policies, or its performance goals. No standard interfaces exist that allow a VMM to query a guest operating system for these details. The semantic gap fundamentally limits the kinds of features a VMM-level service can provide.

Previous work in virtualized environments has partially recognized this dilemma and other researchers have developed techniques to infer information about one aspect of a guest operating system, namely, how it makes use of the hardware resources allocated to it [12, 85, 101]. Techniques that provide resource utilization information to a VMM are useful because they allow a VMM to manage the resources of the system more effectively. For example, armed with memory utilization information, a VMM can reallocate an otherwise idle page in one virtual machine to a different virtual machine that could use it more effectively [101].

In addition, some recently proposed VMM-based services use information about the *software abstractions* of the operating systems running above them. VMI [35], for example, uses debugging information about the specific version of its guest operating systems to extract implementation details like the memory addresses of private operating system variables and the layout of compound data structures. IntroVirt [52], requires a priori semantic information about key operating system functions like `fork`, `exec`, and `mmap` to stay informed of important guest events. These explicit approaches are effective, but have significant disadvantages. Explicit information closely couples a VMM-level service to a specific operating system version. A service based on externally provided, explicit guest implementation information must also *trust* that the guest operating system it observes has not been corrupted or compromised.

1.2 Research Statement

In response to these shortcomings, this dissertation explores how a VMM can independently obtain information about the software abstractions of the guest operating systems running above it in the software stack. Techniques that can be used to implicitly extract information about hardware or software components across a system layer boundary using observation and measurement are known as *gray-box* [5] techniques. When explicit information about a guest OS is inconvenient to obtain, unavailable, or unreliable, gray-box techniques can help bridge the semantic gap. In this dissertation, we describe, implement, and evaluate new gray-box techniques and algorithms that can be used by a VMM to implicitly obtain valuable information about two important software abstractions: *operating* *system processes* and the *unified buffer cache and virtual memory system*. In addition to obtaining information, we employ several case studies to show that our implicitly obtained information can be used as the basis of VMM-level services that enhance overall system performance.

One important finding of our work is that implicit information obtained within a VMM about a guest OS can be highly variable or noisy. Variability and noise can complicate making definitive statements about the current state of a guest. For some kinds of applications implemented at the VMM layer, like security monitoring, incorrect decisions can have disastrous results. In this dissertation, we show how statistical inference techniques can transform the naturally noisy implicit information available to a VMM into a reliable indicator of unwanted malicious activity. As a case study, we have built a highly accurate hidden process detection and identification service within a VMM that uses this transformed information to protect guest operating systems.

1.3 Process Information

The *process* is a key operating system concept. Processes provide many of the fundamental abstractions that programmers rely on, like private address spaces, and serve as the basic resource container and security isolation boundary for user tasks. We have developed a set of techniques that enable a virtual machine monitor to implicitly discover and exploit information about processes. By monitoring low-level interactions between guest operating systems and the memory management structures on which they depend, we show that a VMM can accurately determine when a guest operating system creates processes, destroys them, or context-switches between them.

Antfarm is the implementation of our process identification techniques for two different virtualization environments, Xen [27] and Simics [60]. We evaluate Antfarm as applied to multiple architecture and operating system combinations including x86/Linux, x86/Windows, and SPARC/Linux. This range of environments spans two processor families with significantly different virtual memory management interfaces and two operating systems with very different process management semantics.

We demonstrate the utility and efficacy of VMM-level process awareness by building an anticipatory disk scheduler [47] within a VMM. In a virtual machine environment, an anticipatory disk scheduler requires information from both the VMM and operating system layers and so cannot be implemented exclusively in either. Making a VMM process aware overcomes this limitation and allows an OS-neutral implementation at the VMM layer without any modifications or detailed knowledge of the operating system above. Our implementation within the VMM is able to improve throughput among competing sequential streams from processes across different virtual machines or within a single guest operating system by a factor of two or more. Antfarm imposes only a small runtime overhead of about 2.4% in a worst case scenario and about 0.6% in a more common, process-intensive compilation environment.

1.4 Buffer Cache Information

The unified operating system buffer cache and virtual memory system is critical to overall system performance. We have developed techniques to obtain information about the buffer cache by carefully observing guest operating system interactions with virtual hardware like the MMU and storage devices. Our methods detect when pages are inserted into or evicted from the buffer cache.

Geiger is an implementation of these techniques within the Xen VMM. Geiger significantly extends previous buffer cache related gray-box techniques by showing that a VMM must track more than just disk requests to accurately infer buffer cache evictions on modern operating systems. A VMM must also account for anonymous memory allocation to detect a whole new class of evictions when the buffer cache is unified with the virtual memory system. A VMM must also take basic file system behavior into account to accurately report certain cache events. For example, the VMM must track whether a particular data block is live or dead on disk in order to avoid reporting many spurious evictions. In addition, journaling file systems, such as ext3 in Linux, require the VMM to distinguish between writes to the journal and writes to other parts of storage to avoid an aliasing problem that leads to reporting false evictions.

We demonstrate how the inferred eviction information provided by Geiger can enable useful services inside a VMM by building multiple applications as case studies. The first case study represents a novel, VMM-based working set size estimator called MemRx [51] that complements existing techniques [101] by allowing estimation in the case that a VM is thrashing in virtual memory. A second study explores how Geiger-inferred evictions can be used by a VMM to enable remote storage caches to implement eviction-based cache placement [104] without changing the application or operating system storage interface, hence enhancing the adoption of this feature.

1.5 Security Applications

Stealth rootkits that can hide processes are a current and important security issue. Half of unpatched Windows systems surveyed by the *Microsoft Malicious Software Removal Tool* [63] are infested with a single stealth rootkit alone [67]. The ability to detect and respond to malicious hidden processes is a clear advantage in the race to defend network-attached computers.

Lycosid is our VMM-based security service that *detects* and *identifies* hidden processes. Lycosid is resilient to malicious guest attack by virtue of its location within a VMM. Unlike previous VMM-based security services, Lycosid does not depend on the guest operating system for trusted information, rendering it less susceptible to guest evasion attacks.

We have evaluated Lycosid using both Windows and Linux guests and show that it can accurately detect and identify hidden processes in a wide range of extremely challenging environments despite the fact that the implicitly obtained information about guest virtual machines it uses is noisy and sometimes wrong [49, 50]. Accuracy is achieved via a targeted use of statistical inference techniques like hypothesis testing and linear regression that trade time for accuracy. Despite uncertain inputs, Lycosid provides a robust, highly accurate service usable even in security environments where the consequences for wrong decisions can be high.

1.6 Contributions

The primary contributions of this dissertation are:

- The formulation and design of new gray-box techniques which allow a VMM to implicitly obtain accurate information about key events and the current state of guest operating system processes and the unified buffer cache and virtual memory system.
- The implementation of those techniques in a real virtual machine monitor and the evaluation of the implementation along several axes including accuracy, timeliness, and runtime overhead.
- The design and implementation of several case studies that demonstrate the feasibility of using implicit information to build real VMM-level optimizations and services.
- The identification of key system features and parameters that influence the accuracy and practical value of implicit information obtained at the VMM-level.
- The development and evaluation of algorithms, based on statistical inference, to overcome the fundamental variation and uncertainty in our VMM-based process information that enables us to use implicit information in a high consequence security environment.

1.7 Outline

The rest of this dissertation is organized as follows. In Chapter 2 we review the key features of virtual machine technology. In Chapter 3 we provide an overview of our implicit approach. Chapter 4 describes techniques to implicitly track guest OS processes. In Chapter 5 we present techniques that allow a VMM to observe guest OS buffer cache events. Chapter 6 presents our VMM-based hidden process detection and identification service. We survey related work in Chapter 7. Chapter 8 summarizes our findings, discusses lessons learned, and presents future work.

Chapter 2

Virtualization Background

Virtual machine techniques have been around for a long time. In fact, the earliest robust virtual machine implementations were essentially co-incident with the first multi-user operating systems [23, 65]. In this chapter we review the basic ideas behind virtualization technology. We will especially emphasize virtualization features that underlie our VMM-based gray-box techniques.

2.1 The Role of the Virtual Machine

There are several varieties of virtualization [87]. Process-level virtual machines virtualize a limited set of computer system features, including the user-level instruction set and application binary interface, for a single process. Examples of process-level virtual machines include the Java virtual machine runtime or the Microsoft CLR. System-level virtualization provides a complete virtualized computer system to a full operating system including the user and supervisor CPU instruction sets, memory, firmware, and peripheral devices. The primary purpose of system-level virtualization is to allow multiple operating systems to transparently share a single host computer system. This dissertation deals exclusively with system-level virtualization.

The motivation for running multiple operating systems on a single hardware host was originally to allow an expensive machine to be safely shared by independent organizations with different hardware and software requirements. For example, a department running a critical batch-oriented accounting system could share a computer with an engineering department developing the next series of application or system software. Sharing remains a primary application of system virtualization today where consolidating many underutilized, single purpose servers onto a smaller number of more fully utilized hosts can reduce procurement, management, and energy costs.

In a virtualized environment, safe sharing of resources among concurrently executing operating system instances is accomplished via strong isolation. Each operating system instance executes within a *virtual machine* (VM). Operating systems running within a virtual machine are called *guests*. Each VM is provided with virtual copies of system resources like CPUs, memory, disk storage, and network interfaces. The virtual copies are multiplexed in time or space onto the real physical resources by a thin layer of control software called the *virtual machine monitor* (VMM) or *hypervisor*. The VMM is the primary resource manager for a virtualized system, *i.e.*, it has the responsibility of allocating and scheduling access to all physical resources.

Virtual machine technology has advanced considerably in the past 40 years and is set to become a core feature in most server platforms. Virtualization has expanded its scope from large centralized computers like mainframes [23], to mini-computers [88], and (more recently) to PC-based servers and desktops [91]. Virtual machine techniques are experiencing a research and commercial renaissance and are being used to enable interesting new features like flexible resource management [101], workload migration [19, 82], service and device driver isolation [31, 59], security services [34, 35, 52], and fault tolerance [11]. All of IBM's POWER5-based server platforms now include an always-on, firmware-based hypervisor, the main-line Linux kernel now includes hosted virtualization features, and Microsoft plans to include a hypervisor as a core component in its next generation server operating system.

2.2 Deprivileged Operation

One of the key techniques that enables a virtual machine monitor to safely support multiple, concurrent operating systems is deprivileged operation. Normally, an operating system has complete and sole control over the underlying system hardware. That level of control cannot be shared safely among multiple operating systems. Hence, a VMM *deprivileges* all guest operating systems by executing them, including the kernel, in an unprivileged mode of the host computer's CPU. Sensitive operations, like those that affect system configuration or that directly access a shared resource, are not allowed in unprivileged modes. When a deprivileged operating system attempts to invoke a sensitive operation, the CPU generates an exception or *trap*. On startup, a VMM registers itself as the handler for all interrupts and exceptions. Thus, a VMM is informed via a trap whenever a guest operating system uses a sensitive instruction. Within the trap handler, a VMM may emulate the effects of the trapping instruction by, for example, updating virtual CPU registers, updating a page table entry, or initiating an I/O request. This general technique is called *trap and emulate virtualization* and is the most common virtualization technique used by VMMs today.

The techniques described in this dissertation rely on the ability of a VMM to observe certain exceptions delivered to it and to derive useful information about the internal states of its guest operating systems. Deprivileged operation ensures that the VMM gets that opportunity. The next sections describe the specific events we use and why a VMM is informed when they occur.

2.3 Virtualizing Memory

Inside a VMM, we make special use of information about how a guest operating system manages virtual address spaces. Specifically, we need to be informed about all page faults, page table updates, TLB flushes, and address space context switches.

A VMM receives notification of page faults because of its role in virtualizing system interrupts and exceptions. Delivery of page faults to the VMM is a natural consequence of interrupt processing. Information about TLB flushes, and address space context switches is available within a VMM because of its role in virtualizing the CPU's memory management unit (MMU).

To observe page table updates, however, a VMM must often employ additional techniques. The most common approach is called *shadow page tables* and ensures that a VMM retains control over virtual to physical address translation. In shadow paging, guest page tables are never used directly by the processor to perform address translation. A VMM employs its own page tables, called shadow page tables, to cache selected portions of a guest's page tables. In the shadow tables used by the processor, memory used for guest page tables is marked read-only so that a VMM is informed via a page fault when entries are updated. This allows the VMM to maintain consistency between its cached version and the original guest page tables. See Adams and Agesen's description of the VMware VMM [2] for a detailed description of one implementation of shadow page tables. By employing shadow page tables, a VMM can observe all relevant page table updates.

2.4 Virtualizing Disk I/O

Some of our techniques also use information about how a guest operating system utilizes disk storage. Information about disk requests is available to a VMM because it implements the virtual disk I/O devices that are available to a guest. An I/O request can be initiated by a guest in two different ways. A VMM may provide a virtual model of a real hardware device. An unmodified guest OS device driver communicates with such a device using memory mapped or programmed I/O [91]. The VMM can configure the underlying hardware to ensure that all such accesses are privileged. Hence, the VMM is informed when each operation occurs. Alternatively, a VMM may provide a high-level virtual device with which a virtual machine aware device driver within the guest communicates using a private interface similar to a system call [31]. In either case, the VMM can always observe the memory address, the disk address, and the operation type (read or write) of each disk I/O request.

2.5 Trap and Emulate Limitations

Some architectures, notably the Intel x86, include sensitive instructions which do not trap when invoked in an unprivileged mode. Instead, they fail silently. This type of instruction set is not formally virtualizable using only trap and emulate techniques [73]. To virtualize the x86, additional techniques like binary analysis and dynamic code translation are used by popular x86 VMMs like VMware and Microsoft Virtual Server [2, 45, 62]. Robin and Irvine [77] provide a comprehensive discussion of the problematic x86 instructions. By using binary analysis and code translation, a VMM like VMware receives the same set of notifications as a pure trap and emulate VMM.

2.5.1 Paravirtualization

Another virtualization approach called paravirtualization [27, 103] solves the problems associated with the x86 architecture by defining them away. A paravirtual VMM implements a slight different virtual architecture than the underlying host. It replaces problematic instructions with equivalent VMM operations similar to system calls. Hence, a significant benefit of paravirtualization on platforms like the x86 is reduced VMM complexity. A paravirtual VMM can also reduce virtualization overhead on any architecture by introducing private, streamlined interfaces for certain high-cost operations. The benefits of paravirtualization come at the cost of porting guest operating systems to the modified paravirtual architecture. One of our implementation platforms (Xen) has a paravirtual mode. Paravirtual Xen is notified of the same architectural events as a more conventional x86 VMM like VMware. The notification mechanism, however, is slightly different. Our results using paravirtual Xen are equally applicable to other VMMs that use trap and emulate or hybrid techniques.

2.5.2 Hardware Trends

More recent processors from Intel and AMD include virtualization extensions that transform the x86 into a formally virtualizable platform where classic trap and emulate virtualization can be used directly [3, 46]. The new processors also include features meant to improve virtualization performance by reducing the frequency of virtualization-related traps. These optimizations can, in some configurations, prevent a VMM from observing certain events. For example, when these features are in use it is no longer guaranteed that a VMM will observe every guest page fault. The techniques we describe in this dissertation depend on the ability of the VMM to observe events like page faults. Modifications to our techniques may be required if a CPU is configured to hide information from the VMM. This dissertation, as well as other research [2] show that software techniques have much to offer virtual machine performance and security. Commodity x86 processors that include virtualization extensions are still new and it remains to be seen which of their features will be used by VMMs in practice.

2.6 Summary

Our implicit techniques exploit the principle of deprivileged operation to ensure that the VMM is notified of critical configuration, exception, and I/O events, such as page faults, page table updates, and disk requests. There are different ways to implement deprivileged operation including trap and emulate, paravirtualization, and hardware assisted virtualization. In each case, the VMM and the underlying architecture can be configured to provide low overhead access to the notification events we rely on.

Chapter 3

Implicit Operating System Awareness

In this chapter we provide a high-level overview of our research goals and approach. We also discuss alternatives to our approach and how their advantages and disadvantages compare to ours in general.

3.1 Goals

Our primary research goal is to develop and evaluate techniques that enable the construction of practical VMM-level system services. We believe that a VMM is a natural place to implement certain kinds of system services because of the rapid spread of system virtualization technology. A practical VMM-level service should be easy to deploy and manage, impose low overhead, and retain the strong isolation and security properties of a VMM. Our techniques strive to enable easy deployment and management of VMM-based services through portability without compromising system performance or security.

3.2 Approach

Portability is one major factor leading to easy deployment of VMM-based services. A portable service can be installed in more diverse environments than an OS-specific solution. Planning and provisioning for a portable service can be accomplished independent of which operating system is selected to provide guest services. Ideally, a portable VMM-layer service should be implemented once and apply to any guest operating system the VMM encounters.

We have designed our techniques to be portable by avoiding the use of vendor or version-specific guest implementation information. Instead, we observe the stream of architecturally-defined events like page faults, hardware interrupts, configuration register updates, page table modifications, and I/O requests that are intrinsically visible to a VMM in its role as a service provider to guest operating systems. We employ a gray-box approach that applies a top-down, generic understanding of the common responsibilities and goals of modern operating systems to interpret the events delivered to a VMM and to infer useful information about the internal state of a guest OS. Limiting our use of guest knowledge to features and responsibilities that are generic across all the operating systems that a VMM supports means we can decouple our VMM-based services from guest peculiarities.

Our approach requires no new, non-standard interfaces between guests and the VMM. No modifications to the guest operating system or application software are required, which makes our approach equally applicable to legacy or closed guest software.

An additional benefit of the implicit approach is that the information we obtain reflects actual guest activity. A corrupt or compromised guest cannot hide information from or mislead the VMM except by changing its externally visible behavior, which is more difficult than simply supplying incorrect information. By way of analogy, it is easier to hide a building on a map provided to an adversary than it is to hide the building from an adversary standing next to the building. As we show in Chapter 6, this property can be especially useful in a security context.

3.2.1 Limitations

Our approach, however, is not perfect. It is unlikely that information on every aspect of a guest operating system that a VMM could find useful will be available implicitly. In this dissertation, we have limited ourselves to extracting information about two guest abstractions (processes and the buffer cache) that cast a strong architectural shadow, *i.e.*, for which virtual hardware is intimately involved. We believe, however, that information about additional abstractions can be implicitly obtained. For example, we have preliminary approaches for obtaining information about guest operating system threads and users. The well of implicit guest information is not yet dry.

We have also found that implicit information can be delayed or, in limited cases, wrong. A major contribution of our work is measuring this aspect of implicit information and demonstrating that many services are resilient to delay or short-term errors. Statistical techniques like hypothesis testing and regression have also proven effective in transforming highly variable implicit information into reliable intelligence about the internal state of a guest.

3.3 Alternative Approaches

There are other approaches for obtaining information about guest operating systems in support of VMM-services. These techniques can be divided into two categories. The first introduces new VMM-to-guest interfaces. The second uses explicitly provided details about how a specific guest operating system is implemented.

3.3.1 New Interfaces

New interfaces, through which arbitrary information about guest activities and state may be passed, can be added to a VMM. This approach has the significant advantage of being straightforward to implement and use. Information provided via such interfaces is timely and reflects the guest's true instantaneous state. The kinds of guest information available to a VMM is not limited when arbitrary interfaces between the VMM and guests exist. Any information the guest operating system has and is willing to export can be made available.

Adding new interfaces also has some interesting disadvantages; most importantly, these interfaces do not exist today. Current interface standardization efforts suggest that the process to define a standard VMM-to-guest interface will likely be long and contentious, leaving a large window of time in which alternative approaches will be required. Determining what types of guest information are most useful and how to provide that information to a VMM in a safe and portable way is an interesting question that researchers have begun to explore in related contexts [6, 38].

Guest operating systems must be modified to take advantage of new VMM interfaces. The cost of porting an operating system and subsequently maintaining a VMM-aware version can be high. Proponents of paravirtualization claim that the required changes to guest operating systems are minor [27]. Unfortunately, the changes required are often in the most complex and error-prone portions of an OS like the virtual memory and I/O systems. Other researchers cite the high engineering cost of porting operating systems to a paravirtual VMM as motivation for a clever, but complex, automated porting architecture [58]. In either case, the cost of creating and maintaining yet another operating system version is non-trivial.

Finally, adding interfaces may have negative security implications. Adding interfaces enlarges the attack surface of a VMM and tends to reduce the security advantages a VMM enjoys relative to other locations in a system. In addition, a VMM that depends on a guest to provide information about itself enters into an implicit relationship of trust with the guest. A buggy or compromised guest could mislead the VMM and thwart a VMM-based security monitoring service. Adding generic public interfaces to a VMM should be undertaken with extreme caution.

3.3.2 Explicit Information

A second approach for obtaining high-level guest information uses knowledge of explicit guest OS implementation details. Memory addresses of variables and the semantic information needed to interpret those variables are examples of the kind of explicit information a VMM can use to extract current guest state. Such information can often be derived from debugging symbols and debugging libraries. Additional information about the semantics of specific operating system functions can be obtained by reading source code or from binary reverse engineering.

Similar to the new interfaces technique, the explicit information approach provides access to timely information that corresponds exactly to the guest's view of its own current state. Rich, detailed information is available. Any information that is encoded in an interpretable guest data structure and any event that corresponds to a known guest function is available for consumption by the VMM.

Unfortunately, it may be inconvenient to get and maintain the explicit information that a VMM requires. If the guest operating system is legacy or closed, such information may simply be unavailable. Since implementation details can change between versions or even between patch-levels of a guest OS, keeping the information about guest memory and function locations up-to-date can be challenging. Microsoft's distributed, web-based debugging symbol repository is a testimony to how difficult it is to keep the debugging information that its partners use up-to-date for the many hundreds of active operating system versions and patch-levels it supports.

Reading information from OS data structures without understanding the locking protocol used to protect them from concurrent update could lead to inconsistent or corrupt data. Uhlig *et al.* [98], show that a processor in an unprivileged mode implies that no kernel locks are held and that all kernel data structures are consistent. On today's increasingly parallel hardware and multi-threaded applications, waiting for all processor cores to enter an unprivileged mode may severely restrict the opportunities of the VMM to access guaranteed consistent guest data directly.

3.4 Summary

Using implicit information to implement VMM-based services represents an unexplored region of the design space. It may be harder for the VMM to get the information it needs using only implicit techniques and that information may be subtly inaccurate. However, a VMM can use implicit information without knowing any details about its guest operating systems and no changes to those operating systems are required.

From a security standpoint, implicit information is less vulnerable to evasion attacks by a compromised guest OS because it is based on external observations of a running system rather than information supplied explicitly or implicitly by the guest itself. By using implicit techniques, a VMM need not trust the guest OS it observes.

Chapter 4

Tracking Guest Operating System Processes

This chapter introduces a set of techniques that enable a virtual machine monitor to implicitly discover and exploit information about one of the most important operating system abstractions, the *process*. Processes provide some of the basic simplifying illusions that help programmers manage complexity like large, flat, private address spaces and private CPUs. The process is the container within which each user program runs. Operating systems allocate and schedule resources to processes. The boundaries defined by a process are used to ensure program isolation. Each logical unit of a user's work is often encapsulated within by a process. Hence, knowledge about operating system processes can reveal useful information about resource usage, workload organization, scheduling policies, and security goals.

We show how a VMM can accurately infer when a guest operating system creates processes, destroys them, or context-switches between them. The basic mechanism consists of monitoring low-level interactions between guest operating systems and the memory management structures, like page tables and TLBs, on which they depend. These techniques achieve our portability goals by operating without any explicit information about the guest operating system vendor, version, or implementation details.

We demonstrate the utility and efficacy of VMM-level process awareness by building an anticipatory disk scheduler [47] within a VMM. In a virtual machine environment, an anticipatory disk scheduler requires information from both the VMM and the operating system layers, so it cannot be implemented exclusively in either. Making a VMM process aware overcomes this limitation and allows an OS-neutral implementation of anticipatory scheduling at the VMM layer without any modifications or detailed knowledge of the guest OS. Our implementation within the VMM is able to improve throughput among competing sequential streams of disk read requests from processes across different virtual machines or within a single guest operating system by a factor of two or more.

In addition to I/O scheduling, process information within the VMM has several other important applications, especially in the security domain. For example, it can be used to detect that processes have been hidden from system monitoring tools by malicious software, an application we discuss at length in Chapter 6. Code and data from particularly sensitive or vulnerable processes can be identified that should be monitored for runtime modification [35]. Patterns of system calls associated with a process can be used to recognize when a process has been compromised [33, 84]. In addition to just detecting intrusions, techniques exist to slow or thwart intrusions at the process level by affecting process scheduling [89]. Finally, process information can be used as the basis for discovering other high-level OS abstractions. For example, the parent-child relationship between processes can be used to identify groups of related processes associated with a *user*. All of these applications are feasible within a VMM only when process information is available.

Antfarm is the implementation of our process identification techniques for two different virtualization environments, Xen and Simics. We have evaluated Antfarm as applied to a range of platform and guest-OS combinations including x86/Linux, x86/Windows, and SPARC/Linux. This range of environments spans two processor families with significantly different virtual memory management interfaces and two operating systems with very different process management semantics, providing empirical evidence for our claim of portability. Antfarm imposes only a small runtime overhead of about 2.4% in a worst case scenario and about 0.6% in a more common, process-intensive compilation environment.

4.1 Background

The techniques we describe in this paper are based on the observations that a VMM can make of the interactions between a guest OS and virtual hardware. Specifically, Antfarm monitors how a guest uses a virtual MMU to implement virtual address spaces. In this section we review some of the pertinent memory management details of the Intel x86 and the SPARC architectures used by Antfarm.

4.1.1 x86 Virtual Memory Architecture

Our first implementation platform is the Intel x86 family of microprocessors. We chose the x86 because it is the most frequently virtualized processor architecture in use today. This section reviews the features of the x86 virtual memory architecture that are important for our inference techniques.

The x86 architecture uses a two-level, in-memory, architecturally-defined page table. The page table is organized as a tree with a single 4 KB memory page called the *page directory* at its root. Each 4-byte entry in the page directory can point to a 4 KB page of the *page table* for a process.

Each page table entry (PTE) that is in active use contains the address of a physical page for which a virtual mapping exists. Various page protection and status bits are also available in each PTE that indicate, for example, whether a page is writable or whether access to a page is restricted to privileged software.

A single address space is active per processor at any given time. System software informs the processor's MMU that a new address space should become active by writing the physical address of the page directory for the new address space into a processor control register (CR3). Since access to this register is privileged the VMM must virtualize it on behalf of guest operating systems.

TLB entries are loaded on-demand from the currently active page tables by the processor itself. The operating system does not participate in handling TLB misses.

An operating system can explicitly remove entries from a TLB in one of two ways. A single entry can be removed with the `INVLPG` instruction. All non-persistent entries (those entries whose corresponding page table entries are not marked "global") can be flushed from the TLB by writing a new value to CR3. Since no address space or process ID tag is maintained in the TLB, all non-shared entries must be flushed on context switch.

4.1.2 SPARC Virtual Memory Architecture

In this section we review the key aspects of the SPARC MMU, especially how it differs from the x86. We chose the SPARC as our second implementation architecture because it provides a significantly different memory management interface to system software than the x86.

Instead of architecturally-defined, hardware-walked page tables as on the x86, SPARC uses a software managed TLB, *i.e.*, system software implements virtual address spaces by explicitly managing the contents of the hardware TLB. When a memory reference is made for which no TLB entry contains a translation, the processor raises an exception, which gives the operating system the opportunity to supply a valid translation or deliver an error to the offending process. The CPU is not aware of the operating system's page table organization.

In order to avoid flushing the entire TLB on process context switches, SPARC supplies a tag for each TLB entry, called a *context ID*, that associates the entry with a specific virtual address space. For each memory reference, the current context is supplied to the MMU along with the desired virtual address. In order to match, both the virtual page number and context in a TLB entry must be identical to the supplied values. This allows entries from distinct address spaces to exist in the TLB simultaneously.

An operating system can explicitly remove entries from the TLB at the granularity of a single page or at the granularity of an entire address space. These operations are called `page demap` and `context demap` respectively.

4.2 Process Identification

The key to our process inference techniques is the logical correspondence between the abstraction *process*, which is not directly visible to a VMM, and the *virtual address space*, which is. This correspondence is due to the traditional single address space per process paradigm shared by all modern operating systems.

There are three major process events we seek to observe: creation, exit, and context switch. To the extent address spaces correspond to processes, these events are approximated by address space creation, destruction, and context switch. Hence, our techniques track processes by tracking address spaces.

Our approach to tracking address spaces on both x86 and SPARC is to identify a VMM-visible value with which we can associate a specific address space. We call this value an address space identifier (ASID). Tracking address space creation and context switch then becomes simply observing the use of a particular piece of VMM-visible operating system state, the ASID.

For example, when an ASID is observed that has not been seen before, we can infer that a new address space has been created. When one ASID is replaced by another ASID, we can conclude that an address space context switch has occurred. We identify address space deallocation by detecting when an ASID is available for reuse. We assume that the address space, to which an ASID refers, has been deallocated if its associated ASID is available for reuse.

4.2.1 Techniques for x86

On the x86 architecture we use the physical address of the page directory as the ASID. A page directory serves as the root of the page table tree that describes each address space. The address of the page directory is therefore characteristic of a single address space.

Process Creation and Context Switch

To detect address space creation on x86 we observe how page directories are used. A page directory is in use when its physical address resides in CR3. The VMM is notified whenever a guest writes a new value to CR3 because it is a privileged register. If we observe an ASID value being used that has not been seen before, we can infer that a new address space has been created. When an ASID is seen for the first time, the VMM adds it to an ASID registry that it maintains for tracking purposes. The ASID registry is similar to an operating system process list.

When a new value is written to CR3 it implies an address space context switch. By monitoring writes to this privileged register, a VMM always knows which ASID is currently "active".

Process Exit

To detect address space deallocation, we use knowledge about the generic responsibilities of an operating system to maintain address space isolation. Isolation requirements lead to distinctive operating system behavior that can be observed and exploited by a VMM to infer when an address space has been destroyed.

Operating systems must strictly control the contents of page tables being used to implement virtual address spaces. Process isolation could be breached if a page directory or page table page were reused for distinct processes without first being cleared of their previous contents. To ensure that no stale page table entries that point outside a process's allocated memory exist in reused page tables, Windows and Linux systematically clear the non-privileged portions of page table pages used by a process when it exits. Privileged portions of the page tables, which are used to implement the protected kernel address space, do not need to be cleared because they are shared between processes and map memory that is not accessible to untrusted software.

An operating system must also ensure that no stale entries remain in any TLB once an address space has been deallocated. Since the x86 architecture does not provide a way for entries from multiple address spaces to coexist in a TLB, a TLB must be completely flushed prior to reusing address space structures like the page directory. On x86, the TLB is flushed by writing a value to CR3, an event the VMM can observe.

Hence, to detect user address space deallocation, a VMM can keep a count of the number of *user* virtual mappings present in the page tables describing an address space. When this count drops to zero, the VMM can infer that one requirement for address space reuse has been met. It is simple for a VMM to maintain such a reference count because the VMM must be informed of all updates to a process's page tables so that it can reflect the changes in its shadow page tables. Multi-threading does not introduce additional complexity, because updates to a process's page tables are always be synchronized within the VMM for correctness.

By monitoring TLB flushes on all processors, a VMM can detect when the second requirement for address space deallocation has been met. Once both events have been observed for a particular ASID, the VMM can consider the corresponding address space dead and its entry in the ASID registry can be removed. A subsequent use of the same ASID implies the creation of a new and distinct process address space.

4.2.2 Techniques for SPARC

The key aspect that was used to enable process awareness on x86 is still present on SPARC. Namely, there is a VMM-visible identifier associated with each virtual address space. On x86 this was the physical address of the page directory. On SPARC we use the virtual address space context ID as an ASID. Making the obvious substitution leads to a process detection technique for SPARC similar to that for x86.

Creation and Context Switch

On SPARC, installing a new context ID is a privileged operation; hence, it is always visible to a VMM. By observing context ID switches, a VMM can maintain a registry of known ASIDs. When a new ASID is observed that is not in the ASID registry, the VMM can infer the creation of a new address space. Context switch is detected on SPARC whenever the context ID is changed on a processor.

	x86	SPARC
ASID	Page directory PA	Context ID
Creation	New ASID	New ASID
Exit	No user mappings and TLB flushed	Context demap
Context switch	CR3 change	Context ID change

Table 4.1: **Process identification techniques.** *The table lists the techniques used by Antfarm to detect each process event on the x86 and SPARC architectures.*

Exit

The only requirement for the reuse of a context ID on SPARC is that all stale entries from the previously associated address space be removed from each processor's TLBs. SPARC provides the context demap operation for this purpose. Instead of monitoring page table contents, as on x86, a VMM can observe context demap operations. If all entries for a context ID have been flushed from every processor it implies that the associated address space is no longer valid.

4.3 Resource Association

In addition to detecting process creation, exit, and context switch, *associating* other system events with particular processes is important to effectively utilize process information within a VMM. Processes are primarily important to the VMM in their role as containers for *resources*. Hence, associating resource consumption at the granularity of a process enables the VMM to make more informed and precise allocation and scheduling decisions. Examples of resource association that could be useful to a VMM include CPU processor time, disk and network I/O, and memory events like page cache insertion and eviction.

4.3.1 Context Association

The simplest and most generic means of associating resources with processes is to associate them in time. We call this method *context association*. Using the process identification and context switching inferences described previously, we can associate a specific process with a series of time intervals. The interval during which context association will attribute an event to a process begins when its address space is installed on the processor and ends when it is replaced by another process's address space or the virtual machine is de-scheduled. The advantages of this technique are its extreme simplicity and its generality: any event detectable by a VMM can be associated with a process using context association.

Unfortunately, context association is not always accurate, due to the *asynchrony* that is common within operating systems. Consider the case of a process making a disk request. If the operating system chooses to forward the request to the virtual disk device immediately, context association will attribute the request correctly to the issuing process. If, on the other hand, the operating system delays issuing the request, for example because other requests are ahead of it in the disk queue, the originating process is likely to be suspended and another process will be chosen to run. Hence, the request and the process have become decoupled in time.

4.3.2 Event Chaining

To overcome the inaccuracy of asynchronous event association, we develop a new technique: *event chaining*. The idea is to link synchronous events that occur in the context of the issuing process with the event of interest.

To improve the accuracy of disk read associations, for example, event chaining based on memory accesses can be used. When a read operation completes, the requesting process will likely access the resulting data in memory at some point in the future. This access may occur inside the operating system, for example, when the kernel copies the data into a user-supplied buffer; conventional read system call semantics lead to this behavior. Alternately, the access may occur at the resolution of a page fault, incurred by the process when it touches a page of a memory-mapped file for the first time. If we can identify any of these access events (which likely occur in the process's context that initiated the read), we can associate the related disk read more accurately with the issuing process.

One drawback of event chaining is that the more accurate results it provides are necessarily delayed. This can be problematic if the VMM wishes to make a decision based on process association at the time the event is detected and postponing the decision is not practical. However, even in such cases, event chaining can be used to detect and correct event misassociation, hence enabling recovery in some situations.

We have implemented event chaining for association of disk read requests, using access to the memory buffer where the read results are deposited as the chaining event. When the VMM receives a disk read request, the physical memory buffer into which the requested data is to be placed by the disk controller is recorded. When the request is ready to complete, all known existing virtual mappings for that physical page are invalidated such that any access using one of those mappings will result in a page fault and will be visible to the VMM. When such a fault occurs, the process in whose context the faulting address is located is associated with the original disk read request. The affected mapping is then returned to its original status, and the process can transparently proceed as normal.

4.3.3 Data Structures

To implement simple context association, all we need to track is which process is currently running, something we already do for basic process awareness.

Implementing I/O event chaining is more complex. To enable modification of all existing mappings for a given physical page frame, the VMM must maintain a reverse mapping data structure; this structure is roughly the same size as the normal set of page tables for each actively tracked process.

4.4 Antfarm Implementation

Antfarm is the name of the implementation of our process-awareness techniques. Antfarm has been implemented for two virtualization environments. The first, Xen [27], is a true VMM. The other is a low-level system simulator called Simics [60] which we use to explore process awareness for operating systems and architectures not supported by the version of Xen used in this research.

4.4.1 Antfarm for Xen

Xen is an open source virtual machine monitor for the Intel x86 architecture. Xen provides a paravirtualized [103] processor interface, which enables lower overhead virtualization at the expense of porting system software. We explicitly do *not* make use of this feature of Xen; hence, the mechanisms we describe are equally applicable to a more conventional virtual machine monitor such as VMWare [91, 101]. Because operating systems must be ported to run on Xen, proprietary commercial operating systems like Microsoft Windows are not currently supported.

Antfarm for Xen is implemented as a set of patches to the Xen hypervisor version 2.0.6. Changes are concentrated in the handlers for events like page faults, page table updates, and privileged register access. Additional hooks were added to Xen's back-end block device driver. The Antfarm patches to Xen, including debugging and measurement infrastructure, total approximately 1200 lines across eight files.

4.4.2 Antfarm for Simics

Simics [60] is a full system simulator capable of executing unmodified, commercial operating systems and applications for a variety of processor architectures. While Simics is not a virtual machine monitor in the strict sense of native execution of user instructions [73], it can play the role of a VMM by allowing Antfarm to observe and interpose on operating system and application hardware requests in the same way a VMM does. Simics allows us to explore process awareness techniques for SPARC/Linux and x86/Windows which would not be possible with a Xen-only implementation.

Antfarm for Simics is implemented as a Simics extension module. Simics extension modules are shared libraries dynamically linked with the main Simics executable. Extension modules can read or write OS and application memory and registers in the same way as a VMM.

Simics provides hooks called "haps" associated with various hardware events for which extension modules can register callback functions. Antfarm for Simics/x86 uses a hap to detect writes to CR3 and Antfarm for Simics/SPARC uses a hap to detect when the processor context ID is changed. Invocation of a callback is akin to the exception raised when a guest OS accesses privileged processor registers on a true VMM. A memory write breakpoint is installed by Antfarm for Simics/x86 on all pages used as page tables so that page table updates can be detected. A VMM like Xen marks page tables read-only to detect the same event.

Antfarm for Simics/x86 consists of about 800 lines of C code. For Simics/SPARC the total is approximately 450 lines.

4.5 Process Awareness Evaluation

In this section we explore the accuracy of Antfarm in each of our implementation environments. We also characterize the runtime overhead of Antfarm for Xen. Our analysis of accuracy is decomposed into two components. The first measures the ability of Antfarm to correctly detect process creations, exits, and context switches. We call this aspect *completeness*. The second component we explore is the time difference or *lag* between process events as they occur within the operating system and when they are detected by the VMM.

4.5.1 x86 Evaluation

We evaluate Antfarm for x86 as implemented within the Xen hypervisor version 2.0.6. Version 2.6.11 of the Linux kernel was used in Xen's privileged control VM. Linux kernel version 2.4.30 and 2.6.11 are used in unprivileged VMs as noted. Our evaluation hardware consists of a 2.4 GHz Pentium IV PC with 512 MB of RAM. Virtual machines are each allocated 128 MB of RAM in this environment.

We also evaluate our techniques as applied to Microsoft Windows NT4 guests. Since Windows is not supported by Xen 2.0, Simics/x86 is used for this purpose. Our Simics/x86 virtual machines were configured with a 2.4 GHz Pentium IV CPU and 256 MB of RAM.

Completeness

To quantify completeness, each guest operating system was instrumented to explicitly report process creation, exit, and context switch. The resulting event records include the appropriate ASID, as well as the time of the event obtained from the processor's cycle counter. These OS traces were compared to similar traces generated by Antfarm. Guest OS traces are functionally equivalent to the information that would be provided by a paravirtualized OS that included a process event interface. Hence, our evaluation implicitly compares the accuracy of Antfarm to the ideal represented by a paravirtual interface.

In addition to process creation, exit, and context switch, guests report address space creation and destruction events so that we can discriminate between errors caused by a mismatch between processes and address spaces and errors caused by inaccurate address space inferences made by Antfarm.

We categorize incorrect inferences as either false negatives or false positives. A false negative occurs when a true process event is missed by Antfarm. A false positive occurs when Antfarm incorrectly infers events that do not exist.

To determine if false negatives occurred, one-to-one matches were found for every OS-reported event in each pair of traces. To be considered a match we require that the Antfarm event have the same ASID, and that it occur within the range for which the event is plausible. For example, to match an OS process-creation event, the corresponding event

	Proc Create	ASpc Create	Inf Create	Proc Exit	ASpc Exit	Inf Exit	Ctxt Switch	CS Inf
Linux 2.4								
Fork Only	1000	1000	1000	1000	1000	1000	3331	3331
Fork + Exec	1000	1000	1000	1000	1000	1000	3332	3332
Vfork + Exec	1000	1000	1000	1000	1000	1000	3937	3937
Compile	815	815	815	815	815	815	4447	4447
Linux 2.6								
Fork Only	1000	1000	1000	1000	1000	1000	3939	3939
Fork+Exec	1000	**2000**	**2000**	1000	**2000**	**2000**	4938	4938
Vfork + Exec	1000	1000	1000	1000	1000	1000	3957	3957
Compile	748	**1191**	**1191**	748	**1191**	**1191**	2550	2550
Windows								
Create	1000	1000	1000	1000	1000	1000	74431	74431
Compile	2602	2602	2602	2602	2602	2602	835248	835248

Table 4.2: **Completeness for x86.** *The table shows the total number of creations and exits for processes and address spaces reported by the operating system. The total number of process creations and exits inferred by Antfarm are shown in comparison. Antfarm detects all process creates and exits without false positives or false negatives on both Linux 2.4 and Windows. Fork and exec, however, lead to false positives under Linux 2.6 (***bold face values***). All false positives are due to the mismatch between address spaces and processes, which is indicated by matching counts for address space creates and inferred creates. Actual and inferred context switch counts are also shown for completeness and are accurate as expected.*

inferred by Antfarm must occur after any previous OS-reported process exit events with the same ASID and before any subsequent OS-reported process creation events with the same ASID.

Table 4.2 reports the process and address space event counts gathered by our guest OSes and by Antfarm during an experiment utilizing two process-intensive workloads. The first workload is synthetic. It creates 1000 processes, each of which runs for 10 seconds then exits. The process creation rate is 10 processes/second. On Linux, this synthetic workload has three variants. The first creates processes using fork only; the second uses fork followed by exec; the third employs vfork followed by exec. Under Windows, processes are created using the CreateProcess API.

The second workload is a parallel compile of the bash shell sources using the command "make -j 20" in a clean object directory. A compilation workload was chosen because it creates a large number of short-lived processes, stressing Antfarm's ability to track many concurrent processes that have varying runtimes.

Antfarm incurs no false negatives in any of the tested cases, *i.e.*, all process-related events reported by our instrumented OSes are detected by the VMM. The fact that inferred counts are always greater than or equal to the reported counts suggests this, but we also verified that each OS-reported event is properly matched by at least one VMM-inferred event.

Under Linux 2.4 and Windows, no false positives occur, indicating Antfarm can precisely detect address space events and that there is a one-to-one match between address spaces and processes for these operating systems. Under Linux 2.6, however, false positives do occur, indicated in Table 4.2 by the inferred event counts that are larger than the OS-reported counts. This discrepancy is due to the implementation of the Linux 2.6 fork and exec system calls.

UNIX programs create new user processes by invoking the fork system call which, among other things, constructs a new address space for the child process. The child's address space is a copy of the parent's address space. In most cases, the newly created child process immediately invokes the exec system call which replaces the child's virtual memory image with that of another program read from disk.

In Linux 2.4, when exec is invoked the existing process address space is cleared and reused for the newly loaded program. In contrast, Linux 2.6 destroys and releases the address space of a process invoking exec. A new address space is allocated for the newly exec'd program. Hence, under Linux 2.6, a process that invokes exec has two distinct address spaces associated with it, which do not overlap in time. In other words, the runtime of the process is *partitioned* into two segments. One segment corresponds to the period between fork and exec and the other corresponds to the period between exec and process exit. Antfarm, because it is based on address space tracking, concludes that two different processes are created leading to twice as many inferred process creations and exits as actually occurred.

Due to the idiomatic use of fork and exec, however, a process is partitioned in a distinctive way. The Linux 2.6/x86 case in Figure 4.1 depicts the temporal relationship between the two inferred pseudo-processes. The duration of the first pseudo-process will nearly always be small. For example, in the case of our compilation workload, the average time between fork and exec is less than 1 ms, compared to the average lifetime of the second pseudo-process, which is more than 2 seconds, a difference of three orders of magnitude.

The two pseudo-processes are separated by a short time period where neither is active. This interval corresponds to the time after the original address space is destroyed and before the new address space is created. During the compilation workload this interval averaged less than 0.1 ms and was never larger than 2.3 ms. Since no user instructions can be executed in the absence of a user address space, the combination of the two pseudo-processes detected by Antfarm encompasses all user activity of the true process. Conventional use of fork and exec imply that nearly all substantive activity of the true user process is captured within the second pseudo-process.

Lag

The second aspect of process identification accuracy that we consider is the time difference between a process event and when the same event is detected by the VMM. We define a process to exist at the instant the fork (or its equivalent) system call is invoked. Exit is defined as the start of the exit system call. These definitions are maximally conservative. In Figure 4.1 create lag is labeled `A` and exit lag is labeled `B`.

Lag is similar in nature to response time, so we expect it to be sensitive to system load. To evaluate this sensitivity, we conduct an experiment that measures lag times for various levels of system load on Linux 2.4, Linux 2.6, and Windows. In each experiment,

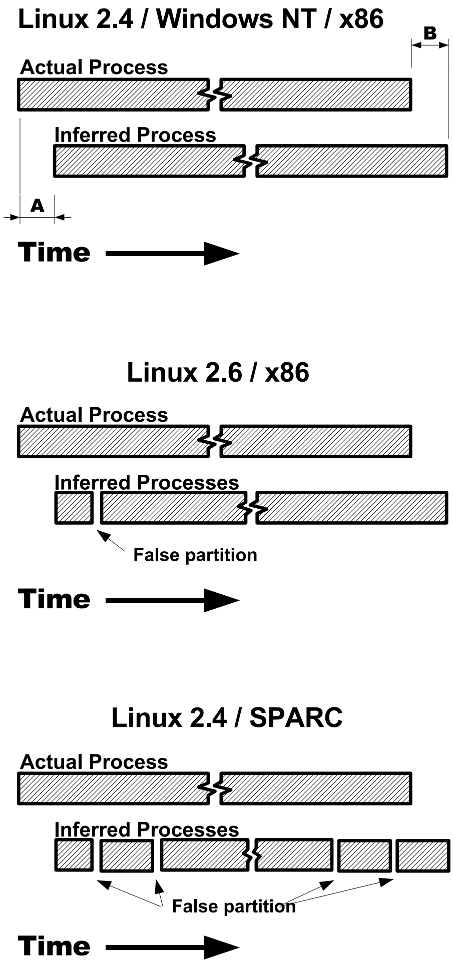


Figure 4.1: **Effects of error.** *The figure shows where each type of process identification error occurs for each tested platform. Error is either lag between when the true event occurs and when the VMM detects it, (e.g., A and B in the figure) or consists of falsely partitioning a single OS process into multiple inferred processes. In Linux 2.6/x86, this only occurs on* `exec`*, which typically happens immediately after fork. On SPARC this partitioning happens whenever a process calls either* `fork` *or* `exec`*.*

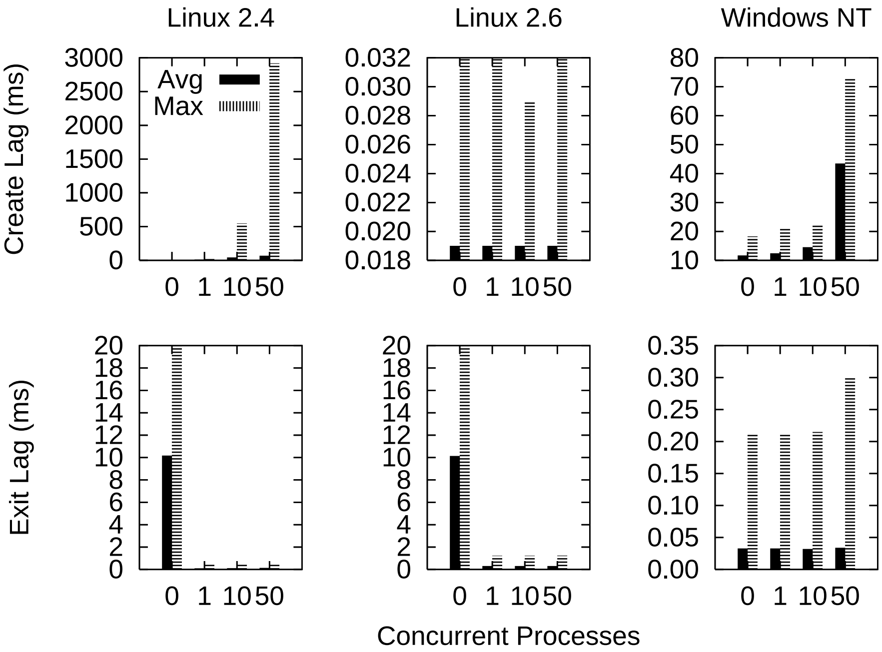


Figure 4.2: **Lag vs. System Load.** *The figure shows average and maximum create and exit lag time measurements for a variety of system load levels in each of our x86 evaluation environments. Average and worst case create lag are affected by system load in Linux 2.4 and Windows, but are small and nearly constant under Linux 2.6. Except for a large exit lag with no competing processes on Linux, exit lag does not appear to be sensitive to system load.*

0, 1, 10, or 50 CPU-bound processes were created. 100 additional test processes were then created and the create and exit lag time of each were computed. Test process creations were separated by 10 ms and each test process slept for one second before exiting.

The results of these experiments are presented in Figure 4.2. For each graph, the x-axis shows the number of concurrent CPU-bound processes and the y-axis shows lag time. Create lag is sensitive to system load on both Linux 2.4 and Windows, as indicated by the steadily increasing lag time for increasing system load. This result is intuitive since a call to the scheduler is likely to occur between the invocation of the create process API in the parent (when a process begins) and when the child process actually runs (when the VMM detects it). Linux 2.6, however, exhibits a different process creation policy that leads to relatively small and constant creation lag. Since Antfarm detects a process creation when a process first runs, the VMM will always be informed of a process's existence before any user instructions are executed.

Exit lag is typically small for each of the platforms. The exception is for an otherwise idle Linux which shows a relatively large exit lag average of 10 ms. The reason for this anomaly is that most Linux kernel tasks, including the idle task, do not need an associated user address space and therefore borrow the previously active user address space when they need to run. This mechanism allows a kernel task to run without incurring the expense of a TLB flush. In the case of this experiment, test processes were started at intervals of 10 ms and each process sleeps for one second; hence, when no other processes are ready to run, approximately 10 ms elapse between process exit and when another process begins. During this interval, the Linux idle task is active and prevents the previous address space from being released, which leads to the observed delay.

The Big Picture

Figure 4.3 shows a set of timelines depicting how Antfarm tracks process activity over time for a parallel compilation workload on each of our x86 platforms. The top curve in each graph shows the true, current process count over time as reported by the operating system. The middle curve shows the current process count as inferred by Antfarm. The bottom curve shows the difference between the two curves calculated as $Inferred - Actual$.

The result of the relatively large creation lag under Linux 2.4 is apparent in the larger negative process count differences compared to Linux 2.6. For this workload and metric combination, creation lag is of greater concern than the false positives experienced by Linux 2.6. In another environment such as a more lightly loaded system, which would tend to reduce lag, or for a metric like total cumulative process count, the false positives incurred by Linux 2.6 could be more problematic.

Exit lag is not prominent in any of the graphs. Large, persistent exit lag effects would show up as significant positive deviations in the difference curves. The fact that errors due to fork and exec do not accumulate over time under Linux 2.6 is also apparent because no increasing inaccuracy trend is present.

4.5.2 Overhead

To evaluate the overhead of our process awareness techniques we measure and compare the runtime of two workloads under Antfarm and under a pristine build of Xen. The first workload is a microbenchmark that represents a worst case performance scenario for Antfarm. Experiments were performed using Linux 2.4 guests.

Since our VMM extensions only affect code paths where page tables are updated, our first microbenchmark focuses execution on those paths. The program allocates 100 MB of memory, touches each page once to ensure a page table entry for every allocated page is created and then exits, causing all of the page tables to be cleared and released. This program is run 100 times and the total elapsed time is computed. The experiment was repeated five times and the average duration is reported. There was negligible variance between experiments. Under an unmodified version of Xen this experiment required an average of 24.75 seconds to complete. Under Antfarm for Xen the experiment took an average of 25.35 seconds to complete. The average slowdown is 2.4% for this worst case example.

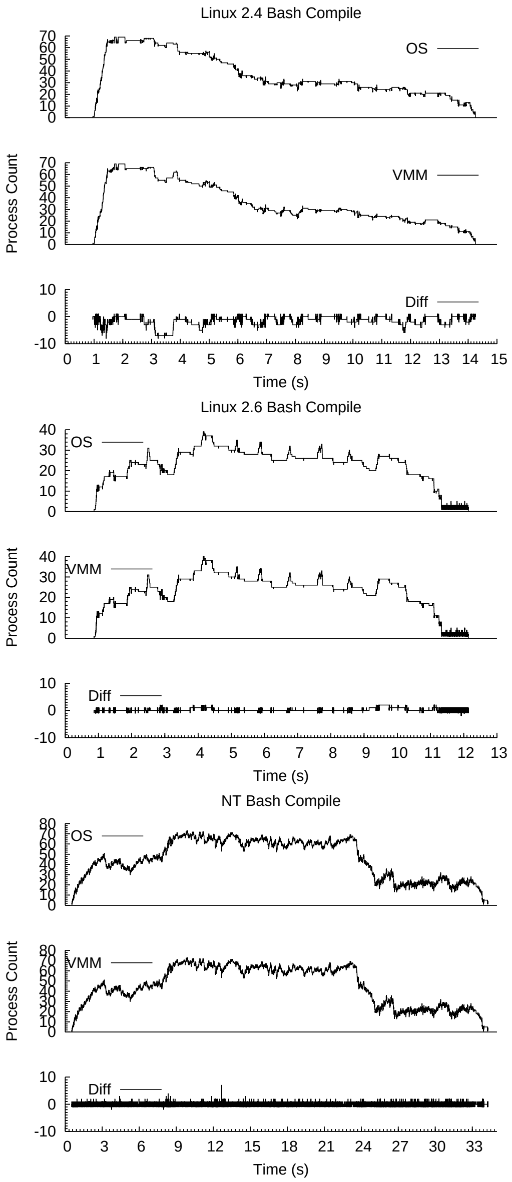


Figure 4.3: **Compilation Workload Timelines.** *For x86/Linux 2.4, x86/Linux 2.6 and x86/Windows a process count timeline is shown. Each timeline depicts the OS-reported process count, the VMM-inferred process count and the difference between the two versus time. Lag has a larger impact on accuracy than false positives. x86/Linux 2.6, which exhibits significantly smaller lag than x86/Linux 2.4 is able to track process counts more accurately.*

	Proc Create	ASpc Create	Inf Create	Proc Exit	ASpc Exit	Inf Exit	Ctxt Switch	CS Inf
SPARC/Linux								
Fork Only	1000	1000	**2000**	1000	1000	**2000**	3419	3419
Fork & Exec	1000	1000	**3000**	1000	1000	**3000**	3426	3426
Vfork	1000	1000	1000	1000	1000	1000	4133	4133
Compile	603	603	**1396**	603	603	**1396**	1678	1678

Table 4.3: **Completeness for SPARC.** *The table shows the results for the same experiments reported for x86 in Table 4.2, but for SPARC/Linux 2.4. False positives occur for each fork due to an implementation which uses copy-on-write. Antfarm also infers an additional, non-existent exit/create event pair for each exec. This error is not due to multiple address spaces per process as on x86, but rather stems from the flush that occurs to clear the caller's address space upon exec.*

The runtime for configuring and building bash was also compared between our modified and unmodified versions of Xen. In the unmodified case the average measured runtime of five trials was 44.49 s. The average runtime of the same experiment under our modified Xen was 44.74 s. The variance between experiments was negligible yielding a slowdown of about 0.6% for this process-intensive application workload.

4.5.3 SPARC Evaluation

Our implementation of process tracking on SPARC uses Simics. Each virtual machine is configured with a 168 MHz UltraSPARC II processor and 256 MB of RAM. We use SPARC/Linux version 2.4.14 as the guest operating system for all tests. We instrumented the guest operating system to report the same information as described for x86.

Completeness

We use the same criteria to evaluate process awareness under SPARC as under x86. Table 4.3 lists the total event counts for our process creation micro-benchmark and for the bash compilation workload.

As on x86, no false negatives occur. In contrast to x86, the fork-only variant of the microbenchmark incurs false positives. The reason for this is the copy-on-write implementation of fork under Linux. During fork all of the writable portions of the parent's address space are marked read-only so that they can be copy-on-write shared with the child. Many entries in the parent's page tables are updated and all of the corresponding TLB entries must be flushed. SPARC/Linux accomplishes this efficiently by flushing all of the parent's current TLB entries using a context demap operation. The context demap is incorrectly interpreted by Antfarm as a process exit. As soon as the parent is scheduled to run again, we detect the use of the address space and signal a matching spurious process creation.

The false positives caused by the use of fork under SPARC are different in character than those caused by exec under x86. These errors are not limited (by convention) to the usually tiny time interval between fork and exec. They will appear whenever fork is invoked, which for processes like a user shell can occur repeatedly throughout the process's lifetime. The Linux 2.4/SPARC case in Figure 4.1 depicts how a process that repeatedly invokes fork might be partitioned into many inferred pseudo-processes by Antfarm.

When exec is used we see additional false positives, but for a different reason than under x86/Linux 2.6. In this case the process inference technique falsely reports the creation of new address spaces that don't really exist. The cause of this behavior is a TLB demap operation that occurs when a process address space is cleared on exec. This error mode is different than under x86 where observed errors were due to a faulty assumption of a single address space per process. On SPARC, the error occurs because our chosen indicator, context demap, can happen without the corresponding address space being deallocated.

Given these two sources of false positives, one would expect our compilation workload to experience approximately the same multiple of false positives as seen for the fork+exec synthetic benchmark. We see, however, fewer false positives than we expect, due to the use of vfork by both GNU make and gcc. Vfork creates a new process but does not duplicate the parent's address space. Since no parent page tables are changed, no flush is required. When exec is invoked we detect the creation of the *single* new address space. Hence, when vfork and exec are used to create new processes under SPARC/Linux, Antfarm experiences no false positives. The build process, however, consists of more than processes created by make and gcc. Many processes are created by calls to an external shell and these process creations induce the false positives we observe.

Lag

Lag between OS-recorded and VMM-inferred process events under SPARC/Linux is comparable to Linux on x86. The average and maximum lag values for SPARC/Linux under various system loads are shown in Figure 4.4. Create lag is sensitive to system load. Exit lag is unaffected by load as on x86.

Limitations

While the SPARC inference technique is simple, it suffers drawbacks relative to x86. As shown, the technique incurs more false positives than the x86 techniques. In spite of the additional false positives, Figure 4.5 shows that the technique can track process events during a parallel compilation workload at least as accurately as x86/Linux 2.4.

Unlike the x86, where one can reasonably assume that a page directory page would not be shared by multiple runnable processes, one cannot make such an assumption for context IDs on SPARC. The reason is the vastly smaller space of unique context IDs. The SPARC provides only 13 bits for this field which allows up to 8192 distinct contexts to be represented concurrently. If a system exceeds this number of active processes, context IDs must necessarily be recycled. In some cases, system software will further limit the number of concurrent contexts it supports. For example, Linux on SPARC architectures uses only 10 of the available 13 context bits, so only 1024 concurrent address spaces are supported without recycling.

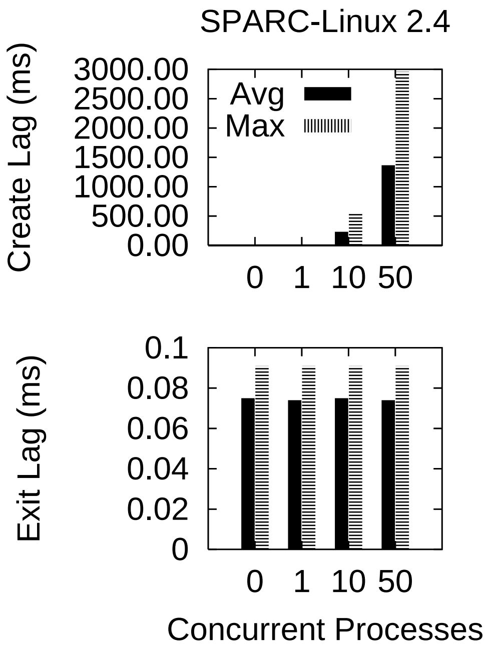


Figure 4.4: **Lag vs. System Load, SPARC.** *The figure shows average and maximum create and exit lag time measurements for the same experiments described in Figure 4.2. Create lag grows with system load. Exit lag is small and nearly constant, independent of load.*

Figure 4.6 shows the behavior of our SPARC process detection techniques when more processes exist than can be distinguished by the available context IDs. Once the limit is reached at 1024, the technique fails to detect additional process creations.

The importance of this second limitation is somewhat reduced because even very busy servers rarely have more than 1000 active processes, a fact which no doubt influenced the selection of the context ID field's size.

Overhead

Since our SPARC techniques are implemented external to a *simulated* machine, they do not contribute overhead to its execution. For this reason we do not experimentally evaluate their overhead. Intuitively the overheads should be very small. One hash table lookup is added to two operations. The first is when a new context ID is written. This happens during context switch, which is already a fairly heavyweight action. The second is context demap. Context demaps most often occur during process creation and exit, which are also heavyweight and relatively infrequent operations.

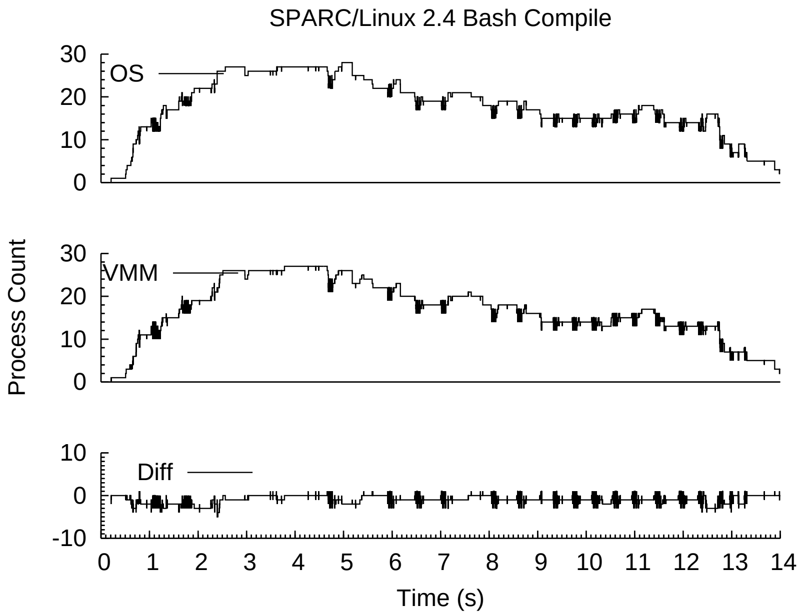


Figure 4.5: **Compilation Workload Timeline.** *SPARC/Linux compilation timeline. Compare to Figure 4.3.*

4.5.4 Association Evaluation

In this section we evaluate our association techniques. Our analysis focuses on I/O association, as we have found that to be the most challenging association to maintain, due to its asynchrony. We focus on accuracy, time overhead, and space overhead.

Accuracy

To measure the accuracy of I/O associations within the VMM, we instrumented the Linux guest operating system to trace the individual requests issued from each process address space. These traces were compared to corresponding traces of inferred VMM associations to calculate the accuracy of the mechanisms.

To stress the I/O association ability of the VMM, we increase the load on the system and plot the resulting I/O association accuracy. Figure 4.7 shows our results.

For low levels of concurrency, the simple context association method achieves a high degree of accuracy. With a large number of process groups, however, the accuracy of the context method declines dramatically; increased queuing delays between an I/O being issued and its observation by the VMM cause a majority of the requests to be incorrectly associated with other (CPU-bound) processes.

Event chaining, on the other hand, is able to achieve nearly perfect accuracy regardless of the level of concurrency. Hence it is a robust technique for associating asynchronous I/O

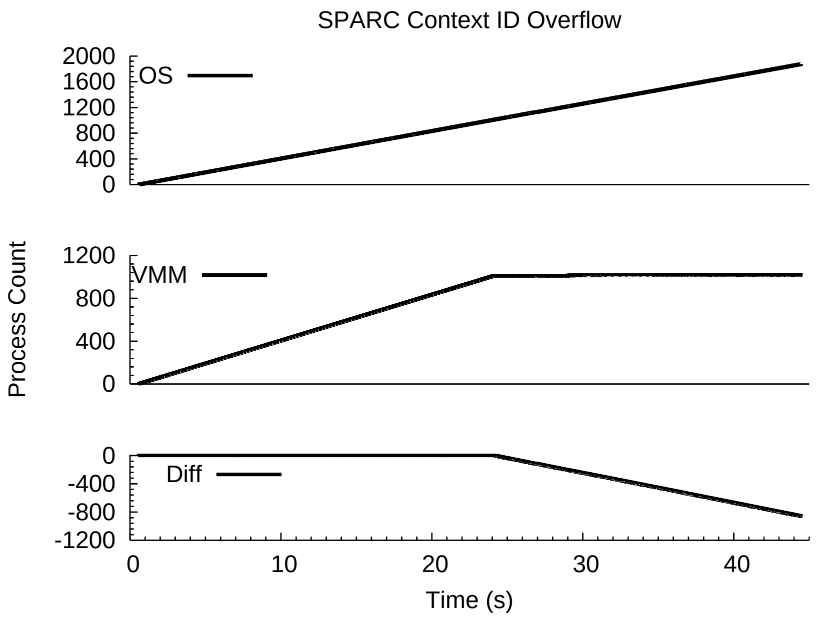


Figure 4.6: **Context ID Overflow.** *When more processes exist than can be represented by the available SPARC context IDs our techniques fail to detect context ID reuse.*

events with the issuing process.

Time Overhead

To measure the overhead imposed by I/O association, we measure the runtime and throughput achieved by an I/O bound process when executed with event-chaining association enabled and again when executed on an unmodified Xen/Linux system. The test program sequentially reads 200 MB of data with minimal think time between read requests. Each experiment was repeated five times and the results averaged. Since all association activity is initiated by I/O, an I/O-bound workload represents a worst case performance scenario for the technique.

No significant difference in runtime or throughput between the experiments was detected. Low overhead is expected because copy-based event chaining adds only a small number of minor page faults to each heavyweight I/O operation.

Space Overhead

The storage requirements for the reverse map are comparable to the storage required for the forward mapping, *i.e.*, the system's page tables; hence there is a noticeable space overhead (*e.g.*, roughly 12 bytes per active mapping). However, these data structures need only

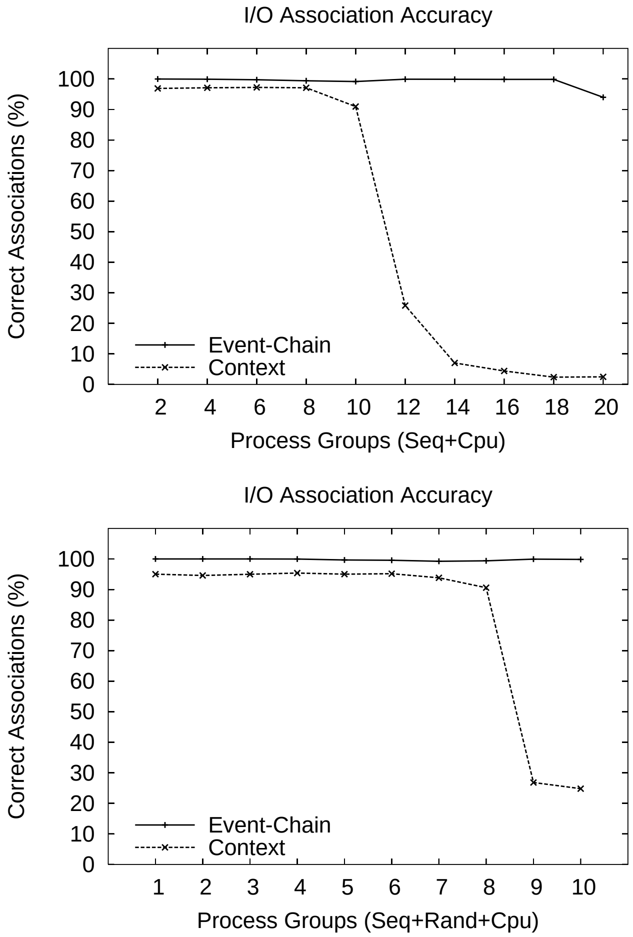


Figure 4.7: **I/O Association Accuracy.** *The figure plots the percentage of I/O requests that were correctly associated with their process on the y-axis as the number of process groups is increased along the x-axis. In the top graph, each process group consists of one process issuing sequential I/Os and one compute-bound process. In the bottom graph, we add one process per group that issues random I/Os. Two lines are plotted: simple context-based association and event-chain association.*

be maintained for processes that are actively running and generating enough I/O to be of interest, likely lowering the space overhead quite substantially in practice.

4.5.5 Evaluation Summary

Our evaluation shows that a VMM can infer process creation, exit and context switch using simple observations of guest OS MMU operations in three different environments (x86/Linux, x86/Windows, and SPARC/Linux). On x86 under Windows and Linux 2.4, Antfarm precisely identifies the desired process events. As one might expect for any inference technique the accuracy is not always perfect. Under x86/Linux 2.6 and under SPARC/Linux some false positives occur. However, the false positives are stylized and affect the ability of Antfarm to keep an accurate process count very little.

Context association is a simple and generic technique for associating any event observable by a VMM with a process. It is accurate for synchronous events where the actual occurrence of an event is not separated from the VMM's detection of the event in time. Event chaining enhances the accuracy of context association by tracking chains of related events until one of them is known to occur in the context of the process of interest. Event chaining incurs delay while the consumer of the event waits for the chain being used to resolve itself. We found that event chaining is especially useful for enhancing the accuracy of I/O associations as their processing inside the operating system is highly asynchronous.

4.6 Case Study: Anticipatory Scheduling

The order in which disk requests are serviced can make a huge difference to disk I/O performance. If requests to adjacent locations on disk are serviced consecutively, the time spent moving the disk head unproductively is minimized. Avoiding unnecessary seeks is the primary performance strategy of most disk scheduling algorithms. This case study explores the application of one innovative scheduling algorithm called *anticipatory scheduling* [47] in a virtual machine environment. The implementation makes use of Antfarm for Xen.

4.6.1 Background

Iyer *et al.*[47] have demonstrated a phenomenon they call *deceptive idleness* for disk access patterns generated by competing processes performing synchronous, sequential reads. Deceptive idleness leads to excessive seeking between locations on disk. Their solution, called anticipatory scheduling, introduces a small amount of waiting time between the completion of one request and the initiation of the next if the process whose disk request just completed is likely to issue another request for a nearby location. This strategy leads to substantial seek savings and throughput gains for concurrent disk access streams that each exhibit spatial locality.

Anticipatory scheduling makes use of process-specific information. It decides whether to wait for a process to issue a new read request and how long to wait based on statistics the disk scheduler keeps for all processes about their recent disk accesses. For example, the average distance from one request to the next is stored as an estimate of how far away the process's next access will be. If this distance is large, there is little sense waiting for the process to issue a request nearby. Statistics about how long a process waits after one request completes before it issues another are also kept in order to determine how long it make sense to wait for the next request to be issued.

Anticipatory scheduling does not work well in a virtual machine environment. System-wide information about disk requests is required to estimate where the disk head is located, which is essential in deciding if a request is nearby. Information about individual process's I/O behavior is required to determine whether and how long to wait. This information is not completely available to either a single guest, which only knows about its own requests, or to the VMM, which cannot distinguish between guest-level processes. While guests and the VMM could cooperate to implement anticipatory scheduling, this requires the introduction of additional, specialized VMM-to-guest interfaces. New interfaces may not be possible in the case of legacy or binary-only components. In any case, such interfaces do not exist today.

4.6.2 Information

To implement anticipatory scheduling effectively in a VMM, the VMM must be able to distinguish between guest processes. Additionally, it must be able to associate disk read requests with specific guest processes. Given those two pieces of information, a VMM implementation of anticipatory scheduling can maintain average seek distance and inter-request waiting time for processes across all guests. We use Antfarm to inform an implementation of anticipatory scheduling inside of Xen.

To associate disk read requests to processes, we employ a simple *context association* strategy that associates a read request with whatever process is currently active. This simple strategy does not take potential asynchrony within the operating system into account. For example, due to request queuing inside the OS, a read may be issued to the VMM after the process in which it originated has blocked and context switched off the processor. This leads to association error. The relatively low concurrency generated by the experiments in this section do not merit the more complicated event-chaining techniques described in Section 4.3.

4.6.3 Implementation

Xen implements I/O using device driver virtual machines (DDVM) [31]. A DDVM is a virtual machine that is allowed unrestricted access to one or more physical devices. DDVMs are logically part of the Xen VMM. Operationally, guests running in normal virtual machines make disk requests to a DDVM via an idealized disk device interface and the DDVM carries out the I/O on their behalf. In current versions of Xen, these driver VMs run Linux to take advantage of the broad device support it offers. A device back-end in the driver VM services requests submitted by an instance of a front-end driver located in all normal VMs.

The standard Linux kernel includes an implementation of anticipatory scheduling. We implement anticipatory scheduling at the VMM layer by enabling the Linux anticipatory scheduler within a Xen DDVM that manages a disk drive. To make this existing implementation process-aware, we introduce a foreign process abstraction that represents processes running in other VMs. When a disk request arrives from a foreign virtual machine, the Xen back-end queries our process-aware Xen hypervisor about which process is currently active in the foreign virtual machine. Given the ability to distinguish between processes we expect that our VMM-level anticipatory scheduler (VMAS) will improve synchronous read performance for competing processes whether they exist in the same or different VMs.

4.6.4 Evaluation

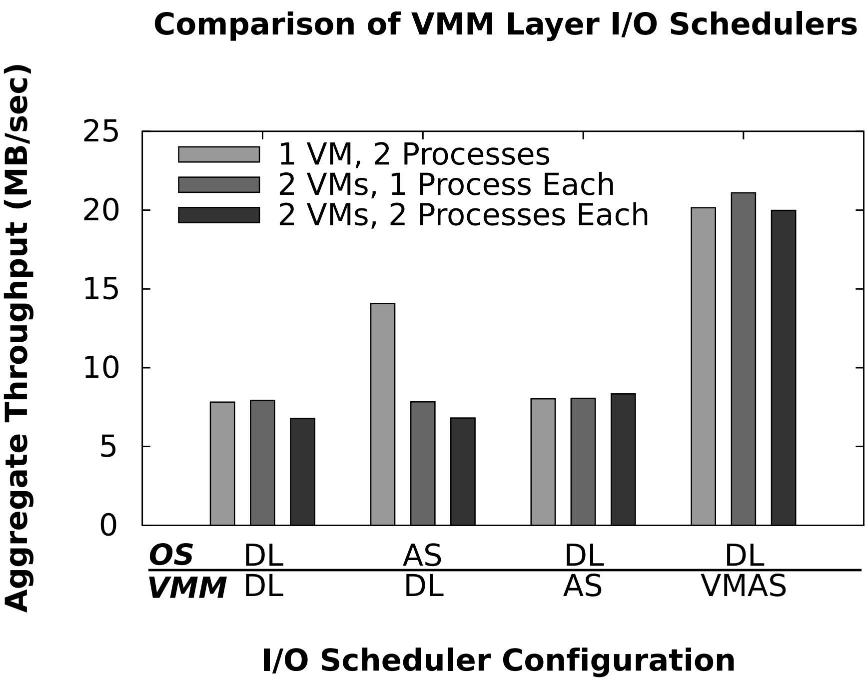
To demonstrate the effectiveness of our implementation of VMAS, we repeat one of the experiments from the original anticipatory scheduling paper in a virtual machine environment. Our experiment consists of running multiple instances of a program that sequentially reads a 200 MB segment of a private 1 GB file. We vary the number of processes, the assignment of processes to virtual machines, and the disk scheduler used by guests and by the VMM to explore how process awareness influences the effectiveness of anticipatory scheduling in a VMM. We make use of the Linux deadline I/O scheduler as our non-anticipatory baseline. Results for each of four scheduler configurations combined with three workloads are shown in Figure 4.8. The workloads are: (1) one virtual machine with two processes, (2) two virtual machines with one process each, and (3) two virtual machines with two processes each.

The first experiment shows the results from a configuration without anticipatory scheduling. It demonstrates the expected performance when anticipation is not in use for each of the three workloads. On our test system this results in an aggregate throughput of about 8 MB/sec.

The second configuration enables anticipatory scheduling in the guest while the deadline scheduler is used by Xen. In the one virtual machine/two process case, where the guest has complete information about all processes actively reading the disk, we expect that an anticipatory scheduler at the guest level will be effective. The figure shows that this is in fact the case. Anticipatory scheduling is able to improve aggregate throughput by 75% from about 8 MB/sec to about 14 MB/sec. In the other cases, guest-level anticipatory scheduling performs about as well as the deadline scheduler due to its lack of information about processes in other virtual machines.

Our third experiment demonstrates the performance of unmodified anticipatory scheduling at the VMM layer. Similar to the case of anticipatory scheduling running at the guest layer we would expect performance improvement for the two-virtual-machine/one-process-each case to be good because a VMM can distinguish between virtual machines just as an operating system can distinguish between processes. The improvement does not occur, however, because of an implementation detail of the Xen DDVM back-end driver. The back-end services all foreign requests in the context of a single dedicated task so the anticipatory scheduler interprets the presented I/O stream as a single process making alternating requests to different parts of the disk. The performance is comparable to the configuration without anticipation for all workloads.

The final configuration shows the benefit of process awareness to anticipatory schedul-



KEY
DL = *Linux Deadline Scheduler*
AS = *Linux Anticipatory Scheduler*
VMAS = *VMM-layer, Process-aware Anticipatory Scheduler*

Figure 4.8: **Benefit of process awareness for anticipatory scheduling.** *The graph shows the aggregate throughput for various configurations of I/O scheduler, number of virtual machines and number of processes per virtual machine. The experiment uses the Linux deadline scheduler (DL), the standard anticipatory scheduler (AS), and our VMM-level anticipatory scheduler (VMAS). Adding process awareness enables VMAS to achieve single process sequential read performance in aggregate among competing sequential streams. AS running at the guest layer is somewhat effective in the 1 VM / 2 process case since it has global disk request information.*

ing implemented at the VMM layer. In each of the workload configurations anticipatory scheduling works well, improving aggregate throughput by more than a factor of two, from about 8 MB/sec to about 20 MB/sec. Because it is implemented at the VMM layer, anticipatory scheduling in this configuration has complete information about all requests reaching the disk. Our process awareness extensions allow it to track statistics for each individual process enabling it to make effective anticipation decisions.

4.7 Assumptions

As is the case for any inference technique, Antfarm requires that certain assumptions hold to produce correct results. This section lists and discusses the assumptions Antfarm makes about the guest operating systems it observes. There are relatively few assumptions and we believe they hold for nearly all widely available operating systems in common use on workstation and server class computing systems today.

Processes: Antfarm assumes that an operating system uses heavy-weight processes to define the basic execution environment of all user-level programs including an address space and I/O environment.

Hardware memory protection: Antfarm assumes that an operating system employs hardware memory protection to implement isolated process address spaces. While nearly all current operating systems take this approach, other options exist. The recent Singularity [43] research operating system uses programming language techniques like type checking to ensure that processes do not interfere with each other even when they share a single hardware address space.

One address space per process: Antfarm uses address space events as a proxy for process events. This implies a one-to-one correspondence between address spaces and processes. This is typically true for most operating systems, but we observed a subtle violation of this assumptions by the implementation of `exec` under Linux 2.6. When `exec` is invoked a new address space is created so a process that is created via `fork` followed by `exec` effectively uses two address spaces for a single logical process. This pattern is highly stylized. The lifetime of the initial address space is nearly always tiny compared to that of the second. For the applications of process information we have developed this violation had no practical effect.

ASIDs are not multiplexed among active processes: Antfarm assumes that the value it uses as an address space identifier (ASID) is used by a single process while that process is active. On SPARC we observed that, due to the limited space of our chosen ASID (the SPARC context ID), these values are subject to reuse when a large number of processes (more than 8192) exist concurrently.

Address space data structures cleared before reuse: Antfarm reports process exit when the data structures used to represent a process address space have been cleared and are ready to be reused. On x86 this corresponds to clearing of page tables in memory. On SPARC it corresponds to a context demap operation. This requirement is derived from the basic operating system responsibility to maintain memory isolation between processes.

Address space data structure clearing is timely: Antfarm assumes that an operating system clears address space data structures in a timely manner. While an OS could arbitrarily delay this operation for selected processes, in practice we have found that the operating systems we have tested, including Linux and the Windows NT family, eagerly reclaim these resources and the lag between actual process exit and data structure clearing is small.

4.8 Summary

In this chapter, we have described and evaluated techniques that allow a VMM to independently discover information about processes for the Windows and Linux operating systems and for the x86 and SPARC architectures. To do so, we have exploited the correspondence between processes and address spaces and the ability of the VMM to observe events like privileged register updates, TLB flushes, and page table updates. Process creation and context switch events can be deduced by simply tracking the use of an address space-specific value like the physical address of the page directory or the SPARC context ID. We detect process exit by tracking the status of memory management structures like the page tables and noting when such resources can be safely reused.

The accuracy achieved by Antfarm is excellent. All true process events are detected without error. For certain versions of Linux, matching pairs of spurious events are detected because the "one address space per logical process" model does not hold. Because of the self-correcting nature and the very brief lifetime of these errors, they have little effect on the ability of Antfarm to track the true current process list.

Antfarm is careful to avoid interposing on high-frequency, critical-path operations; hence, it imposes very little overhead. In our experiments a worst case performance scenario results in a small 2.4% slowdown. Less pathological, but still demanding, workloads impose only a tiny 0.6% overhead.

We used an I/O scheduling case study to demonstrate that process information can be utilized by a VMM to transparently improve overall system performance. By taking process-specific I/O patterns into account, our VMM-layer anticipatory scheduler is able to increase throughput for competing sequential streams, even from different virtual machines, by a factor of more than two.

Chapter 5

Monitoring the Guest Buffer Cache

In this chapter we describe a set of techniques that can be used by a VMM to infer information about a critically important OS sub-system, the unified buffer cache and virtual memory system. The buffer cache's job is deceptively straightforward. It simply caches recently accessed blocks from disk. However, deciding which blocks to cache and for how long involves a subtle trade-off between memory space and performance that depends on workload, cache size, and user preference. The buffer cache is often the largest consumer of memory in a modern system; hence, the memory used by the buffer cache must be carefully balanced with the memory needs of other user processes. Since, disk accesses typically exhibit spatial and temporal locality, a well-managed buffer cache can have a huge impact on overall system performance by transforming glacially slow disk accesses into relatively fast memory references.

A VMM is intimately involved in allocating and managing the memory resources in a virtualized environment. We show in this chapter that a VMM can carefully observe guest operating system interactions with virtual hardware like the MMU and storage devices to detect when pages are inserted into or evicted from the operating system buffer cache. Such information can then be used to more effectively manage local and remote disk caching resources.

Geiger is an implementation of these techniques within the Xen virtual machine monitor [27]. In this chapter, we discuss the details of Geiger's implementation and perform a careful evaluation of Geiger's eviction detection techniques. A few of Geiger's inferencing techniques within the VMM are similar to those used by Chen *et al.* within a pseudo-device driver [17]. Hence, our evaluation focuses on which of Geiger's new techniques are needed in different circumstances. First, we show that the unified buffer caches and virtual memory systems found in modern operating systems require the VMM to track not only disk traffic, but memory allocations as well. Second, we show that a VMM must take basic storage system behavior into account to accurately detect cache eviction. For example, the VMM must track whether a particular data block is live or dead on disk in order to avoid reporting many spurious evictions. We also show that journaling file systems, such as ext3 in Linux, require the VMM to distinguish between writes to the journal and writes to other parts of storage to avoid an aliasing problem that leads to false eviction reporting. In summary, passively detecting cache events within modern operating systems requires new sophistication. Without these techniques, passive inferencing can result in incomplete information which can be worse than no information at all.

Via case studies, we demonstrate how the inferred eviction information provided by Geiger can enable useful services inside a VMM. In the first case study we implement a novel, VMM-based working set size estimator that complements existing techniques [101] by allowing estimation in the case that a virtual machine is thrashing. A second study explores how Geiger-inferred evictions can be used by a VMM to enable remote storage caches to implement eviction-based cache placement [104] without changing the application or operating system storage interface. Using existing interfaces increases the probability that such a feature is adopted in practice.

5.1 Geiger Techniques

We will begin our discussion of Geiger by describing the basic techniques Geiger uses to infer page cache promotion and eviction. We then describe how Geiger performs more complex inferences, in particular, how it handles unified buffer caches and virtual memory systems that are present in all modern operating systems, and how it handles issues that arise due to storage system interactions.

5.1.1 Basic Techniques

Buffer cache *promotion* occurs when a disk page is added to the cache. Buffer cache *eviction* occurs when a cache page is freed by the operating system and its previous contents remain available to be reloaded from disk. For example, an eviction occurs if the contents of an anonymous page are written to a swap partition and then the page is freed. Similarly, an eviction occurs if a page that was read from the file system is later freed without writing anything back to disk, since the data can be reloaded from the original location on disk. However, an eviction does not occur if the OS frees a page and its contents are lost (*e.g.*, an anonymous page when its associated process exits).

To detect promotion and eviction, Geiger performs two tasks. First, Geiger tracks whether the contents of a page are available on disk and, if so, where on disk the contents are stored. We call the on-disk location associated with a memory page the page's *Associated Disk Location* (ADL). Second, Geiger must detect when a page is freed by the OS. We describe each of these steps in turn.

Associated Disk Locations

Geiger associates a disk location with each physical memory page, whenever appropriate. An associated disk location (ADL) is simply the pair `<device, block offset>`, representing the most recent disk location with which a VMM can associate the page. A VMM associates a disk location with a memory page whenever that page is involved in a disk read or write operation. For example, if a page is the target of a read from disk location A, the page becomes associated with A. Similarly, if a page is the source of a write to disk location B the page becomes associated with B. These associations persist until replaced by another association, the memory page is freed, or the relevant disk blocks are freed.

Since the VMM virtualizes all disk I/O, disk reads and writes initiated by a guest are explicitly visible to the VMM and no special action on the part of the VMM is required to establish the ADL of a page. However, to correctly invalidate an ADL when the disk block, to which it refers, is no longer in use requires detecting when the disk block is freed. We discuss this further in Section 5.1.3.

Detecting Page Reuse

Geiger must also determine when a memory page has been freed by the OS. However, the guest OS does not explicitly notify the VMM when it frees a page. Often the only difference between an active and a free page is an entry in a private OS data structure, such as a free list or bitmap. We assume that the VMM does not have the detailed, OS-specific information required to locate or interpret these data structures. Hence, instead of detecting that a page has been freed, Geiger detects that a page has been *reused*. Since reuse implies that a page was freed between uses, it is an appropriate proxy for the page free event.

Geiger uses numerous heuristics to detect that a page has been reused. Each heuristic corresponds to a different scenario in which a guest OS allocates a page of memory. If Geiger detects a page allocation and the newly allocated page has a current ADL, then Geiger signals that the previous contents of the page, as defined by the ADL, have been evicted.

The two most basic techniques used by Geiger are monitoring disk reads and disk writes. This builds on the previous work of Chen *et al.* [17] which monitors reads and writes in a device driver within an OS.

Disk Read: Geiger uses disk reads to infer that a new page may have been allocated. When a page is read from disk, a new page is allocated in the OS buffer cache. If the allocated page has a current ADL that refers to a different disk location than the one currently being read, Geiger reports that the page's previous contents have been evicted. The ADL of the affected page is updated to point to the new disk location as a consequence of this kind of eviction.

Disk Write: Geiger uses disk writes to infer that a new page may have been allocated. If a full page of data is written to disk and the page does not already reside in the page cache, then the OS may allocate a new page to buffer the data until it is asynchronously written to disk. Geiger detects this case by observing all disk writes and signaling an eviction if the write source is a page with a current ADL that is different than the target disk location of the write. Note that if a previous read or write caused the disk block to already exist in the cache, Geiger will not erroneously signal a duplicate eviction since the page's ADL will not change. As with the read-eviction heuristic, the ADL of the affected page is updated to refer to the target disk location.

5.1.2 Techniques for Unified Caches

Techniques from previous research [17] work well with old-style file system buffer caches, which were kept distinct from the virtual memory system. However, virtually all modern operating systems, including Linux, *BSD, Solaris, and Windows, have a unified buffer cache and virtual memory system. Unification complicates inferences: Geiger must be able to detect page reuse for additional cases associated with the virtual memory system. Hence, we introduce two new detection techniques.

Copy On Write: Copy-on-write (COW) is a technique widely used in operating systems to implement efficient read sharing of memory. A page shared using COW is marked read-only in each process's virtual address space that shares it. When one of these processes attempts to write to a COW-shared page, the action causes a page fault. The operating system then transparently allocates a new, private page and copies the data from the old page into the new page. Subsequently, a new writable virtual memory mapping is established which refers to the new page. Because the private copy requires allocation of a free page, it can lead to page reuse.

Geiger detects page reuse that occurs as a result of COW by observing page faults and page table updates. When Geiger detects a page fault whose cause is a write into a read-only page, it saves the affected virtual address and page table entry in a small queue. If, a short time later, the guest OS creates a new writable mapping for the same virtual address, but a different physical page, Geiger infers that the new physical page was newly allocated. If the newly allocated page has an active ADL, then Geiger signals an eviction. This heuristic clears the ADL of the newly allocated page because it is a modified private copy of an existing page and is not associated with any disk location.

Allocation: Most modern operating systems allocate memory lazily. When an application requests memory (*e.g.*, using `brk` or an anonymous `mmap`), the OS does not immediately allocate physical memory; instead the virtual address range is "reserved" and physical memory is allocated on-demand when the page is actually accessed. This property means that physical memory allocation nearly always occurs in the context of servicing a page fault.

Similar to the COW heuristic, Geiger observes page faults that are due to a guest accessing a virtual page that has no virtual-to-physical mapping and saves the affected virtual address in a small queue. If, a short time later, the guest OS creates a new *writable* mapping for the faulting virtual address, Geiger infers a page allocation. If the newly allocated physical page has a current ADL, then Geiger signals an eviction.

The allocation eviction heuristic contains some simplifications that could lead to false positive inferences. First, the technique makes use of the fact that memory is rarely write-shared between address spaces. If a page is write-shared, however, the creation of a new writable mapping as described above does not imply a page allocation, but will be counted as such by our heuristic leading to false positives. Second, if a page belonging to an mmapped disk file is initially brought into the page cache via a write operation, the disk page will first be read from disk (potentially causing a read eviction) and a new writable mapping will be created (causing an allocation eviction). Hence, a single write could lead to two eviction reports, one of which is a false positive. The more common case of a shared, read-only mapping of a disk file is handled correctly, however, since the allocation heuristic ignores it and only a single read-eviction is generated when appropriate.

5.1.3 Techniques for Storage

Storage systems also introduce some nuances into the inferences made by Geiger. In particular, file system features like journaling lead to an *aliasing* problem; further, the fact that disk blocks can be deleted leads to the problem of *liveness detection*. We now describe these issues and how Geiger handles them in turn.

Journaling

The basic write heuristic signals an eviction whenever the contents of a page that has an ADL are written to a location on disk which does not match that ADL. For example if a page has ADL A and is written to disk location B an eviction will be reported for the contents of disk location A. The basic write heuristic over-reports evictions in cases where data are written from the same buffer cache page to multiple disk locations; we view this as an *aliasing* problem, as the same page is wrongly associated with two disk addresses.

Journaling file systems, such as Linux ext3 [96], ReiserFS [76], JFS [10], and XFS [94], routinely write to two locations on disk from the same cache page, namely the journal location and the fixed disk location. In the worst-case journaling scenario, where both data and metadata are first written to the journal, twice the actual number of write evictions will be reported. In the more common case of metadata-only journaling, a much smaller penalty is incurred.

The negative effect of journaling and virtual memory can be mitigated if the VMM identifies writes to the file system journal. This is straightforward in most systems, since the journal is either placed on a separate, easily identifiable partition or in a file within a file system partition to which a reference is made from the file system superblock [97]. Hence, to avoid the problem of journal aliasing, Geiger monitors the disk addresses of write requests and ignores writes directed to the journal.

Block Liveness

Geiger signals that a page has been evicted only if that page has a current ADL. It is possible that the blocks to which an ADL refers are deallocated on disk between the time that the ADL mapping is first established and when Geiger detects that the associated page has been reused. In this case, Geiger will falsely report an eviction, because an ADL exists but the data to which the ADL refers have been deallocated and are no longer accessible. This problem of *block liveness* can lead to large numbers of false evictions for workloads in which files are regularly deleted, truncated, or when processes die that have significant parts of their virtual memory swapped to disk.

File systems: A virtual machine monitor can passively track *file system* block liveness in the same way a smart disk system can track block liveness [85]. The allocation state for each file system block is typically noted in some on-disk structure like a bitmap. The file system superblock, which is stored at a known, fixed location on disk, can be used to locate these bitmap structures. By examining guest operating system writes to these on-disk areas, a VMM can snoop on the file system to determine when disk blocks to which an ADL refers have been freed. If the blocks to which an ADL refers are deallocated, the ADL must be invalidated so that a future reuse of the affected page is not misinterpreted as an eviction.

Implementing block liveness by observing only disk writes has one significant drawback; there is often substantial lag between when a file system structure like an allocation bitmap is updated in memory and when it is written to disk. In many operating systems this interval can be 30 seconds or more. If Geiger does not observe that the file system blocks, to which a page's ADL points, have been deallocated until after the page has been reused, a false eviction will occur. Hence, the timeliness of block deallocation notification is important.

A VMM can improve the timeliness of block deallocation notification by tracking updates to the in-memory versions of the allocation bitmaps. Given the known locations of the bitmaps, the VMM can observe when bitmaps are loaded from disk into memory. At that time, the VMM can mark all such buffers read-only. When a guest updates an in-memory bitmap, a minor page fault will occur. The VMM can observe that the fault is due to an attempted bitmap update and respond by invalidating affected ADLs.

Geiger implements this style of in-memory block liveness tracking. Bitmap blocks are identified by reading and parsing the file system superblock for known file system types. Pages used to cache file system allocation bitmaps are marked read-only in memory by Geiger. When a write to such a page is detected, due to a page protection fault, the effect of the faulting instruction is emulated on the guest memory and register state and the faulting instruction is skipped; hence, every bitmap update is synchronously observed and appropriate action is taken by the VMM. The overhead of block liveness tracking is kept low in spite of additional minor page faults due to the relatively low frequency of disk bitmap updates.

Like Sivathanu [86], we consider embedding file system layout information, such as the format of the superblock, within a VMM a reasonable technique. There are few commonly used file systems and the on-disk data structure formats for those file systems change slowly. A VMM can be provided with layout information for all commonly used file systems and the information can be expected to remain valid for a long time. The on-disk format of ext2, for example, has not changed since its introduction in 1994. This is a far longer interval than the typical system software upgrade cycle.

Swap space: The liveness tracking techniques Geiger uses for file system partitions do not apply to disk space used as a swap area. As a rule, swap space does not contain any on-disk data structures that track block allocation because data in swap is not expected to persist across system restarts. All swap allocation information is managed exclusively in volatile system memory. There are two swap liveness tracking techniques we have found to be effective for some workloads in preventing false evictions due to ADLs that point to deallocated swap space.

The first technique invalidates any ADL that points to a set of disk blocks that is overwritten. When disk blocks are overwritten, the data to which an ADL refers has been destroyed; hence, ADL invalidation is appropriate. This technique is implemented by maintaining a reverse mapping between cached disk blocks and ADLs.

The second technique makes use of implicitly obtained process lifetime information like that provided by Antfarm [49]. Given accurate information about guest OS processes and a mapping of memory pages to the owning OS process, many ADLs can be invalidated when the process exits. Specifically, an ADL from a page belonging to a dead process that points to a swap space disk block can be invalidated. This second technique appears promising but has not been fully implemented in the current version of Geiger.

5.2 Implementation

Geiger is implemented as an extension to the Xen virtual machine monitor version 2.0.7. Xen [27] is an open source virtual machine monitor for the Intel x86 architecture. Xen provides a paravirtualized [103] processor interface, which enables lower overhead virtualization at the expense of porting system software. We explicitly do *not* make use of this feature of Xen; hence, the mechanisms we describe are equally applicable to a more conventional virtual machine monitor such as VMWare [91, 101].

Geiger consists of a set of patches to the Xen hypervisor and Xen's block device backends. Changes are concentrated in the handlers for events like page faults, page table updates and block device reads and writes. The Geiger patches consist of approximately 700 lines of code across three files. About 25 other files from the Xen hypervisor and the Linux kernel required small changes in order to implement instrumentation and tracing.

All experiments described in this paper were performed on a PC with a 2.4 GHz Pentium IV processor, 2 GB of system memory, and two WD1200BB ATA disk drives. We used Linux kernel version 2.6.11 in the Xen control domain and Linux kernel version 2.4.30 for all unprivileged domains. We use either the ext2 or ext3 file system, depending upon the experiment. The Xen control domain is configured with 512 MB of memory. Unless otherwise noted, each unprivileged guest virtual machine is assigned 128 MB of memory.

5.3 Evaluation

In this section we evaluate the ability of Geiger to accurately infer page cache evictions and promotions occurring within guest operating systems. We begin by describing our workloads and metrics; we then evaluate Geiger using a set of four microbenchmarks and four application workloads. We conclude by measuring the overheads that Geiger imposes on the system.

Microbenchmark	Description
Read Evict	Sequentially reads a section of a file larger than available memory multiple times
Write Evict	Sequentially writes a file larger than available memory. Repeated multiple times.
COW Evict	Allocates a memory buffer approximately the size of available physical memory, then writes to each virtual page to ensure a physical page is allocated, then forks and writes to each page in the child.
Allocation Evict	Allocates a memory buffer that exceeds the size of available memory and writes to each virtual page to ensure a page is allocated.

Figure 5.1: **Microbenchmark Workloads.** *This table describes the four microbenchmarks used to isolate a specific type of page eviction.*

Application	Description
Dbench [95]	File system benchmark simulates load on a network file server
Mogrify [44]	Scales and converts a large bitmap image
OSDL-DBT1 [70]	TPC-W-like web commerce benchmark simulating web purchase transactions in an online store.
SPC Web Search 2 [90]	Storage performance council block device traces from a web search engine server. Traces are replayed to a real file system.

Figure 5.2: **Application Workloads.** *This table describes each of the four application workloads.*

5.3.1 Workloads

Throughout the experimental evaluation of Geiger, we use two sets of workloads. The first workload set consists of four microbenchmarks. Each of these four microbenchmarks have

Application	Read %	Write%	COW%	Alloc%
Dbench	41.13%	58.85%	0.00%	0.00%
Mogrify	53.22%	22.31%	0.01%	24.25%
OSDL-DBT1	77.02%	2.14%	0.54%	20.29%
SPC Web Search 2	99.6 %	0.03%	0.00%	0.00%

Figure 5.3: **Application Workload Eviction Mix.** *This table reports the percentage of total eviction events caused by each eviction type.*

been constructed to generate a specific type of page cache eviction: Read, Write, Copy-On-Write (COW), or Allocate. Thus, these microbenchmarks isolate Geiger's ability to track evictions due to specific events. The microbenchmarks are described in more detail in Figure 5.1.

The second set of workloads consists of four application benchmarks. These represent more realistic workloads. Each workload contains a mix of eviction types, whether read, write, COW, or allocation. Figure 5.2 describes the application workloads. Figure 5.3 shows the breakdown of eviction types generated by each application workload. Hence, these application workloads stress Geiger's ability to track evictions that may occur for several different reasons.

5.3.2 Metrics

Our methodology for evaluating the accuracy of Geiger is to compare the trace of evictions signaled by Geiger to a trace of evictions produced by the guest operating system; we have modified the Linux kernel to generate this trace. Since the guest operating system has complete information about which pages are evicted and when, our comparison is against the ideal eviction detector. The eviction records in both traces contain the physical memory address, the disk address of the evicted data, and a time stamp.

We consider three different metrics for accuracy. The first metric is simply the *eviction count* reported by Geiger compared to that reported by the guest OS over time. The second metric is *detection lag*, or the time between when a particular eviction takes place in the OS and when it is detected by Geiger. Finally, the third metric is the *detection accuracy*, which tracks the percentage of records from the inferred and actual traces that match in a one-to-one mapping; we report both the percentage of false negatives (*i.e.*, actual evictions not detected by Geiger) and false positives (*i.e.*, inferred evictions that did not correspond to OS-reported evictions).

5.3.3 Microbenchmarks

We begin by running workloads consisting of the four microbenchmarks. Figure 5.4 shows the resulting eviction count time-lines. For all microbenchmarks, the eviction counts inferred by Geiger closely match the actual OS counts; however, depending upon the work-

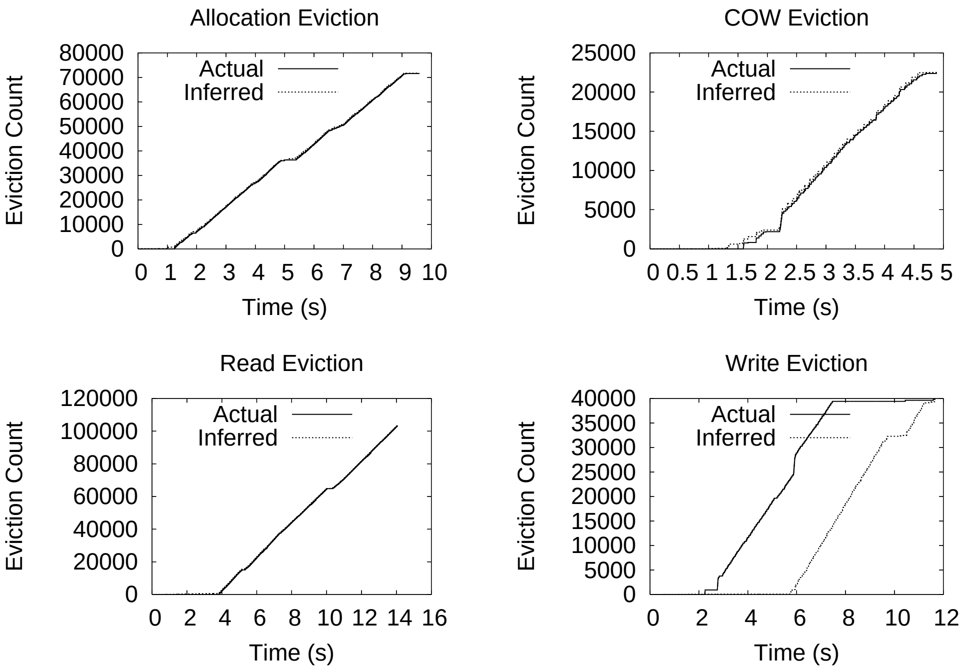


Figure 5.4: **Eviction Inference Counts.** *The figure compares inferred vs. actual eviction counts over time for microbenchmarks that isolate each eviction type inferred by Geiger.*

load, some interesting differences may occur along the way. For example, during the COW workload, the guest OS reclaims pages in groups, leading to a slight stair-step eviction pattern; Geiger's inferences lag slightly behind in this case. In the write workload, the guest OS begins evicting pages early and continues to evict eagerly throughout the experiment. Because the pages being evicted are dirty, they must be written to disk before they are freed which significantly delays their reuse. Geiger's inferences are based on page reuse; hence, eviction is not detected until a page is reused, and inferred evictions lag noticeably behind actual evictions when caused primarily by writes.

Figure 5.5 shows the cumulative distributions of eviction lag times for each of the microbenchmarks. As expected, the lag times for read, COW, and allocation eviction are concentrated at very small values. However, the lag times for the write microbenchmark are concentrated at about three seconds due to the glut of disk writes caused by dirty pages being evicted.

Figure 5.6 reports Geiger's detection accuracy in both false negatives and false positives. For all workloads, false negatives are uncommon: at worst, fewer than 2.5% of the total number of evictions are missed by Geiger. False positives are even less common: at worst, Geiger over-reports 1.45% of its inferred evictions.

In our final microbenchmark experiment, we explore Geiger's ability to detect aliased

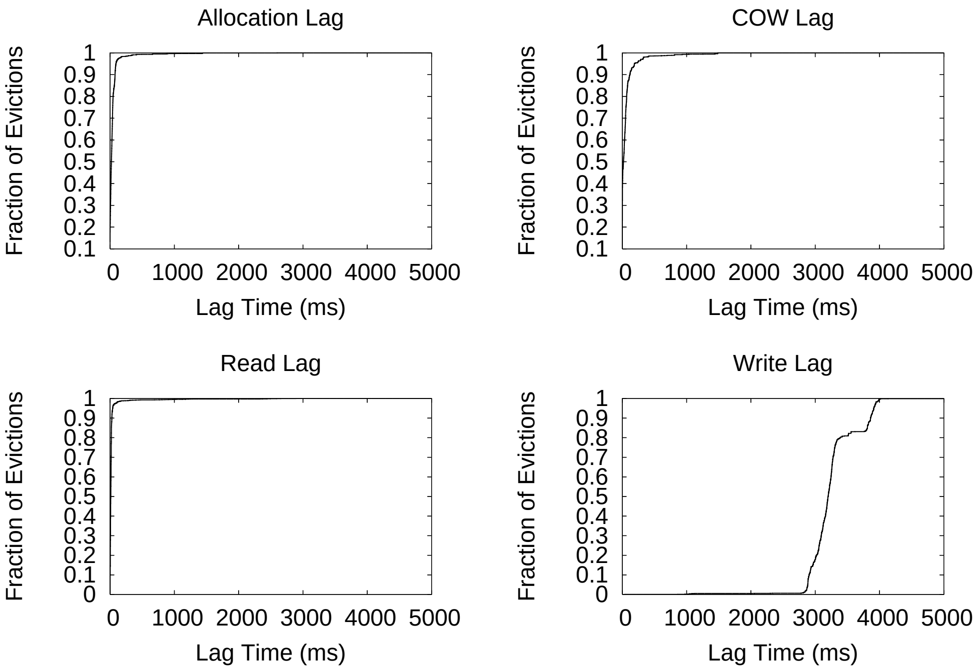


Figure 5.5: **Eviction Lag.** *The figure shows the cumulative lag distribution for microbenchmarks that isolate each eviction type.*

writes to the file system journal. We use the write workload to stress this detection. Figure 5.7 shows the accuracy of Geiger with and without the specialization to disregard write traffic to the file system journal. Without this specialization, Geiger performs satisfactorily when journaling is disabled or when only metadata is journaled (*i.e.*, Linux ext3 ordered-mode); with metadata journaling, relatively few blocks have aliases. However, with data journaling, many blocks have aliases and, as a result, more than half of the evictions reported by the un-specialized Geiger are false positives. In contrast, the full version of Geiger accurately handles all journaling modes of Linux ext3; even with data journaling, Geiger has a false positive percentage of only 0.06%.

5.3.4 Application Benchmarks

We next consider workloads containing more realistic applications. Figure 5.8 reports the detection accuracy of Geiger on these application workloads. For all workloads, false negative ratios are small: in the worst case, Geiger misses only 2.24% of the evictions reported by the OS. However, the Dbench and Mogrify workloads have interesting behavior regarding false positives.

Workload	False Neg %	False Pos %
Read Evict	0.96%	0.58%
Write Evict	1.68%	0.03%
COW Evict	2.47%	1.45%
Alloc Evict	0.17%	0.17%

Figure 5.6: **Microbenchmark Heuristic Accuracy.** *The table reports the false positive and false negative ratios for the complete set of eviction heuristics for each of the microbenchmark workloads.*

Workload	w/o Journal Opt		w/ Journal Opt	
	F. Neg %	F. Pos %	F. Neg %	F. Pos %
No Journal	1.68%	0.03%	1.68%	0.03%
Metadata	1.83%	0.33%	0.61%	0.08%
Data	1.43%	61.91%	2.51%	0.06%

Figure 5.7: **Effect of Journaling.** *The table reports the false positive and false negative ratios for the write-eviction microbenchmark workload when run with no journaling, with metadata journaling (ordered mode), and data journaling with the Linux ext3 file system. The table shows the benefits of turning on the Geiger specialization to detect writes to the journal.*

Block Liveness

The Dbench and Mogrify workloads illustrate the benefit of having Geiger attempt to track the liveness of each block on disk. Dbench creates and deletes many files; as a result, many pages in memory are reused for different files (and different disk blocks). Mogrify causes large amounts of swap to be allocated and deallocated during its execution. If the VMM uses only the change in association between a memory page and its disk block to infer an eviction, then the VMM concludes that many evictions have occurred that actually have not (*i.e.*, many false positives). Thus, without live block detection, Geiger has a 30.2% false positive rate for Dbench and a 23% false positive rate for Mogrify. However, when Geiger tracks whether a particular disk block is free, it can detect when a page is simply reused without the previous contents being evicted; as a result, the false positive rate improves dramatically to 5.7% for Dbench and 2.46% for Mogrify. Thus, to adequately handle delete-intensive (or truncate-intensive) workloads, Geiger includes techniques to track disk block liveness.

Limitations

As mentioned previously, we do not expect our current techniques for tracking block liveness in swap space to be adequate in all situations. To demonstrate this remaining problem, a microbenchmark was crafted that results in large numbers of false positives despite the best efforts of Geiger to track block liveness. The program forces a large buffer (allocated using `mmap`) to be swapped to disk and then the buffer is released. In Linux, as the buffer

Workload	Geiger Opts	False Neg %	False Pos %
Dbench	w/o block liveness	1.10%	30.23%
Dbench	w/ block liveness	2.30%	5.72%
Mogrify	w/o block liveness	0.05%	22.99%
Mogrify	w/ block liveness	0.65%	2.46%
TPC-W		0.14%	3.12%
SPC Web 2		2.24%	0.32%

Figure 5.8: **Application Heuristic Accuracy.** *The table reports the false positive and false negative ratios for Geiger on the four application workloads. For the Dbench and Mogrify workloads, we evaluate Geiger both without and with the optimizations to detect whether a block is live on disk.*

is released, the associated swap space is also deallocated, but Geiger does not detect that event. As additional memory is allocated by the program, pages are reused whose ADLs point to deallocated swap space resulting in an eviction false positive ratio of about 37%.

5.3.5 Overhead

Geiger observes events that are intrinsically visible to a VMM like page faults, page table updates, and disk I/O. Except in the case of disk block liveness tracking, no additional memory protection traps or I/O requests are caused by Geiger. Liveness tracking imposes one additional minor page fault for each disk bitmap update which occur relatively rarely. Hence, we expect the runtime overhead imposed by Geiger to be small. To validate this expectation, we compare the runtime of workloads running on an unmodified version of Xen to that of Geiger. We are interested in two performance regimes. The first regime is the more common case, in which a workload has sufficient memory and few evictions occur. The second regime occurs when a machine is thrashing, since this implies that many evictions are taking place and Geiger's inference mechanisms are being stressed.

We evaluate each of these four cases using two carefully chosen workloads. Since Geiger interposes on code paths for handling page faults, page table updates and disk I/O, we use the microbenchmark "allocation-evict" described in Figure 5.1 and Dbench described in Figure 5.2. Allocation-evict causes many page faults and page-table updates stressing that portion of Geiger's inference machinery. Dbench causes a large number of file creations, reads, writes, and deletes which exercise those portions of Geiger's heuristics.

Figure 5.9 shows the results of the experiment. Each value shown is the average of five runs; the standard deviation is shown with error bars. The largest observed overhead is 2.19%, which occurs for a thrashing Dbench. For all cases, the results for Geiger and the unmodified Xen are comparable.

Geiger requires some extra space per physical memory page to track ADLs. In our prototype this amounts to 20 bytes per memory page. In our test system, configured with 2 GB of physical memory, a total of 10 MB of additional memory is allocated by the VMM, leading to a space overhead of approximately 0.5%. If this space overhead is a concern, it

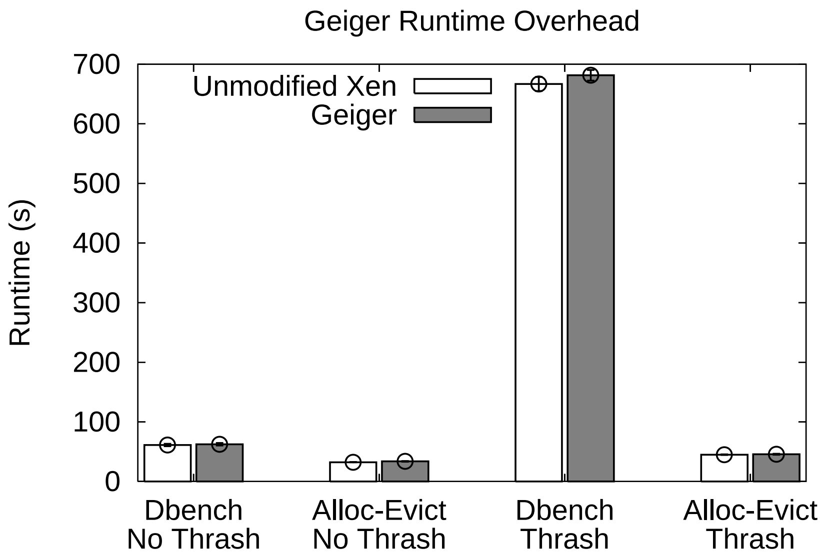


Figure 5.9: **Geiger Runtime Overhead.** *The figure shows that Geiger imposes very small runtime overheads for two workloads that stress its inference heuristics.*

could be substantially reduced, given the preallocated, fixed size, and sparsely-populated data structures of our prototype.

5.4 Case study: Working Set Size Estimation

Geiger's eviction detection techniques are useful for implementing a number of pieces of functionality. In our first case study, we show how Geiger can be used to implement MemRx, a VMM service that tracks the working sets of guest VMs. We begin by describing the implementation of MemRx and then present performance results.

5.4.1 MemRx Design

Previous research by Waldspurger [101] for ESX Server has shown how a VMM can determine the system working set size of a VM whose memory footprint fits in physical memory. MemRx complements the ESX Server technique by enabling a VMM to determine the working set size for a *thrashing* virtual machine.

MemRx does this by simulating the buffer cache behavior of the guest operating system as if more memory were allocated to it. Geiger allows MemRx to monitor buffer cache evictions and promotions. Figure 5.10 shows a schematic of the page cache simulation implemented by MemRx. Using the ADL mechanism, Geiger knows which blocks on

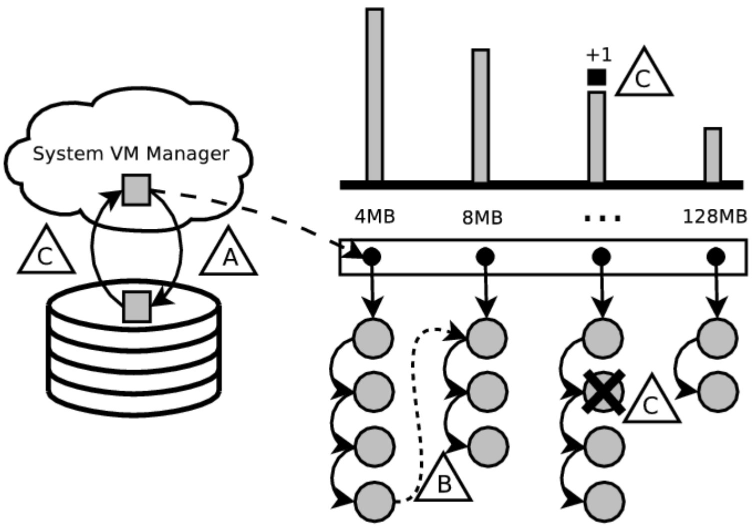


Figure 5.10: **MemRx Operation.** *The figure shows a schematic of the cache simulation implemented by MemRx. A) When a page is evicted by a guest, this event is detected by MemRx and an entry is added to the head of a series of queues. B) If necessary, queue entries ripple from the tail of one queue to the head of the next. C) Upon reload, the associated queue entry is removed and an array entry associated with that queue is incremented. Each entry tracks which sub-queue it appears in to enable fast depth estimation.*

disk correspond to an evicted page. When a page is evicted, a reference to the page's location on disk is inserted at the head of a queue maintained in LRU order by MemRx. Subsequent evictions push previous references deeper in the queue. When a previously evicted page is read from disk, *i.e.*, promoted into the page cache, the reference to that page is removed from the queue and its distance D from the head of the queue is computed. The distance is approximately equal to the number of evictions that have taken place between that page's eviction and its subsequent reload. MemRx then uses D to compute the amount of memory that would have been required to prevent the original evictions from taking place as $size_{page} \times (D+1)$. This information is used to compute a miss-ratio curve [107]. The working set size can be read from the miss-ratio curve by locating the curve's primary knee.

For example, if a page is evicted and immediately reloaded before any other pages are evicted, MemRx would record that the eviction could have been prevented by one additional page of physical memory. If a page's eviction is followed by 1024 evictions of 4 KB pages, MemRx would report that $(1024 + 1) \times 4 \quad KB$ (roughly 4 MB) of additional memory would be required to prevent the original eviction.

Our general strategy, which is similar to Patterson *et al.*'s ghost buffering scheme [71],

Benchmark	Activity
FS Sequential	Sequentially scan a 256 MB section of a file system file 10 times
VM Sequential	Sequentially scan 256 MB section of allocated virtual memory 10 times
FS Random	Randomly read page-sized blocks from a 256 MB file system file two times
VM Random	Randomly touch virtual memory pages from 256 MB virtual memory allocation 2 times

Figure 5.11: **Calibrated Microbenchmarks.** *The table describes each of the microbenchmarks used to evaluate VMM-MemRx.*

relies upon certain properties of the operating system cache replacement policy to function correctly. Specifically, the algorithm used must (roughly) preserve the *inclusion* or *stack* property [61]. The key aspect of the stack property is that a cache of a size $N + 1$ has the same contents as a cache of size N, plus the one additional buffer which has some other block within it. LRU and LFU obey this property; FIFO does not [9]. By assuming the stack property holds, the VEC can efficiently *simulate* the contents of larger caches, safe in the knowledge that the buffers of the main page cache would be comprised of the same contents even if more memory were available.

Neither Linux, nor most other operating systems, employ a page replacement strategy that perfectly maintains the stack property. Our evaluation demonstrates, however, that MemRx is quite robust to these deviations under Linux for many useful cases.

5.4.2 Evaluation

We first evaluate the accuracy of MemRx by using it to measure the working set size of microbenchmark workloads for which the working set size is approximately known. Table 5.11 lists each of the microbenchmarks and the actions they perform; the working set size for each is approximately 256 MB and the virtual machine is configured with 128 MB of memory. Second, we compare the working set size predicted by MemRx to the working set size determined by trial and error for more realistic application workloads, in particular, Mogrify and Dbench.

Figure 5.12 shows the predicted and actual miss ratio curves for the four microbenchmark workloads. The miss ratio curve shows the fraction of the capacity cache misses occurring in the smallest memory configuration (*i.e.*, 128 MB) that remain misses in larger memory configurations. The *predicted* curve is calculated by MemRx using measurements taken during a single run at the smallest memory configuration and then simulating the page cache behavior of the guest operating system on-line for several larger memory configurations in increments of 32 MB. The *actual* curve is calculated by running the workload at each of the noted memory sizes and counting actual capacity misses in the page cache.

These calibrated tests show that MemRx can locate the working set size of simple

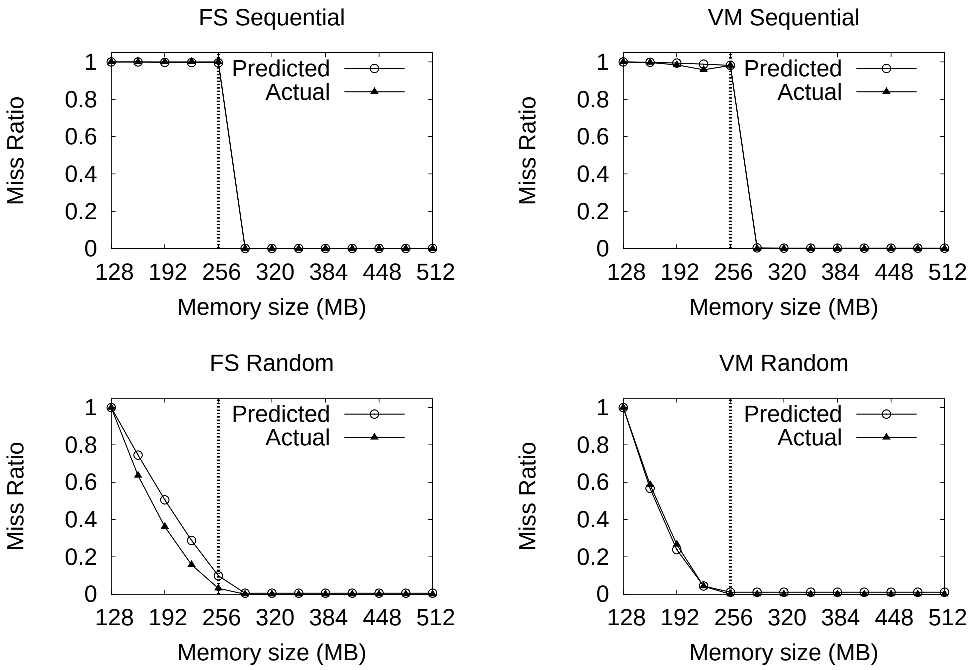


Figure 5.12: **VMM-MemRx Predicted vs. Actual Miss Ratio.** *The figure shows the miss ratio predicted by VMM-MemRx vs. the actual miss ratio measured for varying memory sizes. The known working set of 256 MB is marked by a vertical dashed line.*

workloads very accurately. The prediction made by MemRx is identical to that found by direct measurement using trial and error. The result is not surprising, because under these simple workloads, Geiger incurs few eviction false positives.

Figure 5.13 shows the results for the two application workloads, Mogrify and Dbench. The leftmost two graphs show the predicted and actual miss ratio curves. In these cases, the inferred working set size predicted by MemRx is slightly larger than the actual working set size found using trial and error. To determine whether the discrepancy was due to Geiger (*e.g.*, false positive/negative evictions or lag) or to MemRx (*e.g.*, cache simulation error) we implemented MemRx within Linux [51] and compared the predicted and actual miss ratio curves produced by that version. Within the operating system, MemRx has access to precise eviction and promotion information, which eliminates Geiger as a source of error. The rightmost two graphs in Figure 5.13 show the miss ratio curves obtained for the Mogrify and Dbench workloads using our operating system implementation of MemRx.

For the Dbench workload, the version of MemRx in the OS shows the same deviation as the one produced by MemRx in the VMM; this leads us to conclude that the cause of the deviation is MemRx simulation error. MemRx models the guest buffer cache using a strict LRU policy that does not exactly match the policy used by Linux, which is something more

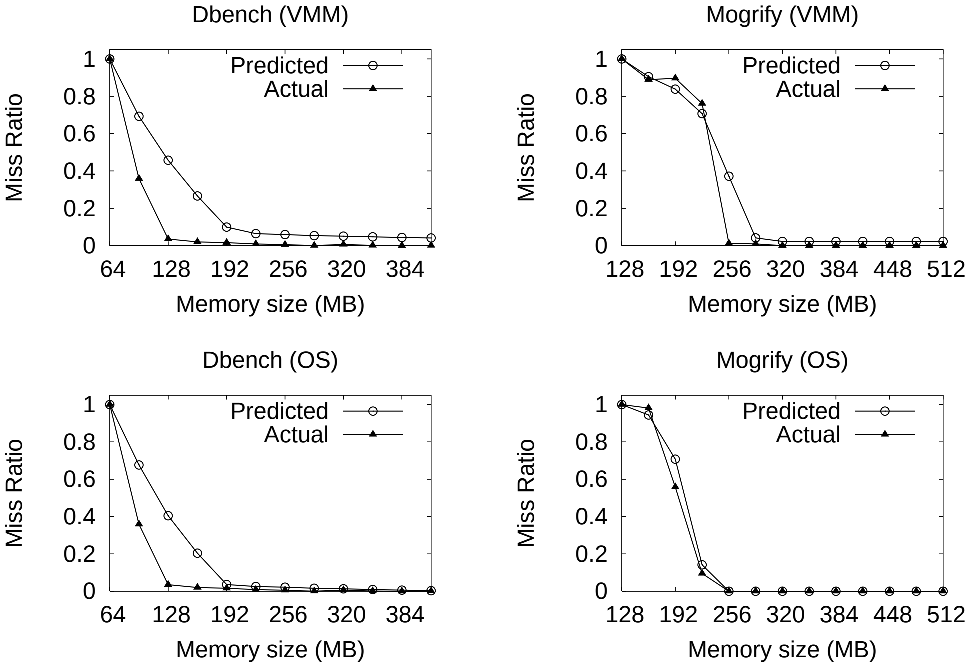


Figure 5.13: **Application Predicted vs. Actual Miss Ratio.** *The figure shows the miss ratio curve predicted by MemRx vs. the actual miss ratio measured for varying memory sizes for two application workloads. Results from MemRx implemented in the VMM (left) and MemRx implemented in the OS (right) are shown.*

akin to 2Q [48]. The difference between the modeled policy and the true policy leads to simulation errors like the one shown. In the case of Mogrify, however, the OS-based miss ratio curve matches the actual curve closely, leading us to believe that the error observed in the VMM-predicted working set size is due to the small inference errors imposed by Geiger and the granularity of the experiment.

In summary, the information provided by Geiger enables a VMM to estimate the working set sizes of thrashing VMs. The predictions made by MemRx are accurate enough to be highly useful when allocating memory between competing VMs on a single machine [101] or when selecting an appropriate target host during virtual machine migration [105].

5.5 Case study: Eviction-Based Cache Placement

In our second case study, we show how Geiger can be used to convey eviction information to a secondary cache. The basic idea is that the VMM uses Geiger to infer which pages have been evicted from the OS buffer cache, then sends this information (*e.g.*, with a DEMOTE operation [104]) to the storage server, which is potentially remote. The storage server uses this explicit information to perform eviction-based cache placement.

5.5.1 Implementation

Our implementation of an eviction-based secondary cache has two components. First, the VMM interposes on the virtual block device interface provided by Xen to see the block request stream generated by the workload. Second, the VMM uses Geiger to infer which blocks have been evicted from the guest OS buffer cache; these events are then communicated to the remote storage server. We simulate the behavior of a storage server by using the actual trace gathered from running Geiger for a given workload as input. We refer to our approach as Eviction-Geiger.

To evaluate our implementation, we compare it with three alternatives. In the first approach (Eviction-OS) the operating system is modified to report actual evictions; this represents the ideal case. In the second approach (Eviction-Buffer), the VMM performs only the eviction detections that are possible using client buffer addresses as used by Chen *et al.* [17] (*i.e.*read and write evictions). Finally, we simulate a storage cache that uses no information about client evictions and performs traditional, demand-based placement. In all cases we use an LRU-based replacement policy.

5.5.2 Evaluation

We use the application workloads listed in Figure 5.2 to evaluate our VMM implementation of eviction-based cache placement. For each workload, we consider remote caches from 32 MB to 512 MB. We evaluate the four placement policies: Eviction-OS, Eviction-Geiger, Eviction-Buffer, and Demand. Our metric is cache hit ratio.

Figure 5.14 shows graphs of the cache hit ratio vs. cache size for the four workloads and four cache policies. In all cases, OS and Geiger eviction-based placement outperform demand-based placement, sometimes significantly. The largest gains occur for moderate cache sizes where the working set of the application fits neither in the client cache nor in the storage cache individually, but does fit within the aggregate cache. OS and Geiger eviction-based placement are able to improve cache hit rate by as much as 28 percentage points for these workloads. For example, under the Mogrify workload using a secondary cache size of 96 MB, the cache hit ratio goes from 14.9% under demand placement to 42.9% when eviction-based placement is used. When the secondary cache size is large enough to contain the full system working set, OS and Geiger eviction-based placement perform similarly to demand-based placement. In the case of SPC web search, the traces exhibit almost no locality. The results are included for completeness only.

For one workload, Dbench, eviction-based placement with OS support outperforms that with inferred evictions, even with Geiger. For example, with a secondary cache size of 416 MB, we observe a difference in hit rates of about 15 percentage points. This performance difference is due to the significant time lag between the actual and inferred write-eviction events (approximately 2 seconds for most events in this experiment). Because some inferred evictions are delayed, the secondary cache loses the opportunity to place a

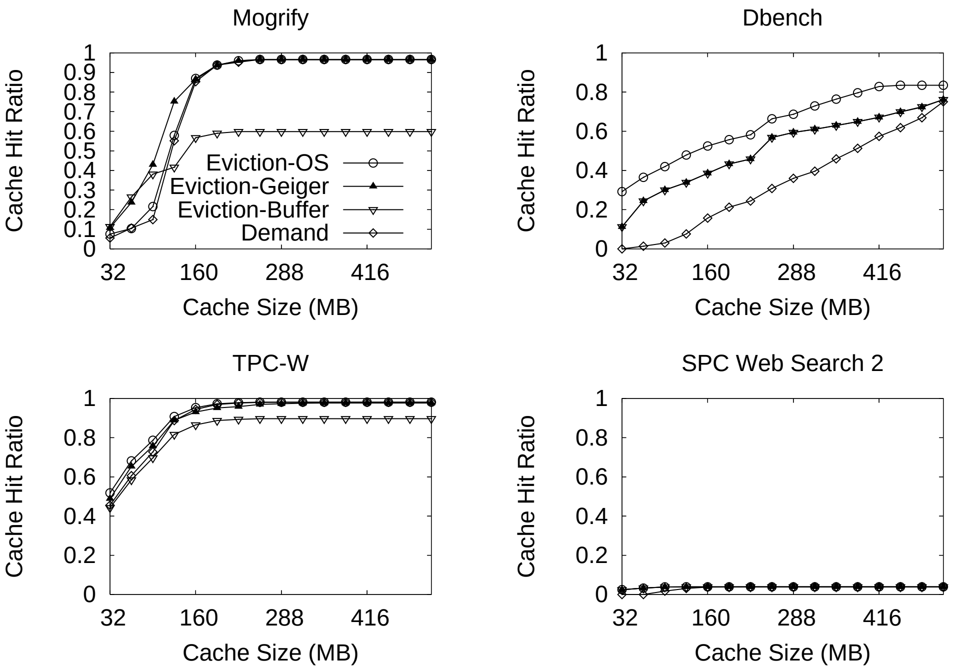


Figure 5.14: **Secondary Cache Hit Ratio.** *The figure compares the cache hit ratio in a secondary storage cache for various workloads when demand placement (Demand), eviction placement based on inferred evictions (Eviction-Buffer and Eviction-Geiger), and eviction placement based on actual evictions (Eviction-OS) is used. Experiments are performed using cache sizes from 32 MB to 512 MB.*

block prior to the block being referenced by the client, and a cache miss occurs. However, the eviction-based approaches still perform significantly better than demand-based placement.

Eviction-Geiger always performs as good or better than Eviction-Buffer. In fact, Eviction-Buffer sometimes performs significantly *worse* than straightforward demand-based placement. The problem occurs because Eviction-Buffer may detect fewer evictions than actually occur (*i.e.*, large false negatives). For workloads, such as Mogrify and TPC-W, where a significant number of non-I/O based evictions occur, missing evictions lead to poorer overall cache performance. Missing evictions are particularly a problem with large secondary caches, because few blocks are placed effectively, even though adequate cache space is available. In the case of TPC-W, missing eviction events change the cache hit rate by about 10 percentage points, while under Mogrify the difference is about 40 points.

In summary, Geiger can be used effectively to notify a secondary cache of the evictions that have been performed by clients. As confirmed in other studies [17, 16, 104], secondary caches using eviction-based placement can perform much better than those using demand-based placement. Our results show that the eviction information provided by Geiger is nearly as good as that which could be provided directly by the OS (if the OS were modified to do so). The one exception occurs when a significant lag occurs in the time between the actual eviction and the inference; however, even in this case, Geiger enables much better hit rates than those with simple demand-based placement. With eviction-based placement, it is essential to not miss evictions in the clients; eviction detection based only on I/O reads and writes can miss important evictions, leading to hit rates that are actually worse than simple demand-based placement. Therefore, the full set of techniques within Geiger should be used for buffer cache inferences.

5.6 Assumptions

As is the case with Antfarm, described in Chapter 4, Geiger also makes some basic assumptions about how operating systems work to form its inferences about the buffer cache. This section enumerates and discusses those assumptions.

Memory allocation mechanisms: Geiger assumes that most memory is allocated by an operating system via the small number of mechanisms listed below.

- Buffer-cache allocation during disk read and write
- Copy-on-write sharing
- Lazily in response to a not-present page fault

Geiger knowingly ignores other types of memory allocation as they are typically rare or concern small amounts of memory. Examples of memory allocations that Geiger ignores are non-cache kernel allocations (*e.g.*, inodes, directory entries, and other special purpose data structures).

Write sharing in memory is rare: The write and allocation eviction heuristics assume that processes do not often write-share large amounts of memory. While this is typically true, it would be simple to create a contrived workload that shared large amounts of writable memory. If this occurs, Geiger will spuriously report an extra eviction whenever a memory page within the writable buffer is lazily allocated by the operating system for any process, except the first, sharing the buffer.

Filesystems update in place: Some of the filesystem optimizations employed by Geiger to reduce false positives assume that the filesystem in use updates data in place. Nearly all filesystems have this feature. Some special purpose filesystems like WAFL [40], which is used on dedicated storage appliances, or ZFS [92], a new dynamic filesystem designed for the Solaris operating system, do not overwrite existing data on update in order to easily support filesystem features like creating and maintaining snapshots. WAFL is used in a proprietary appliance environment where system virtualization is unlikely. ZFS is a general purpose filesystem, but is quite new and not widely used.

Operating systems avoid unnecessary data copying: Some of Geiger's heuristics assume that operating systems avoid copying data around in memory. Such copying could break the association that Geiger tracks between disk locations and memory pages. In practice we have found that operating systems go to great lengths to avoid copying for performance reasons. In some cases copying is unavoidable. For example, in legacy systems using the ISA peripheral bus, devices are limited in the addresses they are allowed to use for DMA. In such a situation, the OS must copy data from low to high memory when necessary, a technique known as *bounce-buffering*.

5.7 Summary

In this chapter, we have explored techniques to make inferences about when pages are added to or removed from a guest OS buffer cache. We have found that modern operating systems, which typically incorporate unified system caches and journaling file systems, require new inferencing techniques that account for some previously ignored subtleties like anonymous memory allocation, aliasing and block liveness.

Geiger's full range of inference techniques are needed under different circumstances. For example, our COW and Allocation techniques are needed to handle an important class of applications that allocate significant amounts of anonymous memory; live block detection improves the accuracy of workloads that delete or truncate files; finally, writes to the journal must be isolated to handle file system data journaling. These features make a real impact on performance. For example, in some cases the number of false positives can be reduced by more than a factor of nine by taking block liveness into account.

Overall, our techniques are efficient. Our largest observed runtime overhead was 2.19% and overheads for more typical workloads were much less than 1%. In some cases we observed that the lag between actual and inferred events reduced the value of inferred information, but in general the information provided by Geiger is timely with average lag measured in small numbers of milliseconds.

Geiger allows us to implement two useful prototype case studies: MemRx, a working set size estimator that compliments and extends existing commercial techniques, and eviction-based cache placement for second-level caches. Recently, another group of researchers have proposed using Geiger and MemRx as components in their dynamic, virtual machine-based cluster management system [105].

Chapter 6

Detecting and Identifying Hidden Guest Processes

Stealth rootkits that can hide processes are an important security issue. According to statistics gathered from Microsoft's widely deployed *Malicious Software Removal Tool* [63], a significant fraction of the malware it encounters and removes consists of stealth rootkits with process and other resource hiding capabilities [67]. Half of unpatched Windows systems surveyed by the Microsoft tool are affected by a single rootkit alone. Often the stealth rootkit components are bundled and used by other kinds of destructive malware like remote control programs for botnets and spyware, extending their capabilities and complicating their detection and removal. The ability to detect and respond to malicious hidden processes is a clear advantage in the race to defend network-attached computers.

One way to detect that processes have been hidden is by using a technique called *cross-view* validation [102]. Cross-view validation works by observing a resource, like operating system processes, from multiple perspectives and noting inconsistencies between them. One view is obtained from an untrusted, high-level vantage point. The other is obtained from a low level in the system that is unlikely to have been subverted by an attacker; hence, its information is considered trustworthy. If a resource appears in the trusted view and does not appear in the untrusted view, a detector based on the cross-view principle can conclude that a resource has been hidden independent of the technique used to hide it.

One serious problem with cross-view validation is the inevitable race that develops between attackers and defenders to control the lowest reaches of a system. If an attacker subverts the level from which the trusted view is obtained, cross-view validation fails. Clearly, the deeper within a system a trusted view can be extracted the better. In this paper we describe a cross-view technique for detecting hidden processes that obtains its trusted view from deep within the system at the VMM-layer.

A virtual machine monitor (VMM) is an attractive place to deploy security monitoring services like anomaly detection systems [35, 52]. By virtue of their location behind the relative security of the virtual machine interface, VMM-based services are better shielded from malicious attacks that originate from within a guest virtual machine [55], even if the guest operating system kernel is compromised. In spite of being separated from guests by a secure barrier, a VMM maintains good visibility into the the activities and state of its guest virtual machines. For example, a VMM can easily read and write guest registers and memory and can observe guest I/O like disk and network requests.

The VMM-based security services that have been proposed to date assume that the VMM has detailed implementation information about the guest operating systems they observe [35, 52]. These services use information about the memory locations of private operating system variables and functions, the layout of compound structures, the call signatures of important operating system functions, and detailed semantics of various operating system components to perform their work. Some of this information can be obtained automatically from debugging symbols [35]. Other kinds of information are only available via careful study of system source code or reverse engineering [52].

VMM-level services based on explicit implementation information are effective, but there are drawbacks. One interesting consequence is that they may be just as susceptible to evasion by an attacker that has subverted the guest operating system as if they were located within the guest itself. In spite of their location at the VMM-layer, these services depend on guest-level information which is still open to simple guest-level manipulation. For example, if a service depends on the correctness of the guest operating system process list, a kernel-resident attacker can modify the list to hide its presence. If a security system depends on monitoring the location of a function like `fork` to be informed of process creation, it may be thwarted by an attacker that re-directs invocations of the system call to their own implementation.

In this chapter we present the design, implementation, and evaluation of *Lycosid*, a VMM-based security service that detects and identifies hidden processes. Lycosid does not depend on explicit guest operating system implementation information. Instead, general operating system principles and observations of architecture-level activity are used to infer required information about the activities and state of guest virtual machines. Like previous VMM-based security services, Lycosid is resilient to malicious guest attack by virtue of its location within a VMM. Unlike previous work, Lycosid obtains and uses true VMM-level information about guest operating systems which should render it less susceptible to guest evasion attacks. Additionally, by decoupling Lycosid from a specific operating system version and patch-level, the service can be deployed in diverse environments without the burden of maintaining version-specific implementation information.

The detection and identification of specific hidden processes provided by Lycosid enable a VMM to engage in a targeted response to this kind of malicious activity. A VMM that knows which processes are hidden can provide more specific and detailed logging. Per-process profiling information can be generated via a technique like on-demand emulation [41]. This additional detail enables a more effective post-mortem malware analysis. Finally, an aggressive VMM security policy might elect to pro-actively kill hidden processes, while allowing untainted processes to continue running.

We evaluate our Lycosid implementation using both Windows and Linux guests and find that it is highly accurate in a wide range of extremely challenging environments. This result comes despite the fact that the implicitly obtained information about guest virtual machines used by Lycosid is noisy and sometimes wrong [49, 50]. Accuracy is achieved via a targeted use of statistical inference techniques like hypothesis testing and linear regression that trade time for accuracy. Despite low quality inputs, Lycosid provides a robust, highly accurate, and portable service usable even in security environments where the consequences for wrong decisions can be high.

Lycosid bases most of its decisions on passively obtained information. In some cases, however, we find that passive information is inadequate to reliably identify which of many candidate processes has been hidden. Lycosid introduces a new technique called *CPU inflation* that allows a VMM to influence the runtime of specific processes by carefully patching a process's executable code. Using CPU inflation, Lycosid can often transform a detectable, but unidentifiable, hidden process into a hidden process that can be reliably identified, enabling an appropriate response.

6.1 Process Hiding

When a system is compromised, it is common for an attacker to leave programs behind that advance the attacker's goals. This approach is especially favored when the attacker accesses a machine from a remote location over a network. For example, an attacker will often leave behind a back door program that listens to the network and allows the attacker to regain a privileged presence on a compromised system without re-exploiting a vulnerability [81]. In other cases, key capture or file system scanning programs are left running to collect additional useful information like login names, passwords, and financial records.

The presence of unexplained processes, network connections, or files is an indicator to a system administrator or intrusion detection system that a successful attack has occurred. To avoid tipping off a defender, an attacker will often attempt to hide their malicious processes, network connections or data files [14]. Hiding is typically accomplished by modifying some aspect of the system using a suite of tools called a stealth rootkit. For example, some rootkits modify program binaries like `ps`, `netstat`, and `ls` [64]. Other rootkits hook into the call path between a user application and the kernel by modifying libraries, dynamic linker structures, system call tables, or operating system functions that report system status [42]. Finally, some rootkits manipulate kernel data structures using so-called direct kernel object manipulation (DKOM) [32]. Rootkit hooks and modified kernel data structures lead to corrupted results of user requests, effectively hiding the presence of malicious resources [20, 83]. The list of techniques available to hide system resources is long and growing.

Long lived malicious processes are the most likely candidates for hiding. The probability of detecting a short lived malicious process via a process introspection tool like `ps` is relatively small, so an attacker rarely goes to the trouble of hiding a short-lived process. The long-lived nature of maliciously hidden processes has implications for the kinds of detection techniques that are feasible.

6.2 Detection

The Lycosid service is partitioned into detection and identification components. We discuss the detection component in this section. Detection consists of determining if any processes running within a guest virtual machine have been hidden. The detection algorithm does not identify which specific processes are hidden. Identification is discussed in Section 6.3.

6.2.1 Approach

If a process has been hidden using any of the methods described in Section 6.1, it will not appear on a user-level process listing. It will, however, appear on a suitably obtained, trusted process list. Hence, to detect a hidden process we can compare the lengths of process lists obtained at a trusted and an untrusted level. If the trusted list is longer than the untrusted list we can conclude that at least one process has been hidden.

On an idle system, simply obtaining a single instance of the two process lists and comparing them would suffice to detect hidden processes. On an active system, however, where processes are being created and destroyed, the situation becomes more complicated. For example, Lycosid cannot perfectly synchronize the times at which it makes its two process list observations, so they may reflect different process-related states of the system. Additionally, the measurements taken within the VMM can be delayed, further complicating the inference. As the system experiences higher levels of process creation and exit activity, the problem becomes worse.

Lycosid overcomes these issues by employing statistical inference techniques. Specifically, it obtains many pairs of measurements over time and performs a series of paired-sample hypothesis tests [75]. Each pair consists of a process count obtained from within the VMM and a process count obtained from within the guest. Using a hypothesis test, we can determine if the two process lists differ in length even when the system process state is in dramatic flux. The test procedure also provides the ability to quantitatively limit the chance that we assert one or more processes are hidden when in fact no hiding is taking place, *i.e.*, the false positive rate can be explicitly controlled.

Formally, let T be the length of the trusted process list and let U be the length of the untrusted process list. Our null and alternative hypotheses are then:

$$H_0 : T - U \leq 0 \tag{6.1}$$

$$H_1 : T - U > 0 \tag{6.2}$$

We use the non-parametric Wilcoxon rank-sign statistic [75] in our tests because it makes no assumptions about the distribution of the population from which our samples are drawn. Data analysis indicates that the distribution of $T - U$ is quite symmetric, but has a slight negative skew and is not normally distributed.

If we can reject the null hypothesis H_0 in favor of the alternative hypothesis H_1 at an appropriate level of confidence, we can quantitatively conclude that one or more processes is being hidden. The hypothesis test p-value indicates the probability of a false positive, *i.e.*, indicating hiding when the null hypothesis H_0 (no processes are hidden) is true. As with most anomaly detection systems, the consequences for false positives in the detection performed by Lycosid are significant. Too many false positives degrade the confidence in the system and render the information it provides less valuable. Hence, we choose a conservative threshold confidence value ($\alpha = 2 \times 10^{-6}$). If the one-sided p-value computed during the hypothesis test falls below α, Lycosid reports that one or more processes have been hidden.

In addition to a hidden process indicator, the average difference observed between the two lists during the detection phase provides an estimate of the number of processes that have been hidden. This point estimate is used as input to the hidden process *identification* algorithm described in Section 6.3.

6.2.2 Details

Lycosid obtains a trusted view of guest processes from within a VMM. The VMM-based approach has advantages over any technique that obtains trusted information from within the guest itself because a VMM is typically much harder to subvert than guest software services or even the guest operating system kernel. This fact follows from the relatively smaller and well-defined virtual machine interface that separates the guest from the VMM [55].

VMI [35], for example, uses this advantage to provide various resilient security services within a VMM, one of which is hidden process detection. Lycosid differs from VMI in the way it obtains trusted information about the guest operating system. VMI exploits detailed information about the location and semantics of private kernel data structures to obtain a low-level guest process list. In contrast, Lycosid obtains its low-level guest information implicitly. This is a key advantage of Lycosid. No detailed implementation information about the guest is required. As a result, Lycosid can be deployed without taking versions and patch levels of the target operating systems into account.

Lycosid uses Antfarm [49] to obtain its low-level view of guest operating system processes. Antfarm is a VMM component that implicitly obtains information about guest operating system events like process creations and exits by observing closely related events like virtual address space creation and destruction. Information about virtual address spaces is explicitly visible to a VMM. Antfarm can also provide estimates of other process-related quantities like CPU time consumed, working set size, and context switch counts by observing their virtual address space analogues.

Lycosid obtains its untrusted view of guest operating system processes the same way that VMI does. A network connection is made from the VMM to the guest and a user-level program within the guest is invoked to enumerate processes. On a UNIX-like system the `ps` command can provide this information. On Windows systems, various utilities like `pslist.exe` [21] or the built-in `tasklist.exe` can be used. To minimize the data that must be transported over the network, Lycosid uses a custom process enumeration utility that returns only the information it requires. We use a custom utility to reduce the time required to obtain information from a guest, improving the synchronization between VMM and guest measurements.

Lycosid obtains trusted and untrusted process lists at short random intervals. A window of the most recent samples is preserved for use in hypothesis testing. The size of the window and the sample interval are configurable. In our implementation, samples are obtained every one second on average. Up to the most recent 600 samples are used in each hypothesis test. Approximately every minute, we test the null hypothesis that the two lists are the same length. Given the detection threshold $\alpha = 2 \times 10^{-6}$, our configuration corresponds to about one expected false positive per year.

6.3 Identification

After detecting that one or more processes have been hidden, the natural next step is to *identify* which processes have been hidden. By identifying the specific processes that have been hidden we enable a more effective VMM response to the malicious activity.

Given only the information provided by the hiding detector, each process visible from within the VMM is equally likely to be the culprit. Our approach for identifying which processes have been hidden is to select a measurable quantity associated with hidden processes and use it to select from the set of candidate processes.

6.3.1 Approach

As a process executes, it consumes CPU time. Both the operating system and a process-aware VMM like Lycosid can account CPU time to specific processes. Let G_i denote the CPU time for process i as observed from within a guest. Let V_j be the CPU time accumulated by process j as seen by the VMM. Then, when hiding occurs, the quantity

$$H = \sum_j V_j - \sum_i G_i \tag{6.3}$$

represents the total CPU time observed within the VMM that is not accounted for by processes visible to the guest, *i.e.*, it is the CPU time used by hidden processes. We can construct a linear equation using H and the per-process CPU times we have obtained from within the VMM.

$$H = \beta_1 V_1 + \beta_2 V_2 + ... + \beta_n V_n \tag{6.4}$$

Equation 6.4 holds if the coefficients β_j take the value 1 for processes that are hidden and 0 for non-hidden processes. We can identify likely hidden processes by fitting a multiple variable linear model using least-squares regression on Equation 6.4 and choosing the N variables from the model that best explain the variance observed in H, where N is the estimated number of hidden processes obtained during the detection phase. Hence, hidden process identification in Lycosid is a multiple linear regression variable selection problem.

There is no universal, automated technique available for variable selection in multiple regression that is guaranteed to select the best set of variables to include in a model. Stepwise procedures attempt to refine an over-specified or under-specified model iteratively, but often choose bad models. All-possible-subsets regression is guaranteed to choose the best model as long as the number of variables to include is known in advance. As the name implies, all-possible-subsets does this by trying all possible variable combinations of the specified size and maximizing a provided model statistic like the multiple R^2 measure. Unfortunately the cost of all-possible-subsets variable selection grows like $\binom{N}{E}$ where N is the total number of processes and E is our estimate of the number of hidden processes. Since the number of processes to choose from is often large in our environment, this technique is usually far too expensive.

Lycosid uses a simple variable selection heuristic that incorporates what we know about the form of the true model. We know that the coefficients of the variables representing hidden processes should be close to 1.0 and we have an estimate for the total number of hidden processes. Once an initial model has been fit, those variables corresponding to processes that are obviously not related to the extra observed CPU time are removed from the model. Specifically, variables with negative estimated slopes and variables whose estimated slopes are much greater than 1.0 (e.g., greater than 5 in our implementation) are removed. A new model is then fit using only the remaining variables. Finally, the N variables whose positive relationship to the extra CPU time is strongest are chosen. The strength of a variable's relationship to the extra CPU time is represented by the p-value that results from testing the null hypothesis that the variable's estimated coefficient is zero. Note that we do not attempt to interpret the resulting p-value as a probability related to our identification task. The p-value is simply used to order the variables according to the strength of their relationship to the extra observed CPU time. The top N variables from the ordered list are selected. As in the detection case, we employ a conservative threshold p-value ($\alpha = 1 \times 10^{-5}$) to reduce the chance of false positives, *i.e.*, of incorrectly identifying a process as hidden when it is not. If we do not find N variables with sufficiently small p-values, additional samples are taken and the procedure continues until a configurable upper limit of samples is reached.

6.3.2 Details

Lycosid obtains CPU time information about processes from both the VMM and from the guest operating system. CPU times for VMM-visible processes are obtained using Antfarm. As in the detection phase, Lycosid uses a custom network utility that calls documented APIs to obtain and return per-process CPU time information.

Samples are obtained from the VMM and from the guest operating system at small random intervals. In our prototype, samples are obtained about once per second on average. A sample consists of a set of process identifiers and the CPU time used by each associated process since the last measurement interval.

Figure 6.1 shows a notional data set used for identification purposes. Note that Lycosid is unaware of the mapping from guest process IDs to the abstract internal process IDs available within the VMM. No simple method of inferring this mapping currently exists. Otherwise identification would consist of a simple set subtraction operation.

Over time, samples are collected and stored. Once adequate samples have been obtained, a model can be fit and evaluated for hidden process identification. In our current implementation, an initial model is fit once `max(40, number of processes)` samples has been obtained. Up to a maximum of 1000 samples are obtained for use in identifi-

```
# VMM PID      VMM proc runtime (s)
0x3a40         1.219
0xad3f         0.203
0xf003         0.491
...
# Guest PID    Guest proc runtime (s)
30             1.103
495            0.422
933            0.001
...
```

Figure 6.1: **Sample Identification Data.** *The figure shows a notional data set used to identify hidden processes. There is no correlation between VMM and guest process IDs.*

cation.

6.3.3 CPU Inflation

The key feature used by our identification algorithm is the CPU time consumed by each process as observed from within the VMM and from within the guest operating system. It is important to note that the identification technique, unlike the detection technique, requires that the hidden process actually runs. Lycosid can detect, but not identify a completely idle hidden process.

Lycosid uses a new technique, called *CPU inflation*, that allows it to influence the CPU time used by a process. It is an intrusive technique used only when the passive methods already described fail to reliably identify a hidden process. CPU inflation works by transparently placing patches in guest program code. By forcing processes to run more frequently and more aggressively than they normally would, CPU inflation effectively increases the resolving power of Lycosid's identification techniques.

Details

When control is about to return from the VMM to a guest and CPU inflation is enabled, Lycosid determines the address where execution will resume and places a small patch containing a tight loop at that location. The patch forces the associated process to fully utilize its scheduling quantum until it is removed, effectively maximizing the amount of CPU time used by a process.

Patches are only placed when control returns to user-mode. In our VMM environment, nearly all VMM-to-guest transitions return to kernel-mode. Lycosid must therefore manufacture situations where the VMM returns to user-mode. It accomplishes this by arranging for high-resolution timer interrupts to occur a short time after a return to kernel-mode. The small extra interval allows the operating system to complete its current task (*e.g.*, interrupt processing) and return to user-mode where the guest is ultimately interrupted. The ideal length of the timer interval can be determined experimentally within the VMM by repeatedly increasing the interval until most timer interrupts occur in user-mode. By limiting patches to user-mode code, the normal guest operating system scheduler is free to de-schedule a patched process and the system remains stable.

In our implementation, after a patched process accumulates a certain amount of CPU time, chosen from a configurable, uniformly random interval, the patch is removed and the process is allowed to continue its normal execution. Patches are installed repeatedly according to a configurable patch schedule. Processes that are patched experience reduced performance, but are still allowed to make progress. When CPU inflation is enabled, patching is applied across all running processes. Lycosid enables CPU inflation when the detection module indicates hiding but the identification module is unable to identify the hidden processes.

6.4 Threat Model

Lycosid assumes few limitations on the abilities of an attacker. Our threat model allows an adversary complete control of a virtual machine including full system administrator privileges and the ability to observe and modify the operating system kernel. Indeed, hiding processes often requires an attacker to possess these abilities because privileged utilities, like `ps` on UNIX, operating system functions, like `EnumProcesses` on Windows, and key OS structures, like the process list, must be modified to implement malicious hiding.

The only limitation we place on the abilities of an attacker is that the VMM itself cannot be compromised. Clearly, an attacker that has control of the VMM could interfere with the functionality of Lycosid, which is also implemented at the VMM layer. We believe this limitation is reasonable because the architectural interface provided by a VMM to a guest operating system is relatively lean and, so far, has proven resilient to misbehaving and malicious guest software. While researchers have shown how to use a VMM to *implement* malware [57, 79], to our knowledge there have been no verified cases where a commercial-grade VMM has been compromised from outside by a guest.

6.4.1 Definition of Success

We consider it a success if Lycosid complicates successfully hiding malicious processes sufficiently such that the cost of hiding is significantly increased. As process hiding becomes more complicated and dangerous, an attacker will typically select a different stealth technique or forgo stealth altogether. We believe that Lycosid is a positive defensive step that helps to gradually remove opportunities to be stealthy from attackers.

6.5 Evasion

We claim that Lycosid is less vulnerable to evasion by guest software than previously presented VMM-based security services. Demonstrating that one system is more secure than another in general is notoriously difficult (or impossible). In this section we describe our rationale for the claim and why we believe implicit techniques can represent a net benefit for VMM-level system defense.

If a VMM-based security service depends on the correctness of any guest-level component, it is vulnerable to malicious corruption of that component [28]. For example, if a VMM uses the integrity of the guest operating system process list to determine when processes have been hidden, it is subject to evasion when a rootkit based on direct kernel object manipulation corrupts the list. The rootkit leaves the list in a consistent, but incorrect state. A VMM could use additional explicit information about other system components (e.g., thread scheduling queues) to detect inconsistency. The same approach has been taken by guest-level hiding detectors [78], for which there are, unfortunately, malicious work-arounds [1]. In this case, the VMM has no detection advantage over a guest-level tool because the information the VMM uses is fundamentally obtained from the guest.

Lycosid is based on implicitly obtained information about the observed guest virtual machine. The information is derived from the basic behavior of the guest operating system. For example, Lycosid uses process information provided by Antfarm. Antfarm obtains its process information by observing how a guest operating system manages its virtual address spaces. To evade Antfarm, an attacker must modify how the operating system implements a fundamental feature (virtual memory) and must do so in a way that remains consistent with its desired user-level view of processes.

In summary, Lycosid is perhaps best described as “differently” subject to gaming and evasion on the part of compromised guests. We believe the effort required to deceive Lycosid about ongoing process hiding while still maintaining a fully consistent outward appearance exceeds that of earlier VMM-based detectors. This is a feature of VMM-based security services based on implicitly obtained information and raises the bar against malicious process hiding.

6.6 Implementation

Lycosid is an extension to the Xen [27] VMM. The implementation of Lycosid is split between the Xen hypervisor and user-level programs that run in Xen’s privileged control virtual machine.

Antfarm [49] is one hypervisor component. It infers information about guest operating system processes by observing architectural events like page table updates and context switches. Antfarm provides the basis for Lycosid’s hidden process detection and identification. CPU inflation is also implemented as a core hypervisor feature. It interposes on Xen’s virtual CPU scheduling and shadow page table handling to selectively and safely patch user-level program code. Lycosid adds approximately 850 lines of C code to the hypervisor.

The data collection and analysis components of Lycosid that implement its hidden process detection and identification features are implemented as user-level programs running in a Linux guest virtual machine. They communicate with the hypervisor components of Lycosid via private VMM interfaces that are only available in Xen’s privileged control VM.

The analysis components are written in python and total approximately 6000 lines of code including statistics libraries and interfaces to libR.so [74], a statistical computing library.

By partitioning Lycosid, only necessary components are added to the hypervisor itself allowing it to remain relatively small, which is a desirable security property. The analysis components are normal user mode programs which can fail and be restarted without compromising the integrity of the whole system. They operate in polled batch mode which removes them from any synchronous critical path and allows them to amortize the cost of their communication with the VMM over many observations.

6.7 Evaluation

In this section we evaluate the performance of Lycosid's process detection and identification. We want to measure accuracy, timeliness, and runtime overhead. Accuracy is the ability of Lycosid to correctly detect and identify hidden processes measured in terms of false positives and false negatives. Our timeliness experiments measure how long it takes Lycosid to come to its conclusions.

6.7.1 Experimental Environment

Lycosid is an extension to the Xen [27] VMM version 3.0.3-testing. We use the Linux kernel version 2.6.16 in Xen's privileged control virtual machine. We evaluate Lycosid using two guest operating systems. The first is the retail version of Microsoft Windows 2000 Professional. The second is a default installation of Redhat Enterprise Linux 4.3. Both guests run unmodified using Xen's full virtualization support enabled by the Intel virtual machine extensions (VTx) [46]. Our experimental host has a 3.0 GHz Pentium D processor and is configured with 4 GB of system memory. Both privileged and unprivileged virtual machines are allocated 512 MB of memory. The system contains a single Seagate 7200 RPM Barracuda SATA hard disk drive.

6.7.2 Detection Evaluation

In Section 6.2 we noted that hidden process detection is complicated by multiple factors. For example, measurements make by the VMM cannot be perfectly synchronized, implicit information can be subtly inaccurate, and unrelated process creation and exit activity make the measurements obtained by Lycosid unstable. The key variable affecting the ability of Lycosid to detect hidden processes is how much unrelated process creation and exit activity is occurring within the monitored virtual machine. Process creation and exit activity tends to inject variability into the quantities measured by Lycosid and can magnify other, latent sources of variance inherent in the implicit measurement process like lag. To evaluate if Lycosid can accurately detect a hidden process in spite of these concerns, we perform many detection tests at various levels of process creation and exit activity.

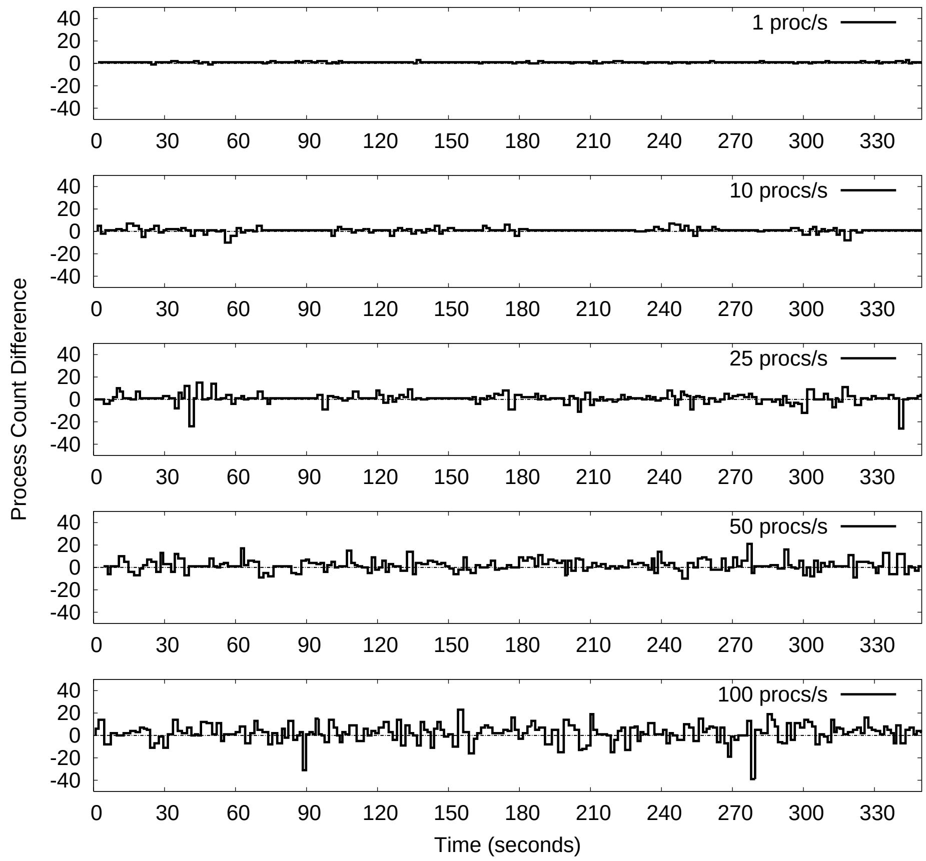


Figure 6.2: **Process Count Difference Timelines.** *The figure shows a timeline of the difference between the process list length obtained within the VMM and from the guest operating system for various levels of process creation and exit activity. As process activity increases the variability in the measured difference increases.*

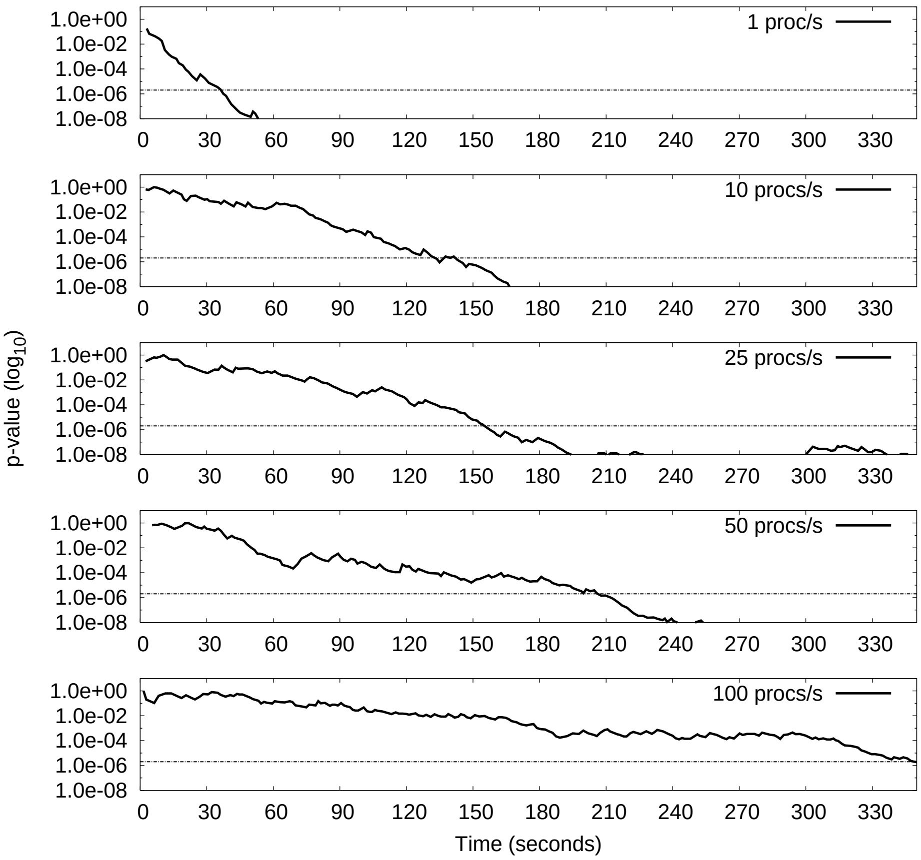


Figure 6.3: **Detection Timelines.** *The figure shows a timeline of the hypothesis test p-values used in the detection process for each of several levels of process creation/exit activity. The p-values approach the detection threshold over time.*

Detection with Interference

Our detection experiments evaluate the accuracy and timeliness of Lycosid when detecting a single hidden process. When more than one process has been hidden, the difference between the VMM and user process lists is larger, making detection easier. Hence, detecting a single hidden process is a worst case detection scenario.

The tests we perform explore how sensitive the detection techniques used by Lycosid are to unrelated process creation and exit activity. To generate process activity we use a synthetic process generator that spawns processes randomly. Harchol-Balter and Downey indicate in their study [39] that process arrivals are burstier than Poisson. We use a pareto distribution with shape parameter $k = 1$ for process inter-arrival times. We control the average rate of process creation by varying the pareto location parameter. This distribution leads to large process creation bursts which stress the detection techniques. The process lifetime distribution described by Harchol-Balter and Downey applies to processes whose lifetime exceeds one second. The arrival rates we use to stress Lycosid, however, are too high to support such long lived processes. As a result, we choose process lifetimes from the uniform distribution on the interval from 0–1 second, which allows our test system to remain stable.

To hide processes under Windows, we use the tool `fu.exe` and its accompanying device driver `msdirectx.sys` [32]. This tool hides Windows processes by unlinking the target process from the kernel process list. `fu.exe` is the most frequently encountered stealth rootkit removed by Microsoft's automated anti-malware tools [67]. Under Linux we simulate hidden processes by filtering process information in our guest process reporting tool. Unlike `fu.exe`, most recent Linux rootkits hide themselves and manipulate various logging and security features making them inconvenient in a research setting.

To motivate our use of statistical techniques, Figure 6.2 shows how the magnitude of the difference between VMM process count and guest process count used by Lycosid varies over time when the system is subjected to different levels of process creation and exit activity under Windows. As process activity increases from one to an average of 100 processes/second, the variance and amplitude of the signal representing the difference increase. This characteristic of the detection problem suggests the use of statistical inference techniques to probabilistically determine if hiding is occurring.

Figure 6.3 provides intuition about how the p-value resulting from the hypothesis test used by Lycosid incrementally approaches the detection threshold for the cases depicted in Figure 6.2. The test process is hidden immediately when each experiment begins. Detection occurs when the p-value drops below $\alpha = 2 \times 10^{-6}$, which is shown as a dashed horizontal line. In each case an orderly progression toward detection can be seen.

Figure 6.3 also hints that detection time increases with process activity. To quantify this effect, time to detection was measured for our various process activity levels. The results for Windows are shown in Figure 6.4 where the Y-axis reports the time to detection and the X-axis indicates the process activity level. The values shown for each level are the average of 10 trials. The standard deviation of detection time is shown using error bars. Both detection time and its variance increase with process creation and exit activity. In the worst measured cases, under severe process load, Lycosid requires several minutes to detect the

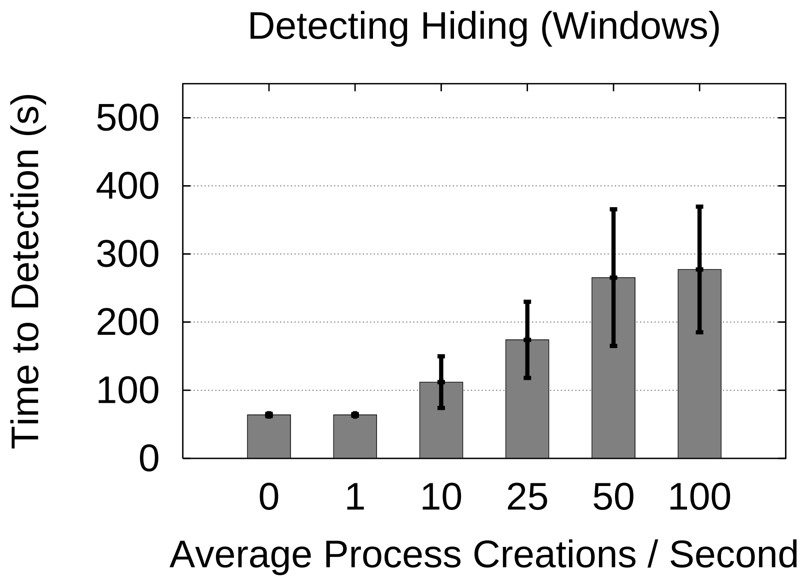


Figure 6.4: **Time to Detection.** *The figure shows how the time to detect a hidden process varies for Windows as process creation and exit activity increases from 0 processes/second to 100 processes/second. The values shown are an average of 10 trials. Error bars show the standard deviation of detection time.*

hidden process. Since hidden processes are typically long lived (on the order of hours or days) detection times of several minutes are not a real concern. In all of the experiments shown, Lycosid correctly detects the hidden process.

An important output of a positive detection result is an estimate of the number of processes that have been hidden. In the detection experiments described above, a single process was hidden, so, in each case a good estimate will be close to one. Figure 6.5 shows a summary of the estimated number of hidden processes obtained when a single process has been hidden under Windows. When process load is small to moderate, the estimated number of hidden processes is good, leading to a correct inference of one hidden process. Under extreme process creation and exit load, the estimates begin to experience larger error and greater variance. Under the most extreme (and most uncommon) load, 5 of 10 estimates are too high. This error may result in falsely identifying a non-hidden process as hidden during the identification phase. However, our conservative p-value identification threshold tends to reduce the chance of false positive identifications.

Portability

To explore the portability of Lycosid we repeat selected experiments performed for Windows guests using Linux. The setup of the Linux experiments mirrors that for the Windows

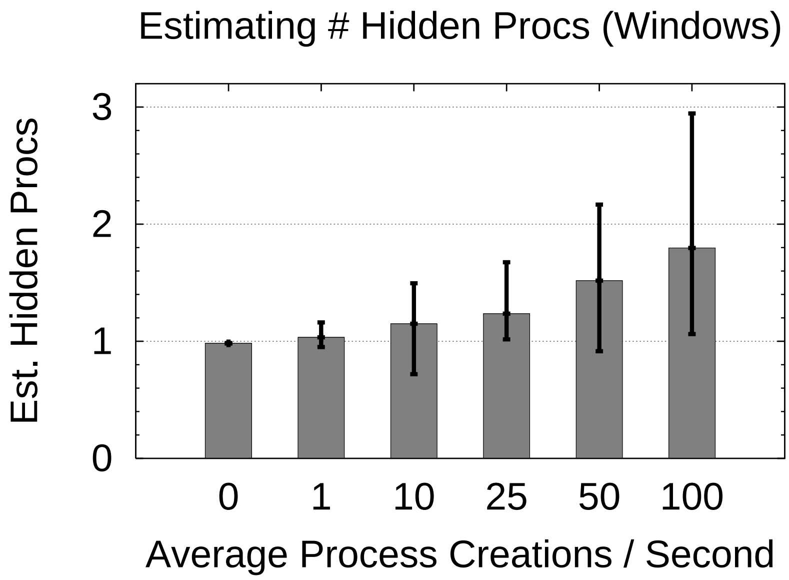


Figure 6.5: **Estimating the Number of Hidden Process.** *The figure shows how the estimate of the number of hidden processes obtained from the detection phase varies for Windows as process creation and exit activity increases from 0 processes/second to 100 processes/second when a single process has been hidden. The values shown are an average of 10 trials. Error bars show the minimum and maximum hidden process estimate observed.*

guests, *i.e.*, a single process is hidden with varying levels of process creation and exit interference. Figure 6.6 and Figure 6.7 show detect time and hidden process estimates. As in the Windows experiments, the values shown are averages of 10 trials. Error bars show the standard deviation of detection time and the minimum and maximum hidden process estimates observed. Under Linux, Lycosid correctly detects the hidden process in all cases. In most cases, detection occurs within the first 60 second test interval. For extreme interference levels, average detection time grows moderately with significantly larger variation between trials.

Under Linux, Lycosid estimates the number of hidden processes accurately except for very large process creation and exit activity. Interestingly, the direction of the error experienced by Lycosid when observing Linux guests is opposite of that experienced under Windows. Under Windows, Antfarm detects process creation *before* the operating system reports its creation, *i.e.*, process creation lag is negative under Windows. The opposite is true under Linux; Antfarm detects process creation after the OS reports it. High interference and load levels exacerbate the lag under both operating systems leading to larger deviations, but in opposite directions. Detection is not hampered, however, as our test statistic is not based on averages and does not depend on a specific distribution.

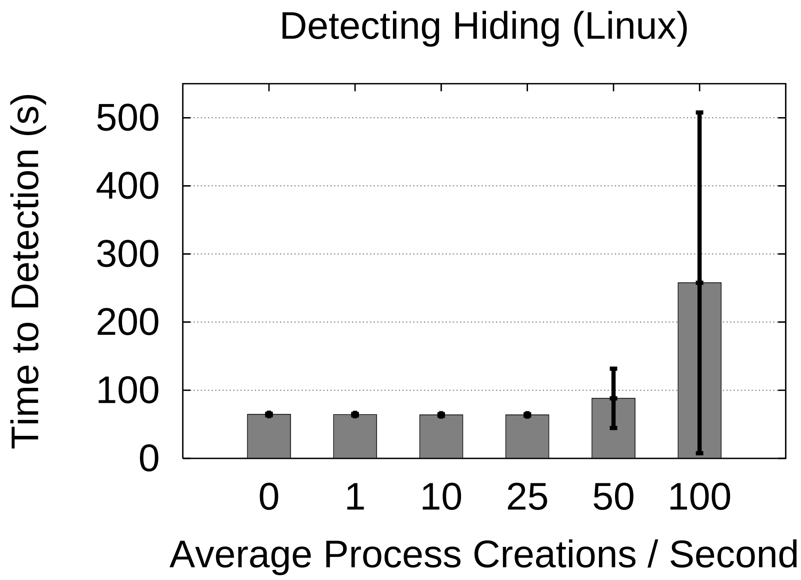


Figure 6.6: **Time to Detection.** *The figure shows how the time to detect a hidden process varies for Linux as process creation and exit activity increases from 0 processes/second to 100 processes/second. The values shown are an average of 10 trials. Error bars show the standard deviation of detection time.*

False Positives

In addition to reliable detection, it is important that Lycosid not report hidden processes spuriously, *i.e.*, that its false positive rate is small. Our statistical procedure predicts about one false positive result per year. To explore this question empirically, an experiment was performed using a Windows guest in which no process was hidden in our most challenging detection environment (100 process creations and exits/second). An 11 hour timeline from the experiment is shown in Figure 6.8. As can be seen, no trend toward false detection is apparent and no false detections occur. The experiment does not prove the formal claim of few false positives, but provides graphic empirical support.

Performance Overhead

Lycosid detection is meant to run continuously, so it is important that it impose minimal runtime overhead. To evaluate the overhead of the detection phase of Lycosid we compare the runtimes for three Windows benchmarks when they are run under Lycosid in detection mode and when run under an unmodified Xen hypervisor. Table 6.1 shows the results. Each value is an average of five trials. We observed no significant variance between trials.

Lycosid primarily adds overhead to Xen's shadow page table handling and virtual ad-

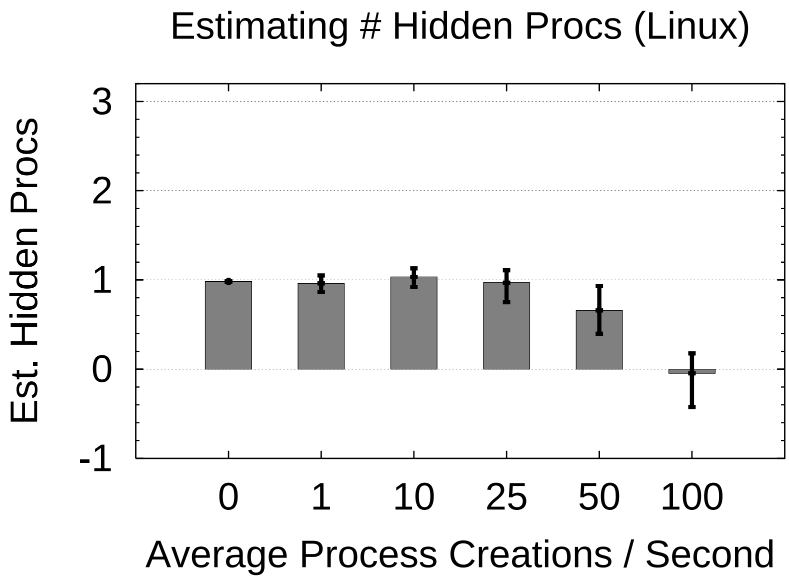


Figure 6.7: **Estimating the Number of Hidden Process.** *The figure shows how the estimate of the number of hidden processes obtained from the detection phase varies for Linux as process creation and exit activity increases from 0 processes/second to 100 processes/second when a single process has been hidden. The values shown are an average of 10 trials. Error bars show the minimum and maximum hidden process estimate observed.*

dress space switching. The first two benchmarks spend nearly all of their time performing these two tasks and can be considered worst case scenarios for Lycosid's detection performance. The *CreateProc* benchmark creates and then destroys 1000 processes as quickly as possible. The *MemAlloc* benchmark allocates a 200 MB segment of memory, then touches each page, causing many minor page faults and page table updates. MemAlloc is repeated five times in each trial. Our prototype experiences 5.3% overhead for CreateProc and 3.6% overhead for MemAlloc. The final benchmark is representative of a more common, but still demanding, workload. It consists of compiling a large C program using gcc. In this case, Lycosid adds a tiny 0.7% overhead.

6.7.3 Identification Evaluation

In this section we evaluate the ability of the identification algorithm described in Section 6.3 to identify which processes have been hidden once the detection component provides a positive hiding indicator. As in the evaluation of the detection phase, this evaluation focuses on Lycosid's accuracy and timeliness. In this case, accuracy is Lycosid's ability to correctly identify hidden processes. Our timeliness experiments quantify how long it takes to positively identify the correct hidden processes.

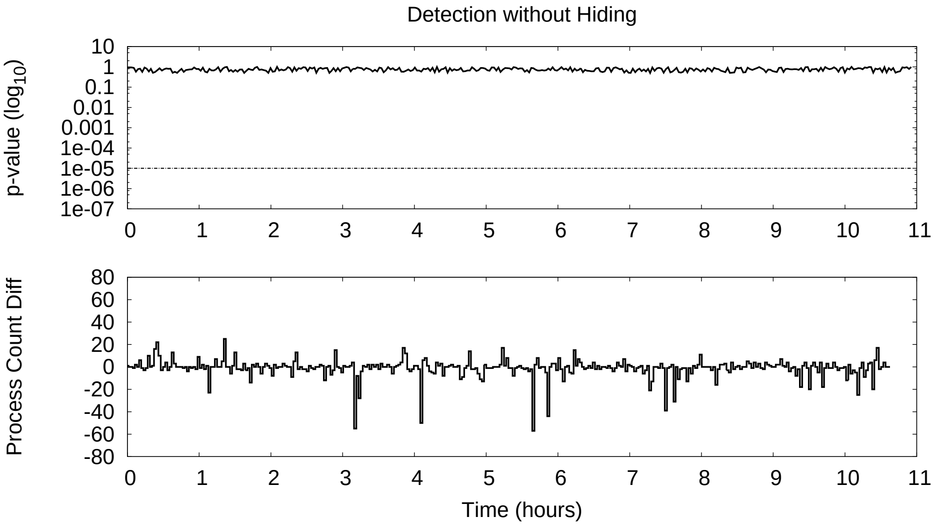


Figure 6.8: **Timeline without Hiding.** *The figure shows an approximately 11 hour detection timeline when no processes are hidden and very aggressive process creation/exit activity (100 processes/second) is present. The top graph shows the single-sided hypothesis test p-value. The bottom graph shows the difference between the VMM and guest process counts. No false detections occur.*

Identification Among Many Running Processes

Our first experiment measures how Lycosid performs when forced to choose among varying numbers of active processes. In the experiments, a number of processes (from 1 to 50) is created. Each of the test processes alternately runs and sleeps. The runtime is chosen randomly from the range 0–500 ms using a uniform distribution. Similarly, a sleep interval is chosen from the interval 0–1000 ms. One of the test processes is hidden using the same techniques described in Section 6.7.2. Experiments were performed with 1, 10, 25, and 50 total processes. At each level, 10 identification trials were performed. Lycosid correctly identifies the single hidden process in all cases. The time to identify the hidden process for both Windows and Linux guests is shown in Figure 6.9. The left hand bars show how identification time and standard deviation increase as the number of active processes grows when one process has been hidden. Detection time and variance grow because larger numbers of competing processes decrease the effective runtime of the hidden process. Hence, more samples are required to associate the runtime of the hidden process with the regression response variable in the face of measurement noise.

Hiding multiple processes is a common scenario when an attacker has several distinct tasks to accomplish on a compromised system. For example, an attacker may leave behind a network backdoor to enable remote control, a keylogger to steal passwords, and a network sniffer to acquire the addresses and open ports for targets on the same network. Does identification become more difficult when more than one process has been hidden? Our second

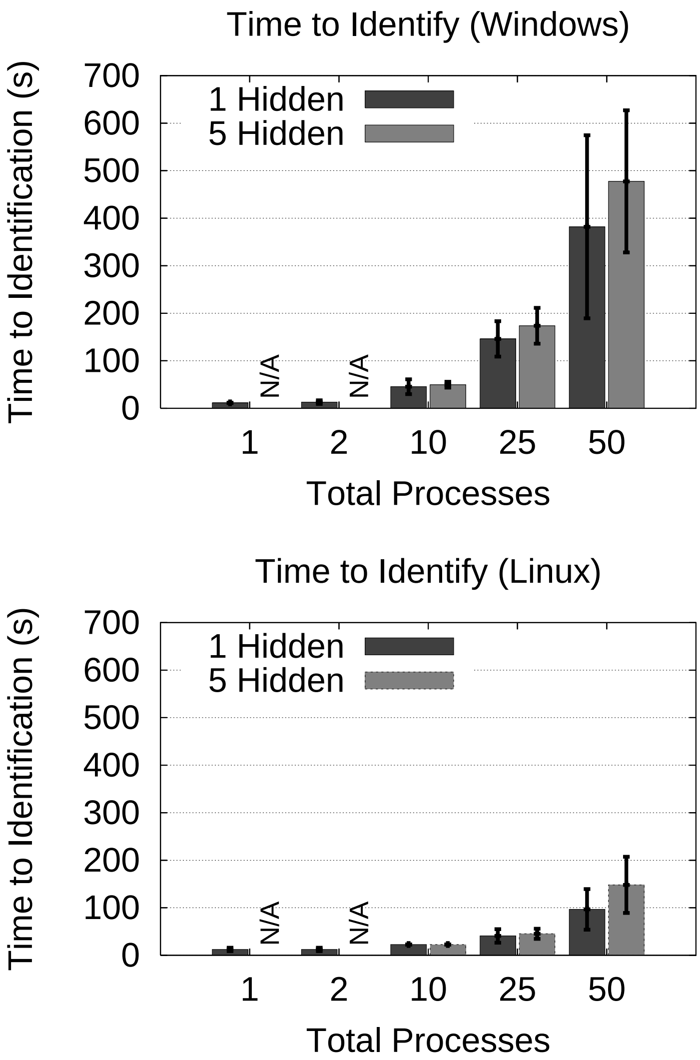


Figure 6.9: **Time to Identification.** *The figure shows how the time to identify hidden processes grows as the number of total active processes increases from 1 to 50 processes for both Windows (upper graph) and Linux (lower graph). The values shown are an average of 10 trials. Lycosid identified the correct hidden processes in all cases on both platforms. Error bars show the standard deviation of identification time. The left bar corresponds to trials in which a single process was hidden. The right bar shows results when 5 processes were hidden.*

Benchmark	Lycosid Runtime	Xen Runtime	% Overhead
CreateProc	6.551 s	6.222 s	5.3%
MemAlloc	6.803 s	6.565 s	3.6%
Compile	25.386 s	25.210 s	0.7%

Table 6.1: **Detection Runtime Overhead.** *The table shows runtimes and overheads for three benchmarks run under Lycosid and under a pristine version of Xen.*

Average Runtime (s)	Average Sleep (s)	% True ID	% False ID	% No ID
0.25	0.5	**100%**	0%	0%
0.025	0.5	**90%**	0%	10%
0.0025	0.5	**0%**	0%	100%
0.25	**5.0**	**100%**	0%	0%
0.25	**50.0**	**0%**	0%	100%

Table 6.2: **Identification under Reduced Runtime.** *The table reports the identification accuracy of Lycosid for a set of experiments in which a single hidden process must be identified among 10 active processes when the hidden process runs exponentially less and less often. As the relative runtime decreases, Lycosid's ability to classify a process as hidden or benign is impaired.*

experiment is similar to the first, but in this case 5 out of the 10, 25, or 50 total processes have been hidden. Again, Lycosid correctly identifies all hidden processes correctly for both platforms. The right hand bars in Figure 6.9 show that the time to identification grows for the multi-process case, but not significantly. Hence, Lycosid identification is accurate, portable across guest operating systems and applicable in cases where multiple processes have been hidden.

Identifying Mostly Idle Hidden Processes

Our next series of experiments demonstrates that a lower runtime bound exists beneath which Lycosid cannot identify which of several processes is hidden. We then test the ability of CPU inflation to overcome the issue.

We first perform two variants of an earlier experiment in which one process is hidden among 10 total active processes under Windows. In each variant we change the runtime of the hidden process along one of two axes. The first axis is busy time, *i.e.*, the time between sleep intervals. The second axis is run frequency, *i.e.*, the length of the sleep intervals. Reducing runtime along either axis decreases the signal-to-noise ratio between hidden process CPU time and the measurement error experienced by Lycosid. The effect is to make identification more challenging.

In the first set of experiments we exponentially reduce hidden process busy time by factors of 10 and measure the ability of Lycosid to identify the hidden process. In the

Average Runtime (s)	Average Sleep (s)	% True ID	% False ID	% No ID
0.025	0.5	**100%**	0%	0%
0.0025	0.5	**100%**	0%	0%
0.00025	0.5	**100%**	0%	0%
0.025	**5.0**	**100%**	0%	0%
0.025	**50.0**	**100%**	0%	0%
0.025	**500.0**	**100%**	0%	0%
0.025	**5000.0**	**20%**	0%	80%

Table 6.3: **Effect of CPU Inflation.** *The table shows how CPU inflation can help make hidden processes that run relatively little identifiable by Lycosid. In the experiments, a single hidden process must be identified among 10 active processes when the hidden process runs very little or infrequently. CPU inflation forces the hidden process to run more, providing Lycosid with the information it needs to make a positive identification. When average sleep time exceeds the maximum sample period, Lycosid naturally fails to reliably identify all hidden processes.*

second round of experiments we exponentially increase the sleep interval by factors of 10 and again evaluate if Lycosid can identify the hidden process. Table 6.2 lists the runtime parameters for the hidden process in each experiment and the percentage of 10 trials in which Lycosid successfully identifies the single hidden process.

When the busy time is reduced from earlier experiments by a factor of 10 Lycosid correctly identifies the hidden processes in only 9 of 10 trials. After reducing the runtime by a factor of 100, no process exceeds the identification threshold p-value before the implementation sample limit of 1000 is reached; hence, no process is identified as hidden. When the sleep time increases by a factor of 10 or 100, none of 10 trials produces a positive hidden process identification. Note that in no case do false positives occur, *i.e.*, no innocent processes are accused of being hidden. We see, however, that if a hidden process runs for limited periods, even if it runs regularly, or if a hidden process runs infrequently, Lycosid cannot identify it properly. Even in these cases, however, Lycosid correctly detects that process hiding is taking place.

Table 6.3 shows the results of applying CPU inflation to identification tasks in which the hidden process runs for short periods of time or rarely runs. Our evaluation shows that CPU inflation enables Lycosid to identify processes whose average busy time is as low as 250 μs. The table also shows that even when a hidden process runs relatively rarely (*e.g.*, once every 500 seconds on average) CPU inflation makes the hidden process identifiable by Lycosid. Finally, when the hidden process's average sleep time exceeds the amount of time over which Lycosid makes observations (once every 5000 seconds vs. approximately 1000 seconds of observation time in this experiment) Lycosid is naturally unable to reliably identify the hidden process. Our evaluation shows that CPU inflation is a powerful tool that significantly extends the set of hidden processes that Lycosid can reliably identify.

6.8 Attacks on Lycosid

Lycosid depends on an untrusted, user-level process view. One way to attack Lycosid is to manipulate its user-level view.

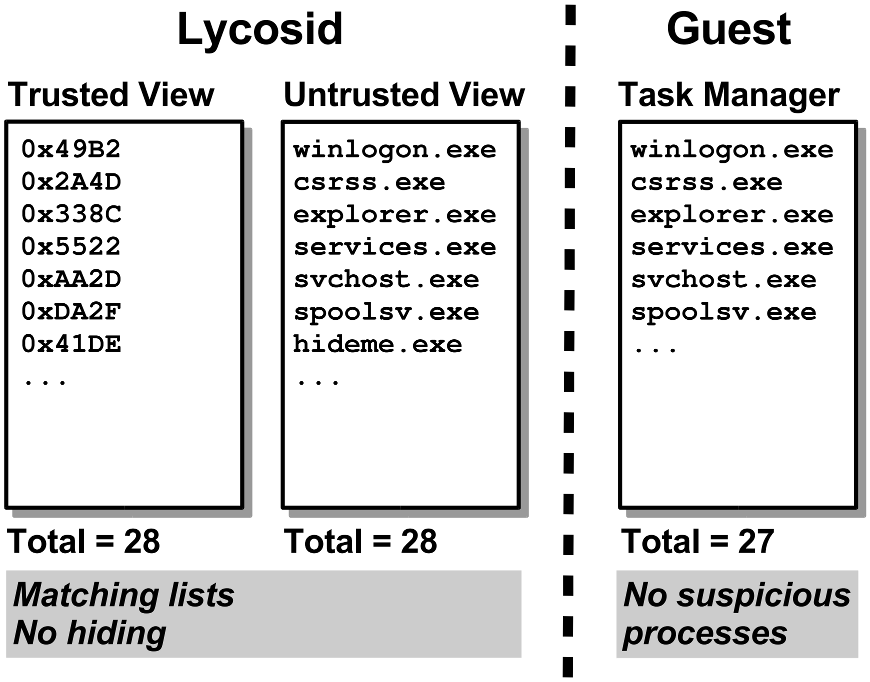


Figure 6.10: **Desynchronization Attack.** *The figure demonstrates the desynchronization attack concept against Lycosid hidden process detection.*

6.8.1 Desynchronization

The attack works by *desynchronizing* the untrusted, user-level view used by Lycosid and the user-level view used by a defender to detect unexpected processes (*e.g.*, Windows task manager). In the desynchronization attack, an adversary hides the presence of a malicious process from a defender, but doesn't hide it from Lycosid. In this way Lycosid fails to detect hiding because, from its perspective, no hiding takes place. A defender fails to detect the hidden process because, from their perspective, the malicious process does not exist. Figure 6.10 shows a conceptual example of the desynchronization attack.

To successfully mount this style of attack, an adversary must be able to reliably identify process enumeration requests made on behalf of Lycosid. In the general case, this task will be difficult because Lycosid uses the same standard APIs to enumerate processes as any other process introspection tool like `ps` or the Windows task manager. Additionally, Lycosid is not limited to using a single tool with a fixed signature to obtain its user-level process view, so an attacker cannot easily rely on a fixed signature database of known Lycosid probe programs. In the same way, there are many different tools that can be used by a defender to enumerate processes (*e.g.*, ps, top, task manager pslist, tasklist). For the sake of this discussion, we will assume an attacker can reliably identify and preferentially handler any Lycosid process enumeration request.

6.8.2 Countermeasures

Lycosid is designed to be a part of a larger, comprehensive security monitoring framework. Such a framework would include a process monitoring component that continuously observes the process list and generates an alert when unexpected or suspicious processes are encountered. It is just such a security feature that an attacker hopes to deceive by hiding their malicious processes. The desynchronization attack described above assumes that the process view used by the process monitor component is different from the view used by Lycosid. By integrating the process monitor and Lycosid so that they both use the same user-level process view, the opportunity to desynchronize is removed and the attack fails.

6.9 Assumptions

Lycosid makes certain assumptions about operating systems and the attacker. Assumptions made beyond the threat model stated in Section 6.4 are enumerated and discussed in this section.

Whole-process hiding: Lycosid targets only *whole-process hiding* in which a malicious user-level program is executed normally and the presence of that normal process is hidden, using arbitrary techniques, from a defender. Other hiding techniques exist such as injecting a thread into an already existing, long running process, hiding in plain sight by mimicking the name and other characteristics of an existing, benign process, or dispensing with a user-level process altogether by deploying completely operating system kernel-resident malware. Lycosid, because it is based on user-level process information, does not detect these less common, alternative hiding techniques.

Statistical inference assumptions: Lycosid uses hypothesis testing and linear regression to detect and identify hidden processes. These techniques require certain assumptions to produce reliable inferences. We use a non-parametric statistic during the Lycosid detection phase, so no distributional assumptions are required. Linear regression, used during identification, assumes independence of errors, constant error variance, linearity, and normality of errors. Data analysis shows that the data used by Lycosid meets all regression requirements except normality of errors.

Residuals obtained using the models created by Lycosid are not normal, but are quite symmetric. Non-normality of errors affects the reliability of the p-value produced during the hypothesis test of whether a model coefficient is zero. However, we do not directly interpret these values as probabilities. They are only used to order the coefficients during model selection. Hence, we believe this slight deviation is justified.

6.10 Summary

Stealth rootkits that allow attackers to hide malicious processes are a current and alarming security issue. In this chapter we have described, implemented, and evaluated a novel VMM-based hidden process detection and identification service called Lycosid. Lycosid differs from prior VMM-layer process hiding detectors because it uses noisy information about internal guest operating system state and events available implicitly to a VMM. Lycosid provides an accurate and reliable service in spite of its noisy inputs by using statistical inference techniques like hypothesis testing and regression to trade detection and identification time for accuracy.

In our evaluation, Lycosid correctly detected process hiding in each of hundreds of trials. Identification is similarly robust except in cases where a hidden process does not run long enough or frequently enough. To overcome this limitation, we have introduced CPU inflation to force processes into an execution regime in which a hidden process can be positively identified.

The performance-critical portions of Lycosid are based on Antfarm and exhibit similar runtime overheads. In a worst-case performance scenario, Lycosid's detection phase exhibits less than 6% overhead. For a more typical, process-intensive workload, Lycosid imposes a mere 0.7% penalty.

An interesting consequence of our use of implicit information is that Lycosid is likely less susceptible to evasion attacks on the part of a compromised guest OS. To evade Lycosid, an attacker must modulate externally visible behavior in very specific ways, and achieve their hiding goals at the same time. This complicates hiding and may drive attackers into more difficult, error-prone, or risky hiding scenarios like thread-injection or kernel-resident malware.

Our implementation of Lycosid demonstrates that implicit operating system information can be effectively used at the VMM-level even when the cost of being wrong is high as in a security monitoring service.

Chapter 7

Related Work

The work described in this dissertation focuses on developing, implementing, and evaluating techniques that allow a VMM to implicitly obtain and exploit information about important software constructs within the guest operating systems running above it. We have benefited from previous research efforts that have explored how to obtain and use information across layer boundaries in hierarchical layered systems. In this chapter we survey related work and describe how our own research fits into the context that it provides.

7.1 Gray-box systems

The term *gray-box* [5] refers to any technique that uses observation and measurement to obtain information about software or hardware across a system layer boundary. The system call interface that separates an operating system from user applications is an example of such a boundary. A system can use gray-box information to optimize its own performance or to control the behavior of cross-layer components.

For example, gray-box information about operating system memory management can be exploited by a user application to reorder its disk accesses to prefer blocks already resident in the OS buffer cache [13] or to carefully manage its memory allocation to avoid paging [5]. Overriding the default file system layout scheme to optimize for cross-directory access patterns in a web server [69] is an example of exerting influence across a system boundary using gray-box techniques when no explicit control interface exists.

Gray-box information has also proven itself useful to components logically below the operating system. For example, Sivathanu *et al.*, have shown that file system semantic information, such as how disk blocks are grouped to form files or metadata, and whether disk blocks are live or dead in the file system, can be used to create RAID storage systems that degrade gracefully in the face of multiple failures [86] and to reduce downtime resulting from failure by recovering only live blocks [85]. Bairavasundaram *et al.*, use gray-box knowledge of file systems within a storage device to infer which blocks are resident in a client buffer cache [7]. Gray-box information about client cache contents can be used to implement an effective exclusive caching component [104] within the storage system without modifying the storage interface.

The techniques described in this dissertation focus on inferring information across a different system boundary, namely the virtual architecture interface that separates a VMM from its guest operating systems. Like gray-box storage systems, our modified VMM resides below the operating system and bases its inferences on interpreting the stream of requests supplied by the OS and user applications. The types of information available to a VMM are considerably richer than that available to a storage system and include processor interrupts, virtual memory configuration, memory contents, and I/O requests. Richer information enables additional applications not feasible or appropriate within a storage device.

7.2 Guest Information in a VMM

Other researchers have recognized the value of OS-level information in a VMM. We categorize this body of work by the method used to obtain information about the guest OS.

7.2.1 Paravirtualization and Explicit Interfaces

One straightforward and widely used method to obtain information about the internal state of a guest operating system is to create new interfaces that provide the information directly. Paravirtualization [27, 103] is a virtualization technique that replaces expensive, implicit guest requests like page table updates and I/O requests [31] by a virtualization-aware interface similar to a system call. The primary goal of paravirtualization is to reduce virtualization overhead. Overhead is typically reduced by streamlining guest-to-VMM interfaces and via batching, which allows a guest to amortize the cost of VMM invocation across many requests.

The goal of our work is to transparently enable useful VMM-level services. In many cases our goal requires information about high-level guest OS abstractions like processes and I/O caches. Paravirtual interfaces were not designed with this goal in mind and do not include the ability for a guest OS to inform a VMM about its high-level internal state or events. While explicit interfaces would greatly simplify the task of obtaining guest information within a VMM, such interfaces do not exist today. Porting operating systems to take advantage of new explicit interfaces requires significant engineering effort. Creating standard, multi-vendor interfaces requires immense political effort. These costs will likely hamper the creation and adoption of standard, multi-OS VMM interfaces.

7.2.2 Explicit Information

Several recent projects obtain guest level information by embedding details about the version-specific memory layout and OS-specific data structure semantics of a guest into a VMM [35, 52, 56, 72, 68]. Required implementation details can sometimes be automatically extracted from debugging symbols and libraries [35], but often detailed source or binary analysis is needed to obtain them [52, 72, 68]. Systems that take this approach read kernel data structures, like the process list, program headers, and system call tables, directly. Some use information about the location and semantics of key kernel functions like `fork`, `exec`, `mmap`, or `try_to_swap_out` to stay informed of important guest events. Reading and using data drawn directly from guest kernel memory ties a VMM to specific guest OS vendors and versions, and creates an implicit relationship of trust between the VMM and the guest. Implicit techniques, like those described in this dissertation, help to combat these drawbacks at the expense of more limited and slightly less reliable information.

7.2.3 Implicit Information

Other attempts to obtain implicit information about guest operating systems have mostly been confined to determining how a guest utilizes the virtual resources it has been allocated. Disco [12], for example, determines when a guest is not using is CPU allocation by detecting when it enters a low-power processor mode. VMWare's ESX Server [101] uses page sampling to determine the utilization of physical memory assigned to each of its virtual machines to aid in page reallocation. Work by Uhlig *et al.* [98] introduces techniques to manage processor resources more intelligently in a multiprocessor VMM environment. It is most similar to our own work because it infers the state of a guest *software* construct. Specifically, they deduce when no kernel locks are held by observing when the guest OS is executing in user versus kernel mode. The techniques we describe target different, more complex software abstractions like processes and disk caches. Implicit techniques can be easily composed to form more comprehensive solutions.

7.3 Statistical Techniques

Lycosid uses statistical techniques like hypothesis testing and regression to transform uncertain, implicitly obtained input information into reliable intelligence that can be confidently acted upon. Many other systems employ statistical techniques to infer behavior, to provide input to control algorithms, and to implement security classifiers. For example, MS Manners [26] uses hypothesis testing to regulate the scheduling of low-priority background processes and to reduce their performance impact on high priority foreground jobs. Jung *et al.* [54] use sequential hypothesis techniques to probabilistically determine whether remote hosts are conducting port scanning by using sequential hypothesis testing techniques [100]. Bayesian spam filters [80] and statistical anomaly detection systems [30] use statistical learning techniques to build a model of normal behavior, then compare that model to arriving email, network packets, or other measurable system features to determine if they are abnormal.

7.4 Case Studies

To demonstrate the practical value of guest OS information within a VMM we have developed several applications as case studies. We drew these case studies from existing applications implemented using different techniques or in a different system layer. In this section we briefly discuss the origin of some of our case studies and how our VMM-based variations compare with previous implementations.

7.4.1 Working Set Size

In a virtualized environment, knowing the *working set size* [24, 25] of a virtual machine is useful for allocating the appropriate amount of memory to it. When migrating VMs [19, 82] in a distributed computing environment [29, 106] working set size information enables the job scheduler to intelligently select a new host with an adequate amount of available memory.

Techniques for estimating the working set size of a virtual machine have been explored by Waldspurger and are part of the VMware ESX Server product [101]. However, the ESX Server technique can only determine the working set size for virtual machines that are using less than their full allocation. Our working set size estimator complements the ESX Server technique by directly supporting situations where a virtual machine needs more memory, *i.e.*, it is thrashing.

7.4.2 Secondary-level Caching

Knowledge of the contents of the OS buffer cache is useful in a virtualized environment when implementing an effective *secondary-level cache*. For example, when multiple VMs run on the same machine, the VMM can manage a shared secondary cache in its own memory, increasing the utilization of memory when the VMs share pages [12]. Additionally, when the hosted OS is a legacy system that cannot address a large amount of memory, a secondary cache can enable the legacy OS to exceed its natural addressing limits. Finally, the VMM can explicitly communicate with a remote storage server cache, informing it of which pages are currently cached within each VM [104].

Designing a secondary cache management policy is non-trivial. Secondary storage caches exhibit less reference locality than client caches because the reference stream is filtered through the client cache [66]. This, plus the fact that secondary storage caches are often about the same size as client caches has led to innovations in cache *replacement* policies [108] and in cache *placement* policies [104]. We have implemented one promising placement policy called "eviction-based placement" which inserts blocks into the secondary cache only when they have been evicted from the client cache. This approach tends to make the caches overlap less and leads to more effective secondary cache utilization [17, 104]. Eviction-based placement is similar to micro-architectural victim caches in the processor cache hierarchy [53].

Passive eviction detection in support of exclusive secondary caching has been explored to some extent by storage system researchers. For example, X-RAY [7] uses file system semantic information (*e.g.*, which storage blocks contain inodes) to snoop on updates to a file's accessed time field. Knowing which files have been recently accessed allows X-RAY to build an approximate model of a client's cache. However, X-RAY is somewhat limited in its inferences because the storage system only has access to the I/O block stream outside the OS.

Other exclusive caching work has assumed that one has access to more OS information; for example, Chen *et al.* [16, 17] perform their inferencing within a pseudo-device driver that has access to the addresses of the memory pages that are being read and written. Thus, they are able to infer that an eviction has occurred when a memory page that is storing disk data is reused for other disk data.

Our approach to secondary disk caching is most similar to that of Chen *et al.*, but uses additional information available to a VMM to improve its ability to accurately infer cache events. The key differences include: handling unified buffer caches and virtual memory systems, recognizing when blocks on disk are free to avoid false evictions, and taking file system journaling into account to avoid disk block aliasing.

7.4.3 Hidden Process Detection

Cross-view validation for hiding detection has been studied and variously implemented in user applications [22], within the operating system kernel [102], inside a virtual machine monitor [35], and using dedicated coprocessor hardware [72]. The key aspect of cross-view validation that differentiates these efforts is the mechanism used to obtain the low-level, trusted view of the resource of interest.

Garfinkel *et al.*, have shown the value of VMM-level cross-view validation for detecting hidden processes with VMI [35]. VMI uses explicit operating system debugging information like the memory addresses of variables and the layout and semantics of compound structures to locate and interpret private kernel data types at runtime. This insight into operating system data structures is used to obtain a trusted view of the guest operating system process list. Lycosid extends the VMI concept by using only implicitly obtained guest information within a VMM.

Chapter 8

Conclusion

Virtualization technology is rapidly penetrating commodity computing systems. Key microprocessor vendors like Intel, AMD, and IBM are supplying hardware virtualization assistance that promises to vastly reduce the overheads imposed by virtualization. VMM implementations like VMware and the POWER5 hypervisor are mature and robust. Leading operating systems like Microsoft Windows, Linux, Solaris, and i5/OS already include or will soon include virtualization as a core feature.

In a system that includes a virtualization layer, the VMM is a natural place to implement certain key features. For example, only the VMM has the necessary insight and control to globally optimize resource allocation and scheduling, making these tasks a good match for VMM-level implementation. VMM-based security services are another example; they can monitor vulnerable, network-facing guest operating systems and applications from behind the relative safety of the virtual machine interface.

We, and other researchers, have shown that many potential VMM-layer innovations require information about high-level guest software abstractions–information that a VMM does not intrinsically have. This dissertation has explored a portion of the VMM-service design space that has been mostly ignored. We have shown how a VMM can independently and implicitly obtain information about key guest OS software abstractions by observing how the guest interacts with virtual hardware resources like the MMU and storage devices. Our techniques have proven to be accurate, low overhead, and portable across multiple guest operating systems.

8.1 Lessons Learned

In the process of developing our techniques and creating our prototype implementations we have been able to make several general observations about building services within a VMM.

OS responsibilities and available architectural features constrain guest implementations

General purpose operating systems like Windows and Linux all share certain key constructs and responsibilities. For example, all support the basic OS abstractions like processes, threads, address spaces, and persistent data storage in file systems. All provide certain characteristics like process memory isolation, starvation-free scheduling, and basic virtual memory semantics. An OS must implement these features using the mechanisms provided by the underlying hardware architecture, such as the memory management unit, timers, memory, and disk devices. The constraints imposed by common architectural features and OS requirements lead to generic, externally observable patterns in the behavior of guest operating systems that a VMM can observe and use.

In general, the less implementation flexibility provided by the architecture to the OS, the more constrained and easy to interpret the behaviors of the OS will be. For example, the hardware-defined page tables of the x86 architecture provide a more concrete basis for address space tracking than the software managed TLB provided by SPARC.

Reuse is a good proxy for release

The inference techniques described in this dissertation are often based on detecting when a resource is allocated and deallocated. For example, process creation and exit correspond to address space allocation and deallocation. We have found that detecting allocation is often quite simple. Detecting deallocation is often more difficult. The principle that reuse implies release has been helpful in detecting deallocation in several cases.

Time is of the Essence

In the process of building VMM-based services, we have found that *when* a certain guest OS event has occurred is important. This stands in contrast to much previous gray-box research which has focused on discovering static configuration parameters (*e.g.*, the cache replacement policy) or the current state of a resource (*e.g.*, whether a file block is cached). Invariably, the time at which a VMM observes that an event has occurred using implicit techniques is different from the time the event occurred as defined by the guest OS. Lag between actual and inferred events places practical limits on how implicit information can be used. We have seen that delayed, but correct, process and cache information can be useless, while short term errors cause no harm or even help certain applications. In general, minimizing lag is just as or more important than rigidly reproducing the same set of events as experienced by a guest OS.

Hardware that hides, hurts

Recent hardware-assisted virtualization [37] has the potential to significantly reduce the overhead imposed by a VMM. The current implementation of hardware-assisted virtualization for the x86 architecture [3, 46] can, in some cases, hide information about certain events, like page faults and page table updates, from a VMM. This side effect inadvertently complicates some powerful software-based optimization opportunities, including some discussed in this dissertation. Other research [2] shows that current hardware virtualization can interfere with important features of a sophisticated software VMM and can ultimately reduce overall performance. Hence, hardware virtualization features must be carefully designed so that they do not unintentionally reduce the flexibility and power of a software VMM to employ code and workload-specific optimizations.

Portability does not imply guest-independence

We have built portable VMM-based services; the same implementation can be applied successfully to very different guest operating systems like Windows and Linux. The concrete results obtained under each guest, however, can vary substantially. For example, Linux kernel version 2.6 exhibits false positive process events not suffered by Windows or Linux 2.4. Creation lag under Linux 2.4 can be three orders of magnitude larger than under Linux 2.6. An application wishing to utilize implicit information within a VMM must take such platform variations into account.

Online statistical inference helps manage uncertainty

Variation and uncertainty in the form of false positive events and lag are a recurring theme in this dissertation. Elementary statistical inference techniques, like hypothesis testing, that can be applied continuously and automatically within a VMM can transform uncertainty due to variance into inferences about a guest that can be used with quantifiably high levels of confidence.

8.2 Future Work

The space of possible applications implemented at the VMM layer or with VMM assistance is large and has only begun to be explored. In this section we discuss some possible future avenues of inquiry relating to OS-aware services within a VMM beginning with tasks closely related to those we have already discussed and then wandering further afield.

8.2.1 Targeting Other Guest Abstractions

There are many other important guest operating system abstractions like threads, users, and network protocols that we have not considered in our work, but which could be useful when implementing services in a VMM. In the same way we have extended a VMM to observe the MMU and disk devices to infer information about OS processes and caches, a VMM can observe other virtual devices, like the microprocessor and network interface cards, to inform itself about additional OS abstractions.

8.2.2 Resource Association

We were able to show in Section 4.3 how a VMM can reliably associate disk read requests with their originating process. This is just one part of a much larger, general problem of logically connecting asynchronous guest OS events (like I/O requests) to implicitly observed entities (like processes, users, and threads). Our existing approach exploits the direction of data movement through memory toward a user process to identify its destination and make an association. The same technique cannot be used when data moves in the opposite direction. A new approach is required.

Other asynchronous requests like network sends and receives are also difficult to associate with a specific process from within a VMM. Does the typical approach to network protocol processing employed by most operating systems provide opportunities for efficiently associating network packets with a sending or receiving process? Accurate and early network packet association can be used, for example, to selectively implement novel VMM-based security features like process-specific filtering to protect applications from exploits targeting known vulnerabilities and taint tracking [41] to prevent network-based code injection attacks.

8.2.3 Observing Memory Structure

There are other sources of information about guest operating systems than the stream of service requests and fault notifications that we use in our work. One that seems particularly interesting is the content and structure of guest OS memory. An operating system's memory image is a collection of dynamic, interconnected structures. The organization and content of these structures can reveal a great deal about the current state of the guest OS. A VMM may be able to discover the location and the semantics of some of these structures on its own, without resorting to external information sources like debugging symbols.

For example, an OS process is typically represented in memory by a compound memory structure, often referred to as the *process control block*. The complete set of processes is usually represented by a dynamic, pointer-based collection of process control blocks like a list or a tree. By employing on-demand emulation [41] to selectively observe which memory addresses are accessed temporally close to process-related architectural events, like address space context switch, a VMM can identify many process management-related memory locations.

Compound memory structures are often accessed using base plus offset variants of load and store instructions which can be parsed to identify the base pointer of a structure and the offsets of commonly used individual fields. Each field accessed within a structure also has an implicit data type consisting of the size, range, and role of the memory operand (*e.g.*, small integer, bit vector, pointer). By combining field offsets and data types, an implicit compound type for a structure can be created and used to identify other structures of the same type in memory. By analyzing the data types of individual fields and how structures of similar type point to each other in memory it may be possible for a VMM to infer the organization and partial semantics of important dynamic data structures like the guest OS process list.

Chilimbi *et al.* [18], have employed similar techniques to discover invariants in application heap memory like the average pointer in-degree and out-degree of heap allocated data structures. They then use the invariants to discover bugs in pointer manipulating code. Petroni *et al.* [68], scan kernel memory from within a PCI device for violations of predefined correctness invariants using explicit, user-supplied kernel memory layout information. Using the approach described above, these techniques could be extended to allow a VMM to identify kernel data structures and check security invariants without requiring explicit implementation information. For example, a self-contained and independent VMM service could ensure that all active processes are linked into the system process list.

8.3 Closing Remarks

In this dissertation we have focused on implicitly obtaining and exploiting information about certain guest operating system abstractions. We assumed as our starting point the basic organization and division of labor between the VMM and guest operating systems that exist today. We have argued that the broad deployment of system virtualization suggests that certain OS features should migrate from the OS into the VMM. Our approach has been to implement these OS-like features within a VMM without changing the guest OS by using implicit information.

Operating systems and VMMs will change over time. The question of if and how the relationship between the operating system and the VMM should change is important. Which features currently implemented in the operating system would make more sense implemented within a VMM? How should the interfaces between the VMM and the OS change to facilitate communication without compromising the key desirable features of each?

The inclusion of a system virtualization layer as a core component in most system-level software represents an exciting and fundamental evolution. Finding an acceptable balance between isolation and cooperation among diverse operating systems and VMMs will require significant technical and political innovation. Until the perfect balance is discovered and adopted, there will be room for implicit methods like those we have described here.

Bibliography

[1] 90210. Bypassing klister 0.4 with no hooks or running a controlled thread scheduler. http://hi-tech.nsys.by/33/.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 2–13, San Jose, California, October 2006.

[3] AMD. Amd64 programmer's manual, volume 2: System programming. http://www.amd.com/us-en/assets/content_type/ white_papers_and_tech_docs/24593.pdf, December 2005.

[4] R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, , and R. C. Swanberg. Advanced virtualization capabilities of power5 systems. *IBM Journal of Research and Development*, 49(4):523–532, sept 2005.

[5] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[6] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing (Lake George), New York, October 2003.

[7] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[8] S. Ballmer. Keynote address. Microsoft Management Summit, April 2005.

[9] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, June 1969.

[10] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[11] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

[12] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[13] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, California, June 2002.

[14] J. Butler, J. L. Undercoffer, and J. Pinkston. Hidden processes: The implication for intrusion detection. In *Proceedings of the 2003 IEEE Workshop on Information Assurance*, pages 116 – 121, June 2003.

[15] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–138. IEEE Computer Society, 2001.

[16] Z. Chen, Y. Ahang, Y. Zhou, H. Scott, and B. Schiefer. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. In *Proceedings of the 2005 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, Banff, Canada, June 2005.

[17] Z. Chen, Y. Zhou, and K. Li. Eviction-based Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 269–282, San Antonio, Texas, June 2003.

[18] T. M. Chilimbi and V. Ganapathy. Heapmd: Identifying heap-based bugs using anomaly detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 219–228, San Jose, California, October 2006.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[20] J. Clemens. Knark: Linux kernel subversion. http://www.sans.org/resources/idfaq/knark.php.

[21] B. Cogswell and M. Russinovich. Pslist. http://www.microsoft.com/technet/ sysinternals/utilities/pslist.mspx.

[22] B. Cogswell and M. Russinovich. Rootkit revealer. http://www.sysinternals.com/Utilities/RootkitRevealer.html.

[23] R. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[24] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[25] P. J. Denning. Working Sets: Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[26] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 247–260, Kiawah Island Resort, South Carolina, December 1999.

[27] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[28] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[29] R. Figueriredo, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[30] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[31] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *OASIS ASPLOS 2004 workshop*, 2004.

[32] fuzen_op. fu.exe and msdirectx.sys. http://www.rootkit.com/vault/ fuzen_op/FU_README.txt.

[33] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, August 2004.

[34] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[35] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[36] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[37] P. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[38] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 317–332, San Francisco, California, December 2004.

[39] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, 1997.

[40] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[41] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 2006 ACM SIGOPS EUROSYS*, Leuven, Belgium, April 2006.

[42] Holy Father. HackerDefender. http://hxdef.org.

[43] G. C. Hunt, J. R. Larus, M. Abadi, P. Barham, M. Fahndrich, C. H. O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report 2005-135, Microsoft Research, 2005.

[44] ImageMagick Studio LLC. Imagemagick image processing software. http://www.imagemagick.org.

[45] InnoTek. Virtualbox virtual machine monitor. http://www.virtualbox.org.

[46] Intel Corporation. ftp://download.intel.com/ technology/computing/vptech/C97063.pdf, 2005.

[47] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.

[48] T. Johnson and D. Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[49] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, Boston, Massachusetts, June 2006.

[50] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, San Jose, California, October 2006.

[51] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. MemRx: What-If Performance Prediction for Varying Memory Size. Technical Report 1573, Department of Computer Sciences, University of Wisconsin-Madison, February 2006.

[52] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104. ACM Press, 2005.

[53] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, pages 364–373, Seattle, Washington, May 1992.

[54] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *IEEE Symposium on Security and Privacy 2004*, Oakland, CA, May 2004.

[55] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. In *IEEE Transactions on Software Engineering*, volume 17, pages 1147–1165, November 1991.

[56] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[57] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 314–327, Berkeley, California, USA, May 2006.

[58] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe, November 2005.

[59] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[60] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[61] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9, 1970.

[62] Microsoft. Microsoft virtual server. http://www.microsoft.com/windowsserversystem /virtualserver/default.mspx.

[63] Microsoft. Windows malicious software removal tool. http://www.microsoft.com/security/malwareremove.

[64] T. Miller. t0rn rootkit. http://www.ossec.net/rootkits/studies/t0rn.txt.

[65] P. Mulvany. Non-unix os history. www.oshistory.net.

[66] D. Muntz and P. Honeyman. Multi-Level Caching in Distributed File Systems - or - Your Cache Ain't Nuthin' But Trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.

[67] R. Naraine. Microsoft: Stealth Rootkits Are Bombarding XP SP2 Boxes. http://www.eweek.com/article2/0,1895,1896605,00.asp.

[68] J. Nick L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the USENIX Security Symposium*, pages 289–304, Vancouver, British Columbia, Canada, July 2006.

[69] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 311–324, San Antonio, Texas, June 2003.

[70] OSDL. Open source developement labs database test suite. http://www.osdl.org/lab_activities/kernel_testing/ osdl_database_test_suite.

[71] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, Colorado, December 1995.

[72] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, August 2004.

[73] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[74] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[75] F. L. Ramsey and D. W. Schafer. *The Statistical Sleuth: A Course in Methods of Data Analysis*. Duxbury Press, Boston, MA, 2nd edition, 2002.

[76] H. Reiser. ReiserFS. www.namesys.com, 2004.

[77] 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 USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

[78] J. Rutkowska. klister. http://www.invisiblethings.org/tools/klister-0.4.zip.

[79] J. Rutkowska. Subverting vista kernel for fun and profit. http://www.invisiblethings.org/papers/joanna rutkowska - subverting vista kernel.ppt.

[80] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[81] SANS Institute. Subseven trojan v 1.1. http://www.sans.org/resources/ idfaq/subseven.php.

[82] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 377–390, Boston, Massachusetts, December 2002.

[83] sd and devik. Suckit. Phrack #58, article 0x07.

[84] R. Sekar, T. F. Bowen, and M. E. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, pages 29–40, Berkeley, CA, USA, 1999. USENIX Association.

[85] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[86] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the Third USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, March 2004.

[87] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, San Francisco, CA, 1st edition, June 2005.

[88] F. G. Soltis. *Inside the AS/400*. 29th Street Press, October 1997.

[89] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

[90] Storage Performance Council. SPC web search engine storage traces. http://traces.cs.umass.edu/storage.

[91] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[92] Sun Microsystems. ZFS: The Last Word in File Systems. http://www.sun.com/2004-0914/feature/, 2004.

[93] Sun Microsystems. Sun consolidation and virtualization. http://www.sun.com/virtualization, 2007.

[94] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[95] A. Tridgell. Dbench filesystem benchmark. http://samba.org/ftp/tridge/dbench.

[96] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[97] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/ release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[98] V. Uhlig, J. Levasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM '04)*, pages 43–56, San Jose, California, May 2004.

[99] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 148–162, Brighton, United Kingdom, October 2005.

[100] A. Wald. *Sequential Analysis*. John Wiley & Sons, Inc., New York, NY, 3rd edition, September 1952.

[101] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[102] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, pages 368–377, June 2005.

[103] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[104] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[105] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, Massachusetts, April 2007.

[106] M. Zhao, J. Zhang, and R. Figueriredo. Distributed File System Support for Virtual Machines in Grid Computing. In *Proceedings of High Performance Distributed Computing (HPDC)*, July 2004.

[107] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamically Tracking Miss-Ratio-Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, Massachusetts, October 2004.

[108] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 91–104, Boston, Massachusetts, June 2001.