Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems

Xiaoxin Chen Tal Garfinkel E. Christopher Lewis Pratap Subrahmanyam Carl A. Waldspurger
Dan Boneh* Jeffrey Dwoskin† Dan R.K. Ports‡

VMware, Inc. *Stanford University †Princeton University ‡MIT

{mchen,talg,lewis,pratap,carl}@vmware.com dabo@cs.stanford.edu jdwoskin@princeton.edu drkp@mit.edu

Abstract

Commodity operating systems entrusted with securing sensitive data are remarkably large and complex, and consequently, frequently prone to compromise. To address this limitation, we introduce a virtual-machine-based system called *Overshadow* that protects the privacy and integrity of application data, even in the event of a total OS compromise. Overshadow presents an application with a normal view of its resources, but the OS with an encrypted view. This allows the operating system to carry out the complex task of managing an application's resources, without allowing it to read or modify them. Thus, Overshadow offers a last line of defense for application data.

Overshadow builds on *multi-shadowing*, a novel mechanism that presents different views of "physical" memory, depending on the context performing the access. This primitive offers an additional dimension of protection beyond the hierarchical protection domains implemented by traditional operating systems and processor architectures.

We present the design and implementation of Overshadow and show how its new protection semantics can be integrated with existing systems. Our design has been fully implemented and used to protect a wide range of unmodified legacy applications running on an unmodified Linux operating system. We evaluate the performance of our implementation, demonstrating that this approach is practical.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection

General Terms Design, Security, Performance

Keywords Virtual Machine Monitors, VMM, Hypervisors, Operating Systems, Memory Protection, Multi-Shadowing, Cloaking

1. Introduction

Commodity operating systems are ubiquitous in home, commercial, government, and military settings. Consequently, these systems are tasked with handling all manner of sensitive data, from individual passwords and crypto keys, to databases of social security numbers, to sensitive documents and voice traffic.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

Unfortunately, the security provided by commodity operating systems is often inadequate. Trusted OS components include not just the kernel but also device drivers and system services that run with privilege (*e.g.*, daemons that run as root in Linux). These components generally comprise a large body of code, with broad attack surfaces that are frequently vulnerable to exploitable bugs or misconfigurations. Once such privileged code is compromised, an attacker gains complete access to sensitive data on a system. While some facets of security in these systems will continue to improve, we believe competitive pressures to provide richer functionality and retain compatibility with existing applications will keep the complexity of such systems high, and their assurance poor.

To ameliorate this problem, many have attempted to retrofit higher-assurance execution environments onto commodity systems. Previous efforts have explored executing applications handling sensitive data in separate virtual machines [10, 29, 8], using secure co-processors [7], or changing the processor architecture to introduce orthogonal protection mechanisms that protect application data from the OS [6, 13, 16, 19, 27]. Unfortunately, these generally demand major changes in the way that applications are written [7, 8, 16, 18, 28] and used [8, 10], and how OS resources are managed [10, 29]. Such radical departures pose a substantial barrier to adoption.

We offer an alternative in a system called *Overshadow*. Overshadow protects legacy applications from the commodity operating systems running them. Unlike other approaches, it requires no changes to existing operating systems or applications, nor any additional hardware support. Instead, it works by extending the isolation capabilities of the virtualization layer to allow protection of entities inside a virtual machine.

Overshadow adds this protection through a novel technique called *multi-shadowing* which leverages the extra level of indirection offered by memory virtualization in a virtual machine monitor (VMM). Conceptually, a typical VMM maintains a one-to-one mapping from guest "physical" addresses to actual machine addresses. Multi-shadowing replaces this with a one-to-many, context-dependent mapping, providing multiple views of guest memory. Overshadow leverages this mechanism to present an application with a cleartext view of its pages, and the OS with an encrypted view, a technique we call *cloaking*. Encryption-based protection allows resources to remain accessible to the OS, yet secure, permitting it to manage resources without compromising application privacy or integrity.

Cloaking is a low-level primitive that operates on basic memory pages. However, nearly all higher-level application resources – including code, data, files, and even IPC streams – are already managed as memory-mapped objects by modern operating systems, or can be adapted as such. As a result, cloaking is sufficiently general to protect all of an application's major resources.

Using cloaking to protect a legacy application running on an unmodified OS requires some changes to the normal execution environment. To accommodate these changes while maintaining compatibility, Overshadow introduces a *shim* at load time into the address space of each cloaked application to mediate all communication with the OS. With assistance from the VMM, the shim interposes on events such as system calls and signal delivery, modifying their semantics to enable safe resource sharing between a cloaked application and an untrusted OS.

The next section presents our design goals and threat model for Overshadow. Section 3 reviews virtualized memory systems, and describes extensions to support multi-shadowing and cloaking. Section 4 introduces the challenges that arise when applying cloaking to protect real applications, and provides an overview of the Overshadow architecture. Section 5 describes how the shim and VMM adapt applications for a cloaked environment, and Section 6 explains how particular system calls are mediated. Section 7 discusses how cryptographic metadata is managed and stored, to protect against reordering and replay attacks. Section 8 evaluates our implementation in the VMware VMM using large unmodified applications running on an unmodified Linux kernel. Section 9 discusses future work. Related work is examined in Section 10, and we summarize our conclusions in Section 11.

2. Design Goals

Overshadow offers a last line of defense for application data in the event of an OS compromise. We begin with a discussion of why Overshadow targets whole-application protection, and the threats it attempts to address.

2.1 Whole-Application Protection

We were motivated to build a practical system that could be adopted easily, deployed incrementally, and used for diverse applications. As a result, we designed Overshadow to protect entire existing applications *in situ* in existing commodity operating systems. This approach has several advantages:

Ease of Adoption. Previous work on protecting applications requires partitioning an application into protected and unprotected parts – forcing developers to modify their applications heavily [8, 28] or port to a new OS [29]. Changes to how software is packaged and used may also be required [10, 29].

Support for Diverse Applications. Solutions for providing higher assurance are often restricted to a limited set of applications or data, such as passwords [8, 7]. However, sensitive data is remarkably diverse, from databases of credit card numbers, to files containing medical patient information. Sensitive data in real applications frequently doesn't lend itself to being placed in a neat separate container, and restructuring applications is often impractical.

Incremental Path to Higher Assurance. Even after taking the operating system out of the application's trusted computing base, large, complex applications will still have significant assurance concerns. Refactoring applications into more-critical and less-critical pieces running in separate protection domains [8, 28] is ultimately a compelling goal. Overshadow provides an incremental path to achieving this, as cloaking can be used for whole application protection as well as fine-grained compartmentalization.

2.2 Threat Model

Overshadow prevents the guest operating system from reading or modifying application code, data and registers, but makes no attempt to provide availability in the face of a hostile OS. All non-application access to cloaked data, including DMA from virtual I/O devices, only reveals the data in encrypted form. Data secrecy, integrity, ordering and freshness are protected up to the strength of the cryptography used. If the OS or other hostile code tries to modify encrypted data, the application will be terminated.

Control transfers to and from a cloaked application are permitted only at well-defined entry and exit points through mechanisms such as system calls and signal delivery. Application registers are also protected from the OS, and are saved and restored securely upon entry and exit from an application's execution context. Overshadow can also protect information shared between cloaked applications via the file system, shared memory or other forms of IPC.

A malicious kernel can still observe an application's memory access patterns, and measure the time that application code sections take to complete. In extreme cases, such side channel information can leak private information, including application crypto keys [15]. Overshadow does not protect against these side-channel attacks. However, most contemporary cryptographic application code (such as OpenSSL) is designed to resist side channel attacks.

Security is ultimately limited by the application being protected. Logical or semantic weaknesses in the application, such as an exploitable buffer overflow, or a `DumpMyMemory` command, could allow a malicious OS to fool it into revealing its data, or otherwise exploit it. The implications of maliciously changing the behavior of seemingly innocuous parts of the system call API, such as those for managing identity and concurrency, are still largely unstudied.

Assurance in Overshadow is ultimately limited by the VMM. While our current implementation uses the VMware VMM, a much simpler, high-assurance hypervisor could be used for running a single VM securely. Regardless, Overshadow offers a valuable additional layer of defense-in-depth. As its protection model is orthogonal to that of the guest OS, protected applications require no additional privileges within the guest.

We make no attempt to protect network I/O, as this is addressed by existing technologies such as SSL. Although a trusted path for user input and secure display is also desirable [8], and could be facilitated by Overshadow, we have not tried to support this in the current system.

3. Multi-Shadowed Cloaking

In this section we review how traditional virtualized memory systems work, and explain how they can be extended to support multi-shadowing. Multi-shadowing is then coupled with encryption to implement cloaking, providing both encrypted and unencrypted views of memory.

3.1 Classical Memory Virtualization

Conventional operating systems use page tables to map virtual addresses to physical addresses with page granularity. A virtual page number (VPN) is mapped to a physical page number (PPN), and VPN-to-PPN translations are cached by a hardware TLB.

A classical virtual machine monitor (VMM) provides each virtual machine (VM) with the illusion of being a dedicated physical machine that is fully protected and isolated from other virtual machines [24]. To support this illusion, physical memory is virtualized by adding an extra level of address translation. The terms *machine address* and *machine page number* (MPN) are commonly used to refer to actual hardware memory [4, 30]. In contrast, "physical" memory is a software abstraction that presents the illusion of hardware memory to a VM. We refer to address translation performed by a guest operating system in a VM as mapping a *guest virtual page number* (GVPN) to a *guest physical page number* (GPPN).

The VMM maintains a *pmap* data structure for each VM to store GPPN-to-MPN translations. The VMM also typically manages separate *shadow page tables*, which contain GVPN-to-MPN mappings, and keeps them consistent with the GVPN-to-GPPN mappings managed by the guest OS [1]. Since the hardware TLB caches direct GVPN-to-MPN mappings, ordinary memory references execute without incurring virtualization overhead.

3.2 Multi-Shadowing

Existing virtualization systems present a single view of guest "physical" memory, faithfully emulating the properties of real hardware. One-to-one GPPN-to-MPN mappings are typically employed, backing each guest physical page with a distinct machine page. Some systems implement many-to-one mappings to support shared memory; *e.g.*, transparent page sharing maps multiple GPPNs copy-on-write to a single MPN [4, 30]. However, existing virtualization systems do not provide flexible support for mapping a single GPPN to multiple MPNs.[1]

Multi-shadowing is a novel mechanism that supports context-dependent, one-to-many GPPN-to-MPN mappings. Conceptually, multiple shadow page tables are used to provide different views of guest physical memory to different *shadow contexts*. The "context" that determines which view (shadow page table) to use for a particular memory access can be defined in terms of any state accessible to the VMM, such as the current protection ring, page table, instruction pointer, or some other criteria.

Traditional operating systems and processor architectures implement hierarchical protection domains, such as protection rings [25]. Multi-shadowing offers an additional dimension of protection orthogonal to existing hierarchies, enabling a wide range of unconventional protection policies.

3.3 Memory Cloaking

Cloaking combines multi-shadowing with encryption, presenting different views of memory – plaintext and encrypted – to different guest contexts. Our use of encryption is similar to XOM [19, 18], which modified both the processor architecture and operating system to encrypt and isolate application memory. The term "cloaking" has also been used by Intel's LaGrande Technology (LT) [13], which introduced a different architectural mechanism for creating orthogonal protection domains.

In contrast to XOM and LT, our virtualization-based cloaking does not require any changes to the processor architecture, OS, or applications. In fact, cloaking based on multi-shadowing represents a relatively small change to the core MMU functionality already implemented by a VMM. We initially describe cloaking using a high-level model. Details concerning metadata management and integration with existing systems are presented in later sections.

Single Page, Encrypted/Unencrypted Views. We represent each GPPN using only a single MPN, and dynamically encrypt and decrypt its contents depending on the view currently accessing the page. This works well, since few pages are accessed simultaneously by both the application and the kernel in practice. As an optimization, the system could keep two read-only copies of the page, one encrypted, and one plaintext, for pages that are read concurrently from both views.

When a cloaked page is accessed from outside the shadow context to which it belongs, the VMM first encrypts the page, using a fresh, randomly-generated initialization vector (IV), then takes a secure hash H of this ciphertext. The pair (IV, H) is stored securely for future use. During decryption, the correct hash is first verified. If verification fails, the application is terminated. If it succeeds, the cloaked page is decrypted, and execution proceeds

[1] Some x86 VMMs do statically map a single GPPN to multiple MPNs to emulate the legacy `A20` line, for compatibility with real-mode applications. The `A20` line forces physical address bit 20 to zero, aliasing adjacent 1MB regions of memory.

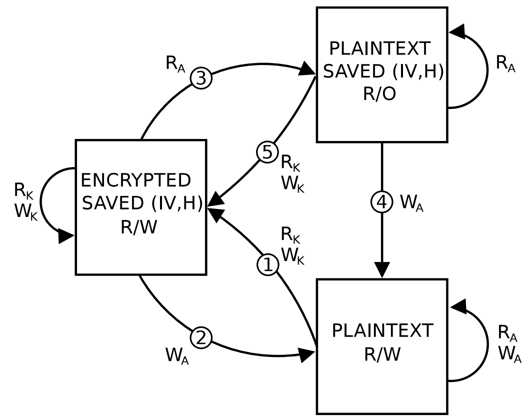

Figure 1. Basic Cloaking Protocol. State transition diagram for maintaining the secrecy and integrity of a single cloaked page. Application reads R_A and writes W_A manipulate plaintext page contents, while kernel reads R_K and writes W_K use an encrypted version of the page. A secure hash H is computed and stored immediately after page encryption, and verified immediately prior to page decryption.

as normal. By checking the hash before decryption, any attempts to corrupt cloaked pages will be detected.

Overshadow currently uses a single secret key K_{VMM} managed by the VMM to encrypt all pages; see Section 7.7 for details. Encryption uses AES-128 in CBC mode, and hashing uses SHA-256; both are standard constructions. An integrity-only mode could be supported easily, but is not part of the current implementation.

Basic Cloaking Protocol. Consider a single guest "physical" page (GPPN). At any point in time, the page is mapped into only one shadow page table – either a protected *application shadow* used by a cloaked user-space process, or the *system shadow* used for all other accesses. When the page is mapped into the application shadow, its contents are ordinary plaintext, and application reads and writes proceed normally.

Figure 1 presents the basic state transition diagram for managing cloaked pages. When the cloaked page is accessed via the system shadow (transition 1), the VMM unmaps the page from the application shadow, encrypts the page, generates an integrity hash, and maps the page into the system shadow. The kernel may then read the encrypted contents, *e.g.*, to swap the page to disk, and may also overwrite its contents, *e.g.*, to swap in a previously-encrypted page from disk.

When the encrypted page is subsequently accessed via the application shadow (transitions 2 or 3), the VMM unmaps the page from the system shadow, verifies its integrity hash, decrypts the page, and maps the page into the application shadow. For an application read (transition 3), the page is mapped read-only and its (IV, H) is retained. If the page is later written by the application (transition 4), the (IV, H) is discarded, and the page protection is changed to read/write. If the page is instead accessed by the kernel (transition 5), the VMM proceeds as in transition 1, except that the hash for the (unmodified) page is not recomputed.

The read-only plaintext state, where the (IV, H) is retained, is required to correctly handle the case where the kernel legitimately caches a copy of the encrypted page contents. For example, this could occur if the kernel swaps a cloaked page to disk, which is later paged in due to an application read, and then swapped out again before the application modifies it. The kernel can optimize the second page-out by noticing that the page is not dirty, and simply unmap the page without reading it, since the on-disk swapped

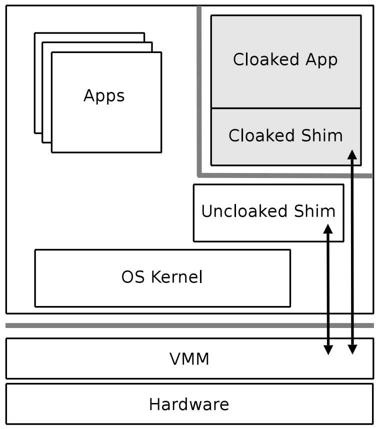

Figure 2. Overshadow Architecture. The VMM enforces two virtualization barriers (gray lines). One isolates the guest from the host, and the other cryptographically isolates cloaked applications from the guest OS. The shim cooperates with the VMM to interpose on all control flow between the cloaked application and OS.

copy is still valid. If the (IV, H) had been discarded, it would not be possible to decrypt the page after it is swapped back in.

Cloaking is compatible with copy-on-write (COW) techniques for sharing identical pages within or between VMs. Plaintext pages can be shared transparently, and page encryption handled like a COW fault.

Virtual DMA. Cloaking is also compatible with virtual devices that access guest memory using DMA. For example, suppose the guest kernel performs disk I/O on a cloaked memory page via a virtual SCSI adapter. For a disk read, the cloaked page contents are already encrypted on disk, and the VMM simply permits the kernel to issue a DMA request to read the page.

For a disk write, the action taken by the VMM depends on the current state of the cloaked page. If the page is already encrypted, the VMM allows the DMA to be performed directly. When the page is in the plaintext read-only state, the VMM first encrypts the page contents with its existing (IV, H) into a separate page that is used for the DMA operation. Similarly, if the page is in the plaintext read-write state, the VMM encrypts its contents into a separate page used for the DMA operation. The cloaked page then transitions to the read-only plaintext state, and is associated with the newly-generated (IV, H). Note that in both plaintext states, the original guest page is still accessible in plaintext form to the application, since a transient encrypted copy is used during the actual DMA.

4. Overshadow Overview

Cloaking is a low-level primitive that protects the privacy and integrity of individual memory pages. Overshadow leverages this basic mechanism to cloak whole applications, cryptographically isolating application resources from the operating system.

Figure 2 provides an overview of the Overshadow architecture. A single VM is depicted, consisting of a guest OS together with multiple applications, one of which is cloaked. The VMM enforces a virtualization barrier between the cloaked application and the OS, similar to the barrier it enforces between the guest OS and host hardware. Overshadow introduces a *shim* into the address space of the cloaked application, which cooperates with the VMM to mediate all interactions with the OS.

Realizing the Overshadow design goal of whole-application protection for unmodified applications running on unmodified commodity operating systems has proved challenging. In this section, we describe several key challenges, sketch high-level solutions, and explain where more complete technical details can be found in subsequent sections.

Context Identification. The VMM must identify the guest context accessing a cloaked resource precisely and securely, in order to use the shadow page table with the correct GPPN-to-MPN view. Section 5 explains how Overshadow leverages the shim to help identify application contexts, without relying on an untrusted OS.

Secure Control Transfer. Applications must interact with the OS to perform useful work, and need to be adapted for cloaked execution. Overshadow performs this adaptation by injecting a *shim* into the address space of each cloaked application. The VMM cooperates with the shim to implement a transparent trampoline that interposes on all control transfers between the application and OS. The detailed mechanics of shim-based interposition for interrupts, faults, and system calls are discussed in Section 5.

System Call Adaptation. Most system calls require only simple argument marshalling between cloaked and uncloaked memory. Others, such as file I/O operations, need more complex emulation. For example, `read` and `write` system calls are implemented using `mmap` for encrypted I/O. Section 6 explains how particular system calls are adapted for cloaked execution.

Mapping Cloaked Resources. Overshadow must track the correspondence between application virtual addresses and cloaked resources. The shim is responsible for keeping a complete list of mappings, which is cached by the VMM. The shim resides in the same guest virtual address as the application, and interposes on all calls that modify it, such as `mmap` and `mremap`. A more detailed discussion is presented in Section 7.

Managing Protection Metadata. The VMM must maintain protection metadata, such as (IV, H) pairs, for each encrypted page, to ensure privacy and integrity. For active mappings, the VMM maintains an in-memory metadata cache that is not accessible to the guest. Metadata associated with persistent cloaked resources, such as file-backed memory regions, is stored securely within the guest filesystem. Section 7 contains a detailed treatment of Overshadow metadata management.

5. OS Integration with Cloaking

The VMM interposes on transitions between the cloaked user-mode application and the guest kernel, using distinct shadow page tables for each. Privilege-mode transitions include asynchronous interrupts, faults, and signals, and system calls issued by the cloaked application. Mediating these interactions in a secure, backwards-compatible manner requires adapting the protocols used to interact with the operating system, as well as some system calls. This is facilitated by a small *shim* that is loaded into a cloaked application's address space on startup.

We describe the shim in the context of our Linux implementation, although we believe this approach could be applied to other operating systems, including Microsoft Windows. While the system call interface varies across kernels, low-level mechanisms for system call vectoring, fault handling, and memory sharing are tied more closely to the processor architecture than to a particular OS.

We begin by discussing the basic operation of the shim, how it helps the VMM manage identity, and its interaction with the kernel and VMM to adapt the application for cloaked execution. Support for handling faults, interrupts, and system calls is presented in detail. A discussion of how particular system calls are mediated is deferred until the next section.

5.1 Shim Overview

The shim is responsible for managing transitions between the cloaked application and the operating system. It uses an explicit *hypercall* interface for interacting with the VMM, *i.e.,* a secure communication mechanism between the guest and the VMM. This arrangement allows relatively complex operations, such as OS-specific system call proxying, to be located in user-mode shim code, instead of the VMM. It also facilitates extensibility, providing a convenient place to add custom or OS-specific functionality without modifying the VMM.

Shim Memory. In memory, the shim consists of both cloaked and uncloaked regions, each with its own distinct code, data and stack space. Each application thread has its own shim instance, and all thread-specific data used by the shim is kept in thread-local storage, preventing conflicts between different instances.

The cloaked shim is multi-shadowed like the rest of the application. It is responsible for tasks where trust is required to maintain protection, such as providing well-defined entry and exit points for control transfers, and moving data between cloaked and uncloaked memory securely. The cloaked shim also includes a *cloaked thread context* (CTC) page, which is set aside for the VMM to store sensitive data used for control transfers. This includes areas for saving register contents, a table of entry points to shim functions, and the identity of the shadow context containing the shim.

The uncloaked shim contains buffer space that provides a neutral area for the kernel and application to exchange uncloaked data. It also contains simple trampoline code to facilitate transitions from the kernel to cloaked code. Nothing in the uncloaked shim is trusted or necessary for protection. If its code or data is corrupted, it will merely cause the application to crash.

Hypercall Interface. The VMM exports a small hypercall interface to the shim. Uncloaked code is allowed to invoke operations to initialize a new cloaked context (used to bootstrap). It can also make calls to enter and resume cloaked execution. Since control can be transferred only to an existing cloaked context, these calls can be initiated safely by untrusted code. Cloaked code can make hypercalls to cloak new memory regions, unseal existing cloaked data, create new shadow contexts, and access other useful interfaces, such as metadata cache operations.

Loading Cloaked Applications. To start a cloaked application, a minimal *loader* program is run with the shim linked into a distinct portion of its address space. The actual loader is part of the shim; before taking steps to load the program, the shim must bootstrap into a cloaked context.

To create a new shadow context, the shim issues a hypercall with a pointer to itself and protection metadata containing hashes for all pages associated with cloaked code and data; see Section 7 for details. The VMM uses this metadata to verify its integrity, as the cloaked shim will have access to the address space of the cloaked application. Thus, to bootstrap a secure protection domain for the application, the shim must be trusted; *i.e.*, not malicious to the application. The call to create a new context also takes a pointer to a portion of thread-local storage in which the VMM can setup a new CTC. Once this setup is complete, the VMM transfers control to start execution in the cloaked shim.

The cloaked shim then runs its loading routine, which reads the application binary, and maps appropriate sections into memory. When creating anonymous memory regions or memory-mapping protected files, the shim performs hypercalls to cloak their corresponding virtual memory ranges. After the cloaked application has been loaded, it may launch additional programs. On a subsequent `execve`, if the target program is cloaked, the loader program is prepended to the `exec` call so that the new program will also be cloaked.

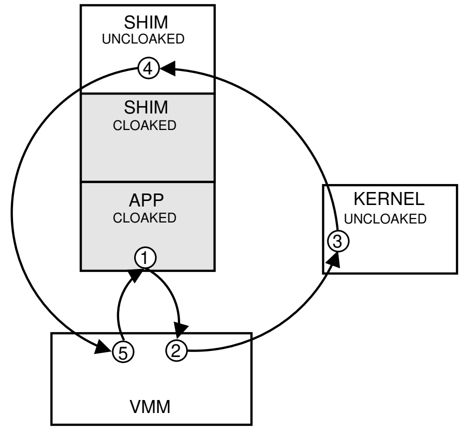

Figure 3. Control Flow for Handling Faults and Interrupts

Identity Management. To switch between shadow page tables appropriately, the VMM must employ some reliable procedure for identifying shadow contexts uniquely. Precise identification is challenging – contexts are associated with guest-level process abstractions, and scheduling is controlled by the OS, not the VMM. For example, the guest kernel may switch contexts while handling a fault or system call.

Existing approaches for VMM tracking of guest-level processes, such as monitoring assignments to the current page table root in Antfarm [14], work fairly well, but are not foolproof. Other schemes, such as accessing guest OS state at fixed kernel addresses (*e.g.*, Linux `current` pointer), or having the VMM store identifying information at some fixed virtual address, are generally fragile, or assume application pages can be pinned in physical memory. Most importantly, these approaches cannot be guaranteed to work in the presence of an adversarial OS. Overshadow takes an alternative shim-based approach that avoids these problems.

The VMM maintains a separate shadow context for each application address space, for which it assigns a unique *address space identifier* (ASID). Each address space may contain multiple threads, each with its own distinct cloaked thread context. When the shim begins execution, it makes a hypercall to initialize its CTC. During this initialization, the VMM writes the ASID and a random value into the CTC, and returns the ASID to the caller. The ASID value is not protected, and can be used by the uncloaked shim. However, since the CTC is cloaked, the random value is protected, and cannot be read by the uncloaked shim.

Shim hypercalls that transition from uncloaked to cloaked execution are self-identifying. The uncloaked shim passes arguments to the VMM containing its ASID, and the address of its CTC. The hypercall handler verifies that the CTC contains the expected random value, and also that its ASID matches the specified value. Note that the CTC resides in ordinary, unpinned application virtual memory. If the hypercall handler finds that the GVPN for the CTC is not currently mapped, it returns a failure code to the uncloaked shim, which simply touches the page to fault it back into physical memory, and then retries the hypercall.

5.2 Faults and Interrupts

While a cloaked application is executing, OS intervention is required to service faults or interrupts, such as application page faults and virtual timer interrupts. Figure 3 illustrates the flow of control for handling a fault from a cloaked application, involving the application, its associated shim, the guest kernel, and the VMM. The procedure for handling a virtual interrupt is essentially identical.

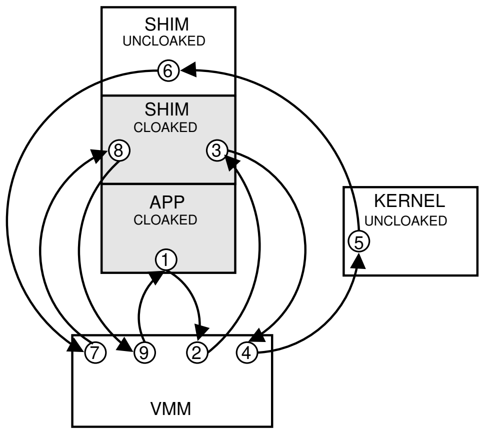

Figure 4. Control Flow for Handling System Calls

The fault occurs in step 1, and control is transferred to the VMM. In step 2, the VMM saves the contents of all application registers to the CTC in the cloaked shim. The VMM then zeros out the application's general-purpose registers to prevent their contents from being leaked to the OS. Next, the return instruction pointer (`IP`) and stack pointer (`SP`) registers are modified to point to addresses in the uncloaked shim, setting up a simple trampoline handler to which the kernel will return after servicing the fault. Finally, the VMM transfers control to the kernel.

The kernel handles the fault as usual in step 3, and then returns to the trampoline handler in the uncloaked shim setup in step 2. In step 4, this handler performs a self-identifying hypercall into the VMM to resume cloaked execution. In step 5, the VMM restores the registers saved in step 2, and returns control to the faulting instruction in the cloaked application.

Note that the active shadow page table must be switched when transitioning between uncloaked and cloaked contexts. Two shadow page table switches are required to handle a fault, in steps 2 and 5.

5.3 System Call Redirection

Unlike faults and interrupts, which are intended to be transparent to the application, system calls represent an explicit interaction between the cloaked application and the kernel. A system call is issued by the application using the standard OS calling convention. Figure 4 depicts the flow of control for handling a system call from a cloaked application, involving the application, its associated shim, the guest kernel, and the VMM. Note that the transitions involved in performing a system call are a strict superset of the transitions presented for handling a fault in Figure 3.

In step 1, the cloaked application performs a system call, and control is transferred to the VMM. In step 2, the VMM saves the contents of all application registers to the CTC in the cloaked shim. The `IP` is set to an entry point in the cloaked shim corresponding to a system call dispatch handler; similarly, the `SP` is set to a private stack in the cloaked shim for executing this handler. The VMM then redirects control to the dispatch handler in the cloaked shim.

In step 3, the cloaked dispatch handler performs any operations required to proxy the system call on behalf of the application. For some system calls, this may involve marshalling arguments, copying them to a buffer in the uncloaked shim. The dispatch handler then reissues the system call, substituting the marshalled arguments in place of the original application-specified values. As before, the VMM again intercepts the system call.

In step 4, the VMM saves the contents of all application registers in the CTC. Note that the CTC contains two distinct register save areas: one for the application registers saved earlier in step 2, and one for the shim registers saved in this step. The VMM then scrubs the contents of any application registers that are not required by the kernel system call interface. The return `IP` and `SP` are modified to point to addresses in the uncloaked shim, setting up a simple trampoline handler to which the kernel will return after executing the system call. Finally, the VMM transfers control to the kernel.

The kernel executes the system call as usual in step 5, and then returns to the trampoline handler in the uncloaked shim setup in step 4. In step 6, this handler performs a self-identifying hypercall into the VMM to enter cloaked execution. In step 7, the VMM restores the shim registers saved in step 4, and resumes execution in the cloaked dispatch handler.

The cloaked dispatch handler continues execution in step 8, performing any operations required to finish proxying the system call. For some calls, this may involve unmarshalling result values, and copying them into cloaked application memory. The dispatch handler then performs a hypercall into the VMM, requesting resumption of the cloaked application. In step 9, the VMM restores the application registers saved in step 2, and returns control to the instruction after the original system call in the application.

As in the case of fault handling, only two transitions require shadow page table switches between uncloaked and cloaked contexts, during steps 4 and 7.

6. Adapting System Calls

Cloaking necessarily changes the way the OS can manage process memory – it cannot modify it or introduce any sort of sharing without application help. It also changes the way the OS transfers control – it can only branch to well-defined entry and exit points within the application. Accommodating these changes requires adapting the semantics of a variety of system calls.

6.1 Pass-through and Marshalling

A majority of system calls can be passed through to the OS with no special handling. These include calls with scalar arguments that have no interesting side effects, such as `getpid`, `nice`, and `sync`. The shim need not alter arguments to these system calls, so the cloaked shim is bypassed altogether, resulting in control flow like that in Figure 3. Note that the VMM itself is not aware of system call semantics; during initialization, the shim simply indicates which system call numbers can be bypassed.

Many other calls have non-scalar arguments that normally require the OS to read or modify data in the cloaked application's address space, for example, path names and `struct sockaddrs`. Such arguments are marshalled into a buffer in the uncloaked shim, and registers are modified so the system call uses this buffer as the new source (or destination) for non-scalar data. After the system call completes, results are copied back into the cloaked application, if necessary. Implementing all this manually would be tedious and error prone, so we instead generate this code automatically from a simple specification, and the resulting code is used by the shim.

6.2 More Complex Examples

Several system calls require changes to resolve incompatibilities between cloaked semantics and normal OS semantics. We first describe system calls that require non-trivial emulation, and then discuss thread creation and signal handling.

Emulation. We are forced to emulate the semantics of several system calls. For example, `pipe` normally creates a queue in the kernel for communicating bytes. We cannot easily protect this, so instead we emulate a pipe between cloaked applications with a queue in cloaked shared memory. To preserve the normal blocking semantics of calls such as `read`, `write`, and `poll`, reads and writes are performed over the pipe as normal, except that the sender sends zeros instead of actual data. For the receiver, zeros are read, then actual data is copied from the protected queue. Support for `futex` (Linux fast mutex) calls is another example of where emulation is required, as the normal OS implementation involves direct access to process memory.

Thread Creation. Handling the `clone` and `fork` system calls is particularly interesting, since these are intimately related with how the shim manages resources. A `clone` call begins by allocating thread-local storage for the new thread. Next, the child's cloaked thread context (CTC) is setup by making a copy of the parent's CTC, and fixing all thread-local pointers for the child. Finally, it changes the `IP` and `SP` for entering cloaked mode in the child's CTC, arranging for the child to start executing in a `child_start` function located in the child's shim, which will complete its initialization.

Normally, the CTC would be modified by the VMM on a switch from cloaked to uncloaked mode. However, in this case, the child's CTC is not currently being used. Thus, on a `clone` system call, only the parent's CTC is modified. We also setup the uncloaked stack that will be used by the cloned thread when returning from the system call, so that it will start running the new cloaked context. After returning from the system call, the parent thread returns to the original execution context. The child thread begins execution in `child_start`, as described above.

Signal Handling. Normal Unix signal-handling semantics are incompatible with cloaking, as we cannot allow the operating system to transfer control into an arbitrary section of cloaked code. Keeping portions of the shim non-preemptible also simplifies its implementation.

When the application registers a signal handler with `signal`, the shim emulates it, registering the handler in its own table. All actual signal handlers (those registered with the kernel) use a single handler located in the uncloaked shim. This signal handler makes a hypercall to the VMM immediately upon receiving a signal, indicating which shadow context received the signal, the signal that occurred, and any additional signal parameters.

The VMM examines the cloaked context and checks the signal status to determine in which context the signal occurred: the cloaked shim, uncloaked shim, cloaked application, or other uncloaked code. If the signal occurred when the cloaked application was executing, the VMM transfers control to a well-defined signal entry point in the shim, with relevant signal information. If the signal occurred while the shim was executing, the VMM further checks a flag in the CTC to determine whether to safely rollback execution to the last application system call entry point, or to defer the signal delivery until shim exit, when execution has effectively returned to the application.

6.3 File I/O

Extending Overshadow's cryptographic protection to files on disk requires interposing on I/O related system calls. Unprotected files are handled using simple argument marshalling, while protected files must be adapted to utilize cloaking.

Encrypted file I/O for cloaked applications is implemented in the shim using `mmap`. For example, `read` and `write` system calls are emulated by copying data to/from memory-mapped buffers. File data is always mapped using the `MAP_SHARED` flag, to ensure that other processes that may open the same file obtain a consistent view. By transforming all file I/O into memory-mapped I/O, file data is decrypted automatically when it is read by a cloaked application, and encrypted automatically when it is flushed to disk by

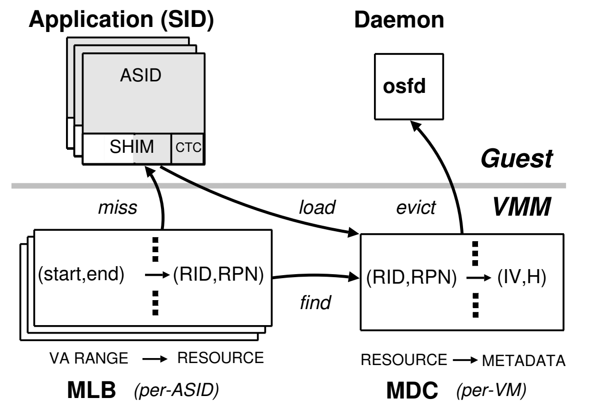

Figure 5. Protection Metadata Management

the kernel. To allow the VMM to protect integrity and ordering of file data, the shim may need to load protection metadata from disk when the file is opened; this is described in detail in Section 7.6. For efficiency, the shim maintains a cache of mapped file regions; our current implementation maps regions using 1MB chunks to amortize the cost of the underlying `mmap` and `munmap` calls.

Using `mmap` for file I/O obviates the need to implement any cryptography in the shim. Also, encryption and decryption are performed only when necessary. An application can `read` and `write` portions of a file repeatedly without causing additional decryptions. Similarly, data is only encrypted when the OS flushes it to disk.

A single-page Overshadow header is prepended to each cloaked file. This header contains the actual file size, which may differ from the current on-disk size due to the 1MB mapping granularity. Each shim using the file maps its header using a shared `mmap`, to properly emulate operations such as `fstat` and `lseek`. The shim also tracks all operations that create or manipulate file descriptors, such as `dup`, and maintains a table of all open files, their offsets, and whether they are cloaked. This table is kept in a shared anonymous region to properly track and share descriptors across process forks.

7. Managing Protection Metadata

Overshadow introduces OS-neutral abstractions for cloaking both persistent and non-persistent resources, such as files and private memory regions. For each resource, *protection metadata*, such as (IV, H) pairs, must be managed to enforce privacy and integrity, ordering, and freshness (to prevent rollback). Figure 5 provides an overview of the components involved in metadata protection. We begin by examining how metadata is stored and mapped to protected objects, then consider how it is used to enforce protection.

7.1 Protected Resources

Each cloaked resource, such as a file or anonymous memory region, is associated with a unique 64-bit *resource identifier* (RID). Each RID has a corresponding resource metadata object (RMD) that stores metadata needed to decrypt, check integrity, and preserve ordering. Concretely, an RMD is an ordered set of (IV, H) pairs, one per encrypted page, addressed by a 32-bit *resource page number* (RPN). In our current system, each RMD is implemented with a data structure similar to a three-level page table to efficiently support large, potentially-sparse address spaces, up to 256GB.

When a resource is mapped into memory, its RMD is loaded into the *metadata cache* (MDC) in the VMM. A single MDC caches metadata for all cloaked resources mapped by the guest.

This design ensures metadata consistency for shared objects, such as files and shared memory regions. When a resource is not in use by any process, its RMD is stored on disk in a metadata file. The MDC provides primitive operations to get, set, and invalidate metadata entries, as well as higher-level operations for cloning and persisting metadata, described later in this section.

7.2 Protected Address Spaces

Access control and sharing for cloaked resources are determined strictly by a unique *security identifier* (SID) that identifies an Overshadow protection domain. In the current implementation, a SID is associated with an application instance, which may contain multiple processes. Processes with the same SID have common access to cloaked resources. The address space for a cloaked process is identified by a unique *address space identifier* (ASID) that defines its shadow context. Portions of multiple cloaked resources are typically mapped into the guest virtual address space associated with a given ASID.

The VMM maintains a per-ASID cache of resource mappings in its virtual address space, called the *metadata lookaside buffer* (MLB). The MLB is used to map a virtual address to a resource. An MLB entry has the form (*start*,*end*) $\mapsto$ (RID, RPN), where *start* and *end* denote the virtual address range into which the resource is mapped, RID denotes the resource being mapped, and RPN denotes the first RPN in the mapping. For example, if file `foo.txt` has RID 4, and its third page is mapped into the first GVPN in the virtual address space, this is modeled as $(0, 4096) \mapsto (4, 2)$.

The shim is responsible for keeping a complete list of resource mappings for both cloaked and uncloaked memory, updating the MLB on any change. The shim resides in the same guest virtual address space, and interposes on all calls that modify it, such as `mmap`, `munmap`, and `mremap` in Linux; more details appear in Section 5. By delegating this responsibility to the user-mode shim, the VMM implementation is kept simple and OS-neutral.

On an MLB miss, the VMM performs an upcall into the shim to obtain the required mapping, and installs it in the MLB, illustrated by the *miss* action in Figure 5. The mappings for the shim itself are pinned in the MLB, preventing recursion. Note that even if some bug caused the MLB to have an incorrect mapping, it generally *fails-closed*; the wrong address range or cloaking status will cause decryption to fail, or the application will end up accessing ciphertext, causing it to fail.

7.3 Page Decryption

When a process accesses a cloaked page in its shadow context, its ASID and GVPN are known. If the page is unencrypted, then the memory access proceeds normally, without any VMM intervention.

If the page is encrypted, the access will fault into the VMM, since the GVPN is not mapped into the shadow for that ASID. The VMM looks up the faulting address in the MLB, and uses the resulting (RID, RPN) to index into the MDC and fetch the (IV, H) needed to decrypt and integrity check the page contents; see the *find* operation in Figure 5. The hash, check, and decrypt steps are performed using the protocol described previously. If the decryption succeeds, and the page is marked writable, (RID, RPN) is invalidated in the MDC. The page is then *zapped*, *i.e.*, removed from all shadows, and mapped into the current shadow for the ASID. The original application access is then allowed to proceed.

There is one special case. Operating systems commonly zero the contents of a page before mapping it into userspace, and applications depend on this initialization. If an access is made to a GVPN that is not mapped in the current shadow, and the (RID, RPN) for that page is not in the MDC, then this must be the first application access to the page, and no decryption is necessary. We check that the page contents are indeed zero-filled, and assuming this succeeds, the page is simply zapped and then mapped into the current shadow, and the original memory access is allowed to proceed.

Finally, the VMM stores the (RID, RPN) used for each decryption with the associated GPPN in the existing VMM pmap structure which stores GPPN-to-MPN translations.

7.4 Page Encryption

When the guest kernel (or any context that doesn't match the application SID) accesses a cloaked page, its GPPN is known, but its ASID and GVPN may not be known. The access could originate from any guest context, *e.g.*, during a virtual DMA operation. If the page is already encrypted, then the memory access proceeds normally, without any VMM intervention.

If the page is unencrypted, the access will fault into the VMM, since it is not mapped in the current shadow. If the page is writable, the VMM generates a new random IV; for a read-only page, the existing IV is reused. The VMM then encrypts the page contents, and computes a secure hash H over the encrypted contents. It stores the resulting (IV, H) in the MDC, at the (RID, RPN) previously associated with the GPPN in the pmap during its last decryption. The page is then zapped and mapped into the current shadow, and the original kernel access is allowed to proceed.

7.5 Cloning Metadata

The MDC also provides operations to facilitate support for address space cloning, such as `clone` or `fork` in Linux. Suppose a cloaked process forks a child. Immediately after the fork, the parent and child processes share their private memory regions copy-on-write (COW). Overshadow must ensure that the metadata associated with all unmodified COW pages remains accessible and synchronized between the parent and child.

When the fork occurs, each of the parent's private RMDs is cloned eagerly for the child, by copying all of its existing metadata entries, and assigning it a new RID. This ensures that metadata for any pages encrypted prior to the fork remain available to the child, even if the parent later modifies them.

However, suppose the parent encrypts a COW-shared page after the fork; a subsequent access by the child would not find the metadata required for decryption. One approach is to forcibly encrypt all pages in the parent during the fork, but this would be extremely inefficient, since few private pages remain encrypted in practice, unless the system is swapping heavily. Another option is to store a complete backmap for every GPPN, containing all (ASID, GVPN) pairs that map it, but this would be extremely complex.

The solution we implemented is to mirror the application's process tree in the MDC; each RMD has pointers to its parent, first child, and next sibling RMDs, if any. The MDC also maintains a global 64-bit version number, which is incremented on every RMD creation and page decryption. A version is stored with each RMD, set to the global version when it is created. Similarly, a version is stored along with the (RID, RPN) in the pmap for each GPPN, and set to the global version each time it is decrypted. Whenever a page is encrypted, the (IV, H) is stored at the (RID, RPN) associated with the GPPN, and also recursively propagated to any child RMDs with versions greater than the GPPN's version. Thus, metadata is propagated to all children with pages whose contents existed prior to the fork, as desired. A subtle point worth noting is that when the parent modifies a COW page, it will be encrypted (and its metadata propagated to the child) prior to the modification, since the guest OS must first read the page to make a private copy for the parent during the COW fault.

7.6 Persisting Metadata

RMDs associated with non-persistent memory regions (*e.g.*, application stack, data, or anonymous shared memory), can be discarded when no longer in use. However, RMDs associated with persistent content, such as file-backed memory regions, must be saved to disk. Each cloaked file has an associated metadata file in the guest for storing its RMD persistently. Metadata file integrity is protected by a message authentication code (MAC) stored in the file, computed using a key derived from the VMM's secret key K_{VMM}. The current implementation uses HMAC with SHA-256.

When a process opens a cloaked file, the shim makes a hypercall to determine if the metadata for its RID is in the MDC. If the metadata is not present, the shim performs a hypercall to allocate a new RMD for that RID, reads the entire metadata file, and passes its contents to the VMM, which verifies its integrity, as illustrated by the *load* action in Figure 5. Frequently reloading the RMD or recomputing its MAC might raise efficiency concerns. This can be optimized by keeping RMDs cached longer in the MDC, instead of evicting them eagerly after they have been committed to disk. Another option would be to store MACs in a Merkle hash tree [21], allowing for more efficient verification and updates.

To ensure freshness, a 128-bit generation number is also written to the metadata file, and protected by the MAC. The VMM checks this number against a master list of valid generations when the file is loaded. This number is stored in the MDC as part of the RMD. Just prior to eviction, it is incremented in both the RMD and master list. The master list is stored in the guest, protected by a MAC and its own counter which is stored outside of the guest by the VMM.

RMDs are written to metadata files by the Overshadow file daemon (`osfd`). The `osfd` communicates with the VMM via a simple hypercall interface, polling for metadata that should be evicted from the MDC and persisted to disk. The daemon extracts the metadata for all of its valid RPNs, obtains their MAC as generated by the VMM, commits everything to disk, and finally evicts the RID from the MDC; refer to the *evict* action in Figure 5. Notably, the `osfd` daemon is not trusted, and all data it handles is protected cryptographically. Its compromise would sacrifice only system availability, not data privacy or integrity.

7.7 Key Management and Access Control

Our usage model during Overshadow development has been to set up a clean VM and cloak an unmodified application in place. However, one could easily use our tools from outside the VM to convert existing Linux packages (*e.g.*, `rpm` files) by encrypting their files, and adding corresponding metadata files to the package.

Given the simple primitives in our architecture, a wide range of access control policies could be supported, as SIDs provide a basic primitive for identifying subjects, and RIDs provide a basic primitive for identifying objects. We currently use a simple model that assumes mutual trust between all parts of an application and dynamically assigns SIDs at startup.

Our current implementation performs all encryption using a single set of encryption and MAC keys. It is important to note that key management and access control in Overshadow are orthogonal. The VMM arbitrates who is allowed to access what resources, regardless of the key with which it was encrypted. Additional keys could be added to support delegation of administrative tasks; *e.g.*, a key per RID would allow different parties to package their own sets of encrypted files outside of the VM.

8. Evaluation

The current Overshadow implementation realizes the full system described in earlier sections. It supports cloaking for all application memory regions – private and shared, anonymous and file-backed. We demonstrate that the system is practical by presenting quantitative results for experiments running substantial, unmodified applications on an unmodified Linux operating system.

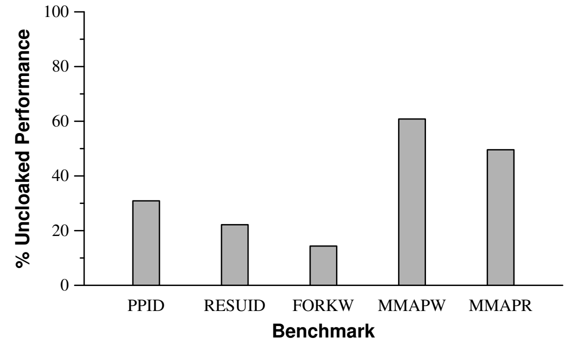

Figure 6. Microbenchmarks. Percentage of uncloaked performance attained with full cloaking of all memory regions and files.

8.1 Implementation

The Overshadow implementation is based on a version of the VMware VMM for 32-bit x86 processors that uses binary translation for guest kernel code [1]. The modified VMM was built as a VMware Workstation binary running in a "hosted" configuration on top of an existing Linux host OS.[2] Since multi-shadowed memory cloaking does not depend on specific features of the VMware VMM, it could also be realized in other virtualization platforms.

Our VMM modifications included approximately 4600 new lines of code, plus 2000 additional lines from publicly-available cryptographic routines. The shim handles nearly all system calls supported by the Linux 2.6 kernel interface, and is sufficiently complete to run large, unmodified Linux programs. The shim consists of 13,100 lines of code, including roughly 8500 lines of new code, and 4600 lines of standard library and utility routines.

Changes would be required to enable hardware-assist for x86 virtualization, such as Intel VT [22] and AMD SVM [2]. For example, system call transitions between guest user-mode and kernel-mode are always trapped by a binary-translating VMM, but are not typically trapped by a hardware-assisted VMM. Forcing system calls to trap for Overshadow interposition would likely introduce additional overhead. Nevertheless, we expect that hardware support for nested page tables will accelerate many Overshadow operations, improving overall performance. Reducing the cost of hardware context switches is also desirable. For Overshadow, the ability to redirect a trap to guest user-mode code would be ideal, making it possible to redirect system calls to handlers in the shim without dynamic VMM intervention.

8.2 Performance

All experiments were conducted on a Dell Precision 390 host configured with a 2.66GHz Intel Core2 Duo processor and 4GB RAM. The VM was configured with one CPU and 2GB memory running an unmodified Fedora Core 7 guest OS (Linux 2.6.21-1 kernel).

Microbenchmarks. Figure 6 presents the results of microbenchmarks that measure the overhead of system call redirection and cloaking. Each data point plots the ratio of cloaked to uncloaked performance, using the mean over the best 5 of 7 trials for both. In these experiments, all the benchmarks exhibited low runtime variability (standard deviation within 2.4% of the mean).

The PPID and RESUID benchmarks measure raw system call overheads, for both `getppid`, implemented using pass-through,

[2] In this configuration, the VMM is not fully protected from the Linux host OS. Secure deployment of Overshadow would require running the VMM directly on hardware, like VMware ESX Server [30], Xen [3], or IBM z/VM.

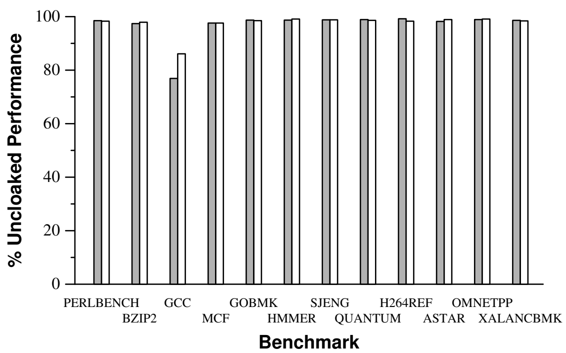

Figure 7. SPEC CINT2006 Benchmarks. Percentage of uncloaked performance attained with full cloaking of all memory regions and files (gray), and cloaking anonymous memory regions only (white).

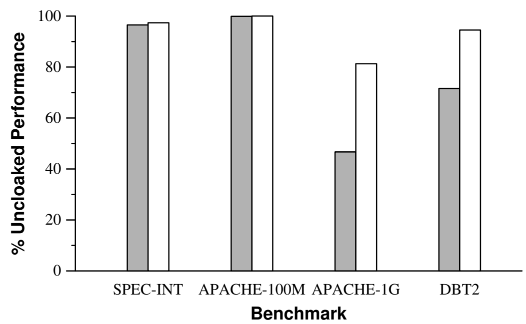

Figure 8. Benchmark Summary. Percentage of uncloaked performance attained with full cloaking of all memory regions and files (gray), and cloaking anonymous memory regions only (white).

and `getresuid`, requiring simple argument marshalling. Cloaking clearly increases system call costs significantly (by a factor of 3 to 5), primarily due to the extra pair of address-space switches on transitions between cloaked and uncloaked guest contexts.

The FORKW benchmark highlights the overhead of process creation, destruction, and synchronization using `fork` and `wait`. Cloaking introduces overhead due to encryption and decryption for copy-on-write (COW) pages, as well as execution in the shim handler and VMM during process creation.

The MMAPW benchmark measures the cost of writing one word to each page in a large file-backed memory region, and flushing the data to disk. The cloaking overhead is dominated by the cost of encryption during disk write operations. MMAPR measures the cost of reading one word from each page of a large memory region backed by the file written by MMAPW. This benchmark incurs page faults, but does not perform disk I/O, as pages containing the file data still reside in the guest buffer cache. In this experiment, cloaking does not cause decryptions because these pages remain accessible to the application in plaintext form during virtual DMA (see Section 3.3). The results indicate that cloaking approximately doubles the cost of a minor page fault.

Application Benchmarks. Figures 7 and 8 present results from the SPEC CPU2006 integer suite and aggressively-loaded Linux web and database servers. All data points are averages over at least three trials. Despite high overheads on some microbenchmarks (Figure 6), real applications perform additional work that amortizes these costs.

Figure 8 plots the geometric mean for the entire SPEC suite, showing that overall the SPEC benchmarks incur very little overhead from cloaking. When we consider the SPEC benchmarks individually (Figure 7), only GCC has non-trivial overhead. This overhead comes from GCC's relatively high system call and page fault rates.

The web server experiment used the standard prefork configuration of APACHE 2.2.4, with caching disabled. A remote host generated client requests for fetching a 28 KB HTML file using the `ab` benchmarking tool with 50 concurrent connections. The client and server were connected by a 100Mbps (APACHE-100M) or 1Gbps (APACHE-1G) switch. We measured the total number of requests served per second. With full cloaking, performance for APACHE-100M was within 1% of uncloaked performance. When using the 1 Gbps switch (APACHE-1G), fully-cloaked performance degraded relative to the uncloaked server. These results are explained by the fact that cloaked applications have higher CPU occupancy; for the more realistic load (few web servers saturate 100 Mbps Internet links), the processor was not fully utilized, and cloaking didn't affect performance. With the network bottleneck removed, cloaking scales well until it saturates the CPU. Naturally, this saturation point will occur later for multicore VM configurations (our experiments utilize only one processor).

The database workload uses the DBT2 transactional database performance test suite running with a PostgreSQL 8.2.4 server. This test simulates a wholesale parts supplier with 22 warehouses and 11 concurrent clients at the peak throughput; the clients run uncloaked in the same VM as the server. We measured the number of "new order" transactions per minute during steady-state operation, the standard metric from this suite. With full cloaking enabled for this 8.6 GB database, performance was more than 70% of the uncloaked baseline.

While there are ample opportunities for optimization, the current implementation of full cloaking is practical for many realistic workloads. For applications that require only anonymous regions to be cloaked, performance is uniformly above 80% of the baseline.

9. Future Work

A variety of interesting research opportunities remain in the areas of retrofitting protection to legacy operating systems, and leveraging multi-shadowing to define new protection models. We also describe extensions to our current implementation.

Retrofitting Protection. Applications are not designed with the expectation that the operating system can become hostile. A more formal treatment of how an OS might mislead an application – and how such attacks can be mitigated – is an interesting topic for future research. For example, an application might be misled into revealing information if it is run with a particular `uid`. One possible defense is to provide a "reverse sandbox" that filters system calls to prevent such attacks.

We are also investigating a trusted path for user interface devices, as this would enable complete protection of many compelling applications, including web, email, and VOIP clients. In principle, user interaction could be protected in the current implementation if the application uses a remote display system that renders to software frame buffers.

Protecting Device Memory. Many I/O devices present a memory-mapped interface to software. For some devices, multi-shadowing can be employed to protect the contents of "physical" device memory from being inspected or modified by untrusted software. For example, an interactive VM typically provides a virtual high-resolution graphics display that uses a memory-mapped frame buffer. A multi-shadowed frame buffer could help implement a trusted path, by ensuring that a cloaked application's output remains private. While this approach can be used to prevent the operating system from observing raw device memory, additional work is needed to cloak off-screen display images and other memory used by window managers and graphics subsystems.

Fine-Grained Cloaking. Applications can be modified to apply multi-shadowing selectively, cloaking only sensitive pages. For example, two shadow contexts could be defined for each application: a protected shadow containing cloaked code and data, and an unprotected shadow for uncloaked code and data. In this simple model, cloaked memory can be accessed only by cloaked code. A shadow context is identified by the virtual address of the current instruction pointer.

In order to interpose on transitions between these shadow contexts, a VMM can change the execute permission of pages in the shadow page tables (independent of guest PTE permissions). In the unprotected shadow, all protected pages are marked non-executable; similarly, in the protected shadow, all unprotected pages are marked non-executable. When the application branches between protected and unprotected code, the resulting permissions-based page fault will trap into the VMM, allowing it to switch between shadow page tables.

Storage Extensions. Our current implementation has a few storage-related limitations. First, the RID for a file is simply constructed from its device and inode numbers; this is problematic on network file systems where uniqueness can't be expected.

Next, Overshadow currently offers no protection for file system metadata; consequently, the OS could maliciously swap inputs on an application. A simple solution is to provide a secure namespace, associating pathnames with (RID, MAC) pairs. This could be implemented by employing a protected daemon or shared file, which would be updated on file operations such as `rename`, `create`, and `unlink`. More sophisticated approaches have been explored by others [11, 9, 17].

Finally, we currently don't maintain consistency between file system data and metadata in the event of a system crash. If the guest OS crashes before we commit the metadata for a given file, or before the data for a given file is committed to disk, we could end up in a state where data and metadata are out of sync. We believe all of these issues are tractable, but a full treatment remains a topic for future work.

10. Related Work

Memory virtualization enables transparent remapping of guest physical pages. Previous systems have leveraged this ability for transparent swapping [4, 30], transparent page sharing [4, 30], transparent page migration across NUMA nodes [4], and transparent VM migration across physical hosts [5, 23]. Overshadow takes advantage of this extra level of indirection to provide isolated, context-dependent views of guest physical memory.

Many prior systems have attempted to tackle the problem of providing a higher-assurance execution environment on commodity platforms. All have aimed to eliminate the need to trust commodity operating systems with the security of sensitive data.

Intel's LaGrande Technology (LT) [13] provides hardware mechanisms for isolating a portion of a machine's address space to create orthogonal protection domains within the guest. The NGSCB [8] (formerly Palladium) architecture proposed using this functionality to split commodity systems into low-assurance and high-assurance partitions. The low-assurance partition would run a commodity operating system (Windows); the other, a simpler trusted operating system (the Nexus). Applications would correspondingly be split into a small trusted part (the agent, run under the Nexus) and the untrusted part, run on the commodity OS.

Proxos [29] takes a similar, but more backwards-compatible approach. It splits the system into multiple VMs, one running an untrusted commodity OS, the other(s) running trusted, application-specific operating systems. Sensitive applications are run in a trusted VM, but still interact with resources in the untrusted VM via a process that proxies system calls for it, manipulating resources on its behalf. The Terra [10] architecture proposes moving the entire application into a separate VM with its own application-specific software stack tailored to its assurance needs. While Overshadow takes an OS-level approach to application protection, unlike all of these earlier systems, it does not introduce any additional resource management mechanisms or new operating systems.

Retrofitting protection via an encryption layer is a familiar concept from networking. In storage, systems like SUNDR [17] and Sirius [11] have examined securing block and file-level storage on untrusted substrates, and similar work exists for databases [20]. However, this approach is rarely encountered in the OS literature, with the exception of architecture-level research such as XOM [18, 19] and SP [6, 16]. XOM and SP also provide a dual encrypted-unencrypted view of memory, like Overshadow, though they achieve this through custom processor architectures, and target a threat model where hardware attacks are possible, *i.e.*, main memory is untrusted.

Overshadow considers only software attacks, but works with off-the-shelf hardware. Compared to architecture-level approaches, Overshadow also gains substantial flexibility by being software-based. XOM requires applications and/or the OS to be substantially modified or rewritten [18]. SP also requires applications to be rewritten, explicitly specifying which code and data to protect. While SP does not need OS modifications, it supports only one protection domain per device [16]. In contrast, Overshadow makes integration with unmodified operating systems and applications feasible, and enables sharing between protection domains. Nevertheless, Overshadow's software mechanisms could be combined with more hardware-centric approaches to provide similar benefits.

We have developed Overshadow as a means of enhancing security in commodity systems, where redesigning applications and using a high-assurance OS [12, 26] is not an option. However, we believe cloaking is useful as a more general OS abstraction, with novel properties not afforded by normal memory protection. In particular, cloaking provides an OS-level analog to end-to-end encryption in networks, eliminating the need to trust those pieces of the system that are merely responsible for moving data from one place to another, versus those that are actively using that data.

11. Conclusions

We have presented *Overshadow*, a system that cryptographically isolates an application inside a virtual machine from the operating system it is running on, offering another layer of protection for application data, even in the face of total OS compromise.

This capability is enabled by *multi-shadowing*, a novel technique for presenting different views of "physical" memory in virtualized systems. This allows memory to be *cloaked*, so that it appears normal to an application, but encrypted to the operating system. Cloaking supports a separation of responsibilities for isolation and resource management, allowing the use of a complex commodity operating systems to manage application virtual memory and other resources, while relying on a much simpler hypervisor to ensure data privacy and integrity.

Unlike previous approaches to enhancing assurance in commodity systems, Overshadow is backwards-compatible, protecting a broad range of unmodified legacy applications, managed by unmodified commodity operating systems. While Overshadow is not a panacea, we believe it demonstrates a promising approach to enhancing data security in commodity computing environments.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, October 2006.

[2] AMD. *AMD64 Virtualization Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 143–156, October 1997.

[5] 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 Second Symposium on Networked Systems Design and Implementation*, pages 273–286, May 2005.

[6] J. S. Dwoskin and R. B. Lee. Hardware-rooted Trust for Secure Key Management and Transient Trust. In *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, pages 389–400, October 2007.

[7] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S. Smith, L. van Doorn, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34:57–66, October 2001.

[8] P. Englund, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *IEEE Spectrum*, pages 55–62, July 2003.

[9] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-only File System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[10] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[11] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Network and Distributed System Security Symposium*, pages 131–145, February 2003.

[12] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proceedings of the International Conference on Collaborative Computing*, December 2005.

[13] Intel. *Intel Trusted Execution Technology Preliminary Architecture Specification*, November 2006.

[14] 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*, pages 1–14, June 2006.

[15] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of Crypto '96*, pages 104–113, 1996.

[16] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 2–13, June 2005.

[17] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pages 121–136, December 2004.

[18] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192, October 2003.

[19] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[20] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[21] R. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, April 1980.

[22] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), August 2006.

[23] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, pages 391–394, April 2005.

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

[25] M. D. Schroeder and J. H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[26] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, December 1999.

[27] W. Shi, J. B. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, pages 222–231, February 2006.

[28] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *Proceedings of the First ACM EuroSys Conference*, pages 161–174, 2006.

[29] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, pages 279–292, November 2006.

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.