Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation

Nathan Dautenhahn[1], Theodoros Kasampalis[1], Will Dietz[1], John Criswell[2], and Vikram Adve[1]
[1]University of Illinois at Urbana-Champaign, [2]University of Rochester
{dautenh1, kasampa2, wdietz2, vadve}@illinois.edu, criswell@cs.rochester.edu

Abstract

Monolithic operating system designs undermine the security of computing systems by allowing single exploits anywhere in the kernel to enjoy full supervisor privilege. The *nested kernel operating system architecture* addresses this problem by "nesting" a small isolated kernel within a traditional monolithic kernel. The "nested kernel" interposes on all updates to virtual memory translations to assert protections on physical memory, thus significantly reducing the trusted computing base for memory access control enforcement. We incorporated the nested kernel architecture into FreeBSD on x86-64 hardware while allowing the entire operating system, including untrusted components, to operate at the highest hardware privilege level by write-protecting MMU translations and *de-privileging* the untrusted part of the kernel. Our implementation inherently enforces *kernel code integrity* while still allowing dynamically loaded kernel modules, thus defending against code injection attacks. We also demonstrate that the nested kernel architecture allows kernel developers to isolate memory in ways not possible in monolithic kernels by introducing write-mediation and write-logging services to protect critical system data structures. Performance of the nested kernel prototype shows modest overheads: $<1\%$ average for Apache and 2.7% for kernel compile. Overall, our results and experience show that the nested kernel design can be retrofitted to existing monolithic kernels, providing important security benefits.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Organization and Design

Keywords intra-kernel isolation; operating system architecture; malicious operating systems; virtual memory

1. Introduction

Critical information protection design principles, e.g., fail-safe defaults, complete mediation, least privilege, and least common mechanism [34, 40, 41], have been well known for several decades. Unfortunately, monolithic commodity operating systems (OSes), like Windows, Mac OS X, Linux, and FreeBSD, lack sufficient protection mechanisms with which to adhere to these design principles. As a result, these OS kernels define and store access control policies in main memory which any code executing within the kernel can modify. The impact of this default, *shared-everything* environment is that the entirety of the kernel, including potentially buggy device drivers [12], forms a single large trusted computing base (TCB) for all applications on the system. An exploit of any part of the kernel allows complete access to all memory and resources on the system. Consequently, commodity OSes have been susceptible to a range of kernel malware [27] and memory corruption attacks [5]. Even systems employing features such as non-executable pages, supervisor-mode access prevention, and supervisor-mode execution protection are susceptible to both user level attacks [25] and kernel level threats that directly disable these protections.

Traditional methods of applying these principles in OS kernels either completely abandon monolithic design (e.g., microkernels [3, 9, 28]) or rely upon transparent enforcement of isolation via an external virtual machine monitor (VMM) [15, 35, 44, 55, 56]. Microkernels require extensive redesign and implementation of the operating system. VMMs suffer from both performance issues and a lack of semantic knowledge at the OS level to transparently support protection easily; also, generating semantic knowledge has been shown to be circumventable [6]. More recent techniques to split existing commodity operating systems into multiple protection domains using page protections [46] or software fault isolation (SFI) [18] incur high overhead and continue to trust "core" kernel code (which may not be all that trustworthy [15, 16]).

To address this problem, we present a new OS organization, the **nested kernel architecture**, which restricts MMU control to a small subset of kernel code, effectively "nesting"

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694386

a memory protection domain within the larger kernel. The key design feature in the nested kernel architecture is that a very small portion of the kernel code and data operate within an isolated environment called the *nested kernel*; the rest of the kernel, called the *outer kernel*, is untrusted. The nested kernel architecture can be incorporated into an existing monolithic commodity kernel through a minimal *reorganization* of the kernel design, as we demonstrate using FreeBSD 9.0. The nested kernel isolates and mediates modifications to itself and other protected memory by 1) configuring the MMU such that all mappings to protected pages (minimally the page-table pages (PTPs)) are read-only, and 2) ensuring that those policies are enforced at runtime while the untrusted code is operating. Although similar to a microkernel, the nested kernel only requires MMU isolation and maintains a single monolithic address space abstraction between trusted and untrusted components.

We present a concrete prototype of the nested kernel architecture, called PerspicuOS, that implements the nested kernel design on the x86-64 [2] architecture. PerspicuOS introduces a novel isolation technique where both the outer kernel and nested kernel operate at the same hardware privilege level—contrary to isolation in a microkernel where untrusted code operates in user-mode. PerspicuOS enforces read-only permissions on outer kernel code by employing existing, simple hardware mechanisms, namely the MMU, IOMMU, and the Write-Protect Enable (WP) bit in *CR0*, which enforces read-only policies even on supervisor-mode writes. By using the *WP-bit*, PerspicuOS efficiently toggles write-protections on transitions between the outer kernel and nested kernel without swapping address spaces or crossing traditional hardware privilege boundaries.

PerspicuOS ensures that the outer kernel never disables write-protections (e.g., via the *WP-bit*) by 1) *de-privileging* the outer kernel code and 2) maintaining that *de-privileged* code state by enforcing lifetime kernel code integrity—a key security property explored by several previous works, most notably SecVisor [43] and NICKLE [38]. PerspicuOS *de-privileges* outer kernel code by replacing instances of writes to *CR0* with invocations of nested kernel services and enforces lifetime kernel code integrity by restricting outer kernel code execution to validated, write-protected code. In this way PerspicuOS *creates two virtual privileges within the same hardware privilege level*, thus virtualizing ring 0.

By isolating the MMU, the nested kernel architecture can enforce *intra-kernel* memory isolation policies that trust only the nested kernel. Therefore, the nested kernel architecture exposes two *intra-kernel* write-protection services to kernel developers: write-mediation and write-logging. Write-mediation enables kernel developers to deploy security policies that isolate and control access to critical kernel data, including kernel code. In some cases, data may require valid updates from a large portion of the kernel, making it hard to protect kernel objects in place, or otherwise not have an applicable write-mediation policy; consequently, we present the write-logging interface that ensures all modifications to protected kernel objects are recorded (a design principle suggested by Saltzer et al. [41]).

To demonstrate the benefit of the write-mediation and write-logging facilities—for enhancing commodity OS security—we present three intra-kernel write-protection policies and applications. First, we introduce the write-once mediation policy that only allows a single update to protected data structures, and apply it to protect the system call vector table, defending against kernel call hooking [27]. In general, the write-once policy presents a novel defense against non-control data attacks [11]. Second, we introduce the append-only mediation policy that only allows append operations to list type data structures, and apply it to protect data generated by a system call logging facility. Additionally, the system call logging facility guarantees invocation of monitored events, a feature made possible by PerspicuOS's code integrity property, and therefore supports a pivotal feature required by a large class of security monitors [35, 44]. Third, we deploy a write-logging policy to track modifications to FreeBSD's process list data structures, allowing our system to detect direct kernel object manipulation (DKOM) attacks used by rootkits to hide malicious processes [27].

We have retrofitted the nested kernel architecture into an existing commodity OS: the FreeBSD 9.0 kernel. Our experimental evaluation shows that this reorganization requires approximately 2000 lines of FreeBSD code modifications while significantly reducing the TCB of memory isolation code to less than 5000 lines of nested kernel code. Our prototype also demonstrates that it is feasible to completely remove MMU modifying instructions from the untrusted portion of the kernel while allowing it to operate in ring 0. Furthermore, our experiments show that the nested kernel architecture incurs very low overheads for relatively OS-intensive system benchmarks: $< 1\%$ for Apache and 2.7% for a full kernel compile. In summary, the key contributions of this paper include

- A new OS organization strategy, the nested kernel architecture, which nests, within a monolithic kernel, a higher privilege protection domain that enables kernel developers to explicitly apply *intra-kernel* security policies through the use of write-mediation and write-logging services.
- A novel x86-64 implementation of the nested kernel architecture, PerspicuOS, that virtualizes supervisor privilege, thereby keeping the outer kernel and the nested kernel at the highest protection level; PerspicuOS uses only the MMU and control registers to protect memory, in lieu of VMM extensions, expensive page table manipulation, or costly compiler techniques.
- An evaluation of three *intra-kernel* memory access control policies: a write-once policy to protect access to

the system call table, a write-logging policy that detects rootkits attempting to hide processes, and an append-only system call logging facility with incorruptible logs and guaranteed invocation.
- An OS design that provides lifetime kernel code integrity, defending against a large class of kernel malware.

2. Nested Kernel Approach

The primary insight and contribution of the nested kernel architecture is to demonstrate how to virtualize a minimal subset of hardware functionality, specifically the MMU, to guarantee mediation and therefore isolation of *intra-kernel* protection domains. By virtualizing the MMU, the nested kernel architecture enables a new set of protection policies based upon physical page resources and their mappings within the kernel. This section presents the nested kernel architecture overview, our foundational design principles, and the challenges of virtualizing the MMU; it concludes with a description of the nested kernel write protection service made available to kernel developers.

2.1 System Overview

The nested kernel architecture partitions and reorganizes a monolithic kernel into two privilege domains: the *nested kernel* and the *outer kernel*. The nested kernel is a subset of the kernel's code and data that has full system privilege, and most importantly, the nested kernel has sole privilege to modify the underlying physical MMU (pMMU) state. The nested kernel mediates outer kernel modifications to the MMU via a virtual interface, which we refer to as the virtual MMU (vMMU). The outer kernel is then modified to use the vMMU.

Similar to previous work [15, 43], the nested kernel architecture isolates pMMU updates at the final stage of creating a virtual to physical translation: the point at which a virtual-to-physical translation is made *active* on the processor (i.e., when the processor can use the translation). For example, on the x86-64, address mappings are added to the system by storing a value to a virtual memory location, called a page-table entry (PTE), that resides on a page-table page (PTP) [2]. By selecting this abstraction, the outer kernel still manages all aspects of the virtual memory subsystem; however, the nested kernel interposes on all pMMU updates, thereby allowing the nested kernel to isolate the pMMU and enforce any other access control policy in the system, such as the one used to protect nested kernel code and data.

2.2 Design Principles

The nested kernel architecture comprises the mechanism and interface to establish virtual address mappings. As such, we seek to accomplish the following:

Separate resource control (e.g., policy) from protection mechanism (e.g., MMU). We seek the lowest level of abstraction possible to virtualize the MMU, providing only a mechanism that performs updates to virtual-to-physical address mappings. This principle has several benefits: it minimizes the TCB of the privileged domain, maximizes the portability of the nested kernel, and gives maximum flexibility to the types of policies implemented in the outer kernel while maintaining isolation of the nested kernel.

Operating system co-design and explicit interface. OS designers are experts in how their systems work: they represent the best opportunity to enhance the security of the system. Therefore, the nested kernel architecture presents a unified design to realize protections explicitly within the OS rather than transparently enforcing protections via external tools, such as in the case with prior work [35, 43, 44].

Privilege separation based upon MMU state, not instructions. Traditionally, systems use the notion of rings of protection, where each ring prescribes what instructions may be executed by code in that ring. In contrast, we enforce privilege separation in terms of access to the pMMU, including both memory (e.g,. PTPs) and CPU state (e.g., *WP-bit* in *CR0*).

Minimal architecture dependence. We want to make the nested kernel architecture design as hardware agnostic as possible, assuming only a hardware paging mechanism with page-granularity protections and the ability to enforce write-protections on outer kernel code.

Fine grained resource control. The protections enabled by virtualizing the MMU can be expressed in many ways; we seek to enable fine grained resource control, i.e., protections at byte-level granularity, so that *intra-kernel* isolation policies can be applied to arbitrary OS data structures.

Negligible performance impact. The nested kernel architecture provides isolation and privilege separation without requiring separate address spaces so that it can be applied to operating system architectures with minimal overhead. In our x86-64 prototype, we also run both the outer kernel and nested kernel in the same protection ring (ring 0) rather than via hardware virtualization extensions to avoid costly hypercalls, as evidenced by measurements in Section 5.3.

2.3 Virtualizing the MMU via API Emulation

We summarize the runtime isolation of the pMMU as the following property, which Invariants I1 and I2 enforce:

Nested Kernel Property. *The nested kernel interposes on all modifications of the pMMU via the vMMU.*

Invariant 1. Active *virtual-to-physical mappings for protected data are configured read-only while the outer kernel executes.*

Invariant 2. *Write-protection permissions in* active *virtual-to-physical mappings are enforced while the outer kernel executes.*

Active virtual-to-physical mappings are those mappings that may be used by the processor to determine page protections; inactive mappings do not affect memory access privileges. Invariant I2 applies to those processors (such as the x86 [2]) which can disable page protections while still performing virtual-to-physical address translation. While these definitions are independent of whether the MMU uses hardware- or software-managed TLBs, we will assume a hardware-managed TLB to simplify discussion.

On a hardware-TLB system, the nested kernel architecture enforces Invariant I1 by 1) requiring explicit initialization of PTPs, 2) creating an explicit interface to update the page-table entries (PTEs), and 3) configuring all PTEs that map PTPs as read-only. Therefore, any PTP that has not been explicitly initialized at boot time by the nested kernel or declared by the outer kernel via the vMMU is rejected from use, enforcing Invariant I1.

Invariant I2 can be enforced by a variety of mechanisms, including internal page protection mechanisms such as used in our prototype or external mechanisms such as a virtual machine monitor running at a higher hardware privilege level. Section 3.2 details how we ensure Invariant I2 is enforced in PerspicuOS on the x86-64 architecture.

2.4 Intra-Kernel Memory Write Protection Services

By isolating the pMMU from the outer kernel, the nested kernel can fully enforce memory access control policies on any physical page in the system. For example, the nested kernel can write-protect all statically defined constant data or a subset of system call function pointers that never change at runtime. Therefore, the nested kernel architecture provides a simple, robust API for specifying and enforcing such policies on kernel memory. The write-protection services API, listed in Table 1, comprises memory allocation and a data write function with an accompanying byte-granularity mediation policy.

Clients use the *intra-kernel* protection services to allocate regions of memory that are protected by and only written from nested kernel code. When an allocation is requested, either statically via `nk_declare` or dynamically via `nk_alloc`, the nested kernel initializes a write descriptor and allocates an associated memory region. The nested kernel also establishes the memory bounds for the region and sets the mediation callback function (as defined below) that implements the write-protection policy. The nested kernel returns to the client both the write descriptor and virtual address of the newly allocated write protected region, and finally, write-protects all existing mappings to the physical pages containing the memory region.

Clients specify write-protection policies in the form of *mediation functions*. Mediation functions enforce the update policies for write-protected kernel objects, and are invoked by the nested kernel prior to any writes. One example of a simple mediation function is a no-write policy for constant data, with function body, `return false;`, which rejects all writes to the memory region. A more complex example is a write-once policy, such as described in Section 4.1.1, where the nested kernel initializes a bitmap for each byte in the allocated memory region, then upon an `nk_write`, validates that the write is only made to memory not previously written. A significant value of the write-protection interface is that even in the absence of a mediation function (e.g., all writes to the object are permitted), the updates must use `nk_write`, thus thwarting overwrites from memory corruption bugs.

Once a write descriptor, `nk_wd`, is created, the outer kernel executes mediated writes via the `nk_write` function. `nk_write` operates similarly to a simple byte-level memory copy operation. `nk_write` performs two checks prior to executing the write: 1) it verifies that the write is within the boundary of the region specified by `nk_wd`, and 2) it invokes the mediation function, if any. By allowing clients to write only a subset of the memory region, the nested kernel allows protection of aggregate data types without requiring any knowledge about its fields. The interface also makes bounds checking fast by including the write descriptor for constant-time lookup of the descriptor information for the given region.

To fully support dynamically allocated memory, the nested kernel provides `nk_free`, which deallocates memory previously allocated by `nk_alloc`. Because an OS exploit could prematurely force `nk_free` to be called on a memory region and then attempt to store to it, any freed memory must be retained in protected memory, and so we design a simple interface that assumes the allocator is part of the nested kernel.

2.5 Preventing DMA Memory Writes

The nested kernel must also prevent DMA writes to protected memory. We require that the system have an IOMMU [1] that the nested kernel can use to ensure that DMA operations do not modify any pages protected by the nested kernel.

3. PerspicuOS: A Nested Kernel Prototype

We present a concrete implementation of the nested kernel architecture, named PerspicuOS, for x86-64 processors. PerspicuOS introduces a novel method for ensuring privilege separation between the outer kernel and the nested kernel while running both at the highest hardware privilege level, effectively creating two virtual privilege levels in *ring 0*. PerspicuOS achieves this goal by taking advantage of x86-64 hardware support for efficiently enabling and disabling MMU write protection enforcement and by controlling which privileged instructions can be used by outer kernel code. More specifically, PerspicuOS applies the design presented in Section 2.3, by configuring all mappings to PTPs as read-only and *de-privileging* the outer kernel so that it cannot disable write-protection enforcement at ring 0. PerspicuOS *de-privileges* the outer kernel by scanning

Function	Selected Arguments	Purpose
nk_declare	mem_start, size, mediation_func	Marks all pages RO; initializes an NK write descriptor nk_wd; returns the nk_wd and the pointer to the region.
nk_alloc	size, mediation_func, nk_wd_p	Allocates memory region; invokes nk_declare on it; stores write descriptor in nk_wd; returns nk_wd and pointer to the region.
nk_free	nk_wd	Deallocates memory identified by nk_wd. Memory must have been allocated by nk_alloc. Freed pages can be reused only by a future nk_alloc.
nk_write	dest, src, size, nk_wd	Verifies write bounds; invokes mediation_func, if any; then copies size bytes from src to dest.

Table 1. Nested Kernel Write Protection API. nk_declare is for static allocation and nk_alloc is for dynamic allocation.

all outer kernel code to ensure that it does not contain instructions that disable the *WP-bit* or the MMU. Additional hardware features (described in Section 3.5) prevent user-space code or kernel data from being used to disable protections.

In this section, we describe our threat model, specify a set of invariants to maintain the *Nested Kernel Property*, and then discuss how PerspicuOS maintains the invariants through a combination of virtual privilege switch management, MMU configuration validation, and lifetime kernel code integrity.

3.1 Threat Model and Assumptions

In this work, we assume that the outer kernel may be under complete control of the attacker who can attempt to arbitrarily modify CPU state. Furthermore, we assume that an attacker can modify outer kernel source code, i.e., that outer kernel code may be malicious. Moreover, we do not assume or require outer kernel control flow integrity, which means that an attacker can arbitrarily target any memory location on the system for execution. For example, since nested kernel and outer kernel code may reside in a unified address space, an attacker could attempt to redirect execution to arbitrary locations within nested kernel code, including instructions that toggle write-protections (i.e., the nested kernel must take explicit steps to prevent such control transfers or render them harmless).

We assume that the nested kernel source code and binaries are trusted and that the nested kernel is loaded with a secure boot mechanism such as in AEGIS [45] or UEFI [48]. We also trust mediation functions, a necessary requirement to ensure security checks execute in PerspicuOS. We assume that the nested kernel and mediation functions are free of vulnerabilities, and given the small source code size (less than 5,000 lines-of-code), the nested kernel could be formally or manually verified. Furthermore, we assume that the hardware is free of vulnerabilities and do not protect against hardware attacks.

3.2 Protection Properties and Invariants

The nested kernel design specifies two invariants that must hold to enforce the *Nested Kernel Property*. Invariant I1 requires that all active mappings to PTPs be configured as read-only; Invariant I2 requires that these configurations be enforced while the outer kernel is in operation. We systematically assessed the x86-64 architecture specification [2] to identify both the necessary hardware configurations to realize invariants I1 and I2 and the hardware configurations that may violate those invariants. For example, write-protections are enforced on supervisor-mode accesses when both the *WP-bit* is set and the mapping is configured as read-only; however, alternative execution modes, such as System Management Mode (SMM), can bypass write-protections when invoked. From this assessment, we derive the following invariants that ensure that invariants I1 and I2 hold.

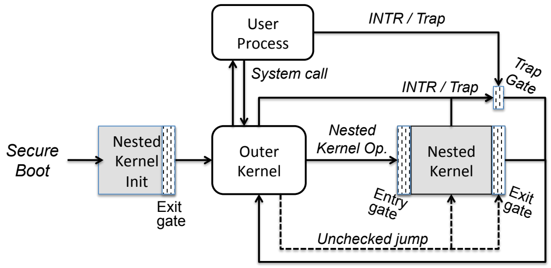

Figure 1. PerspicuOS State Transition Diagram. Only shaded blocks can execute PerspicuOS privileged operations (Table 2). All transitions out of the nested kernel must go through the Exit Gate.

3.2.1 Supporting Invariant I1

The set of active mappings in x86-64 is controlled by the *CR3* register and a set of in-memory PTPs [2]. *CR3* specifies the base address of a "top-level" page serving as the root for a hierarchical translation data structure that is traversed by the MMU [2]. To ensure that all translations to protected physical pages are marked as read-only (thereby asserting I1), PerspicuOS enforces the following invariants:

Invariant 3. *Ensure that there are no unvalidated mappings prior to outer kernel execution.*

Invariant 4. *Only declared PTPs are used in mappings.*

Invariant 5. *All mappings to PTPs are marked read-only.*

Invariant 6. *CR3 is only loaded with a pre-declared top-level PTP.*

3.2.2 Supporting Invariant I2

PerspicuOS must ensure that, while the outer kernel is operating, MMU write-protections are continually enforced. Read-only permissions are enforced by x86-64 when the processor is operating in long mode with write-protections enabled, i.e., Protected Mode Enable (*PE-bit*), Paging Enabled (*PG-bit*), and Write-Protect Enable (*WP-bit*) bits are set in *CR0*; Physical Address Extensions (*PAE-bit*) bit is set in *CR4*; and Long Mode Enable (*LME-bit*) bit is set in the *EFER* model specific register (MSR) [2]. Therefore, PerspicuOS considers scenarios where the outer kernel attempts to 1) disable the *WP-bit* while in operation, 2) disable paging by modifying the *PG-bit*, or 3) subvert control flow of the nested kernel so that the outer kernel gains control of execution while the *WP-bit* has been legitimately disabled for nested kernel operations. PerspicuOS ensures that the *WP-bit* is always set while the outer kernel is in operation and that any instantaneous mode changes that could disable paging, such as an SMM interrupt, are directed to nested kernel control.

Invariants I7 and I8 capture the requirements of the *WP-bit*.

Invariant 7. *The WP and PG flags in CR0 are set prior to any outer kernel execution.*

Invariant 8. *The WP-bit in CR0 is never disabled by outer kernel code.*

When the *PG-bit* is disabled, the processor immediately interprets virtual addresses as physical addresses [2]. As Section 3.7 describes, preventing the outer kernel from clearing the *PG-bit* is impossible. Instead, PerspicuOS enforces the following invariant:

Invariant 9. *Disabling the PG-bit directs control flow to the nested kernel.*

Additionally, SMM may be invoked by the outer kernel and therefore, PerspicuOS must also assert control on the SMI interrupt.

Invariant 10. *The nested kernel controls the SMM interrupt handler and operation.*

Given that the previous set of invariants hold, the outer kernel might attempt to manipulate CPU state or outer kernel memory in such a way as to cause control-flow to move from nested kernel code to outer kernel code without re-enabling the *WP-bit*. Therefore, to ensure write-protections are always enforced, PerspicuOS must protect against control-flow attacks on nested kernel execution in two specific cases: interrupt control flow paths and nested kernel stack state manipulation.

PerspicuOS ensures that all exit paths from the nested kernel to the outer kernel enable the *WP-bit* (shown in Figure 3), which is captured in the following invariant:

Invariant 11. *Enable the WP-bit on interrupts and traps prior to calling outer kernel interrupt/trap handlers.*

Because the trap handlers are a part of the nested kernel, the Interrupt Descriptor Table (IDT) [2] must be placed in protected memory and modifications of the Interrupt Descriptor Table Register (IDTR) must be solely a nested kernel operation.

Invariant 12. *The IDT must be write-protected, and the IDTR is only updated by the nested kernel.*

On a multiprocessor system, code running in outer kernel context on one core could modify the return address stored on the stack by code running in nested kernel on another core if the stack is in outer kernel memory. This would cause nested kernel code to return to outer kernel context without enabling the *WP-bit*. Therefore, PerspicuOS must ensure that code running in the nested kernel uses its own stack located in nested kernel memory.

Invariant 13. *The nested kernel stack is write-protected from outer kernel modifications.*

3.3 System Initialization

PerspicuOS must ensure that all mappings to protected pages (e.g. PTPs, code, nested kernel data, etc.) are configured as read-only and that paging is enabled prior to outer kernel execution, as suggested by invariants I3 and I7. Therefore, PerspicuOS, as depicted in Figure 1, initializes the paging system so that invariants I3—where validation implies invariants I4, I5, and I6 by registering all protected pages in nested kernel data structures—and I7 are enforced prior to outer kernel execution by using secure boot and "nested kernel init" functionality, thereby initializing all PTEs in the system.

3.4 Virtual MMU Interface

PerspicuOS provides a set of functions, called the nested kernel operations, that allow the outer kernel to configure the pMMU. The nested kernel operations interpose on underlying x86-64 instructions, called *protected instructions*, to isolate the pMMU. There are two classes of nested kernel operations: those that control the configuration of the hardware PTPs via memory writes and those that control updates to processor control registers.

The nested kernel enforces pMMU update policies by assigning types to physical pages based upon the kind of data stored in each physical page. The page types include PTPs, nested kernel code and data, outer kernel code and data, user code and data, and data protected by the intra-kernel write-protection service. This type information, along with the number of active mappings and a list of all virtual address mappings to the page, is kept in a *physical page descriptor*.

The outer kernel uses the `nk_declare_PTP` operation to specify the physical pages to be used as PTPs. The

Operation	x86 Instruction	Description	Constraints
nk_declare_PTP	None	Initialize physical page descriptor as usable in page tables	Asserting invariant I4
nk_write_PTE	mov VAL, PTEADDR	Update pMMU mapping	Asserting invariants I4 and I5
nk_remove_PTP	mov VAL, PTEADDR	Remove physical page from being used as PTP	Supporting invariants I4, I5, and I6.
nk_load_*CR0*	mov %REG, *%CR0*	Controls enforcement of read-only mappings	*WP-bit* must be set: invariant I8
nk_load_*CR3*	mov %REG, *%CR3*	Controls MMU mapping base PML4 page	Value must be a declared PML4-PTP
nk_load_*CR4*	mov %REG, *%CR4*	Controls user mode execution with *SMEP* flag	*CR4 SMEP* flag must be 1
nk_load_MSR	wrmsr Value, MSR	Control enforcement of no-execute permissions	*EFER* NX-Bit must be set to 1

Table 2. Nested Kernel Operations, *Protected Instructions*, Description, and Constraints

nk_declare_PTP operation takes, as arguments, the level within the page table hierarchy at which the physical page will be used and the address of the physical page being declared, then zeros each page to eliminate any stale data, write-protects all existing virtual mappings to the physical page, and registers the physical page as a PTP by updating the page's physical page descriptor.

Once declared, a physical page cannot be modified directly by outer kernel code. Instead, the outer kernel uses the nk_write_PTE operation, which inspects and validates all mappings prior to insertion. The nested kernel uses the previously described physical page type information along with a list of existing mappings to each page to ensure that 1) if the PTE does not point to a data page then it targets a declared PTP and 2) all mappings to PTPs are write-protected, thereby ensuring invariants I4 and I5 respectively. The nested kernel also protects nested kernel code, data, and stack pages to avoid code modifications that would eliminate mediation or functionality of the pMMU update process. We also ensure that the update does not write to any kernel data protected by the nested kernel; this is done via a simple check that ensures that the physical page being updated was previously declared as a page table page.

The second group of operations configure the paging hardware itself. We expose an interface for updating *CR3* to ensure that it only points to a declared top-level PTP, called PML4-PTP, thereby ensuring invariant I6. The interface for modifying other registers ensures that paging and lifetime kernel code integrity protections are not disabled by outer kernel code. The description of these mechanisms are in Sections 3.5 and 3.7.

```
entry:
  pushfq                             Save current flags
  cli                                Disable interrupts
  mov %rax, -8(%rsp)                 Spill regs for temps
  mov %rcx, -16(%rsp)
  mov %rsp, %rcx                     Save stack ptr in rcx
  mov %cr0, %rax                     Get current CR0 value
  and ~CR0_WP,%rax                   Clear WP bit in copy
  mov %rax, %cr0                     Write back to CR0
  cli                                Disable interrupts
  mov PerCPUSecureStack,%rsp         Switch to secure stack
  push %rcx                          Save orig stack ptr
  mov -0x8(%rcx), %rax               Restore spilled regs
  mov -0x10(%rcx), %rcx
```

Figure 2. Nested Kernel Entry.

```
exit:
  mov 0(%rsp), %rsp                  Restore orig stack ptr
  push %rax                          Spill scratch reg
  mov %cr0, %rax                     Get current CR0 value
1:
  or CR0_WP, %rax                    Set WP in CR0 copy
  mov %rax, %cr0                     Write back to CR0
  test CR0_WP, %eax                  Ensure WP set
  je 1b                              If not, loop back
  pop %rax                           Restore clobbered reg
  popfq                              Restore flags
                                     (incl interrupt status)
```

Figure 3. Nested Kernel Exit

3.5 Lifetime Kernel Code Integrity

To prevent *protected instructions* from being executed while in outer kernel context, PerspicuOS first validates all code before making it executable in supervisor-mode, and second, protects the runtime integrity of validated code by enforcing lifetime kernel code integrity, thereby maintaining invariants I6 and I8. PerspicuOS enforces *load time outer kernel code validity* by scanning binary code to ensure that it does not contain any *protected instructions*, including at unaligned instruction boundaries. Then PerspicuOS enforces dynamic lifetime outer kernel code integrity by configuring the processor and pMMU so that 1) by default all kernel pages are mapped as non-executable (enforced by the no-execute bit (*NX-bit*) in the *EFER* MSR), 2) validated kernel code pages are mapped with read-only permissions, and 3) user-space code and data are mapped as non-executable in supervisor-mode by employing supervisor-mode execution prevention (*SMEP* in *CR4*) [2], thereby preventing the outer kernel from executing any *protected instructions* contained within user-mode pages. Note that because protecting the nested kernel depends upon kernel code integrity both *EFER* and *CR4* must also be removed from outer kernel's ability to execute and are thus *protected instructions* as depicted in Table 2.

3.6 Virtual Privilege Switches

In PerspicuOS, the nested kernel and outer kernel share a single address space. Therefore, nested kernel operations are essentially function calls to nested kernel functions that are wrapped by entry and exit gates that (among other things) disable and enable the *WP-bit*. Virtual privilege switches occur when write-protection is disabled (which only occurs on nested kernel operations). In this section, we detail PerspicuOS entry and exit gates and describe the ways in which PerspicuOS ensures that the outer kernel does not gain control while write protections are disabled (enforcing I11, I13) and how the gates ensure that mediation functions execute (ensuring I4 and I5).

3.6.1 Nested Kernel Entry and Exit Gates

The nested kernel entry and exit gates ensure that there is a clear and protected privilege boundary between the nested kernel and the outer kernel. The routines depicted in Figures 2 and 3 perform the virtual privilege switch. The entry gate (Figure 2) disables interrupts, turns off system-wide write protections, disables interrupts, and then switches to a secure nested kernel stack; the exit gate (Figure 3) executes the reverse sequence. PerspicuOS by default disables interrupts while in operation; however, we include the second interrupt disable instruction to avoid instances where the outer kernel invokes interrupts that may corrupt internal nested kernel state.

3.6.2 Interrupts

PerspicuOS disables interrupts when executing in the nested kernel. Because the nested kernel is limited to a very small set of functionality, disabling interrupts is not expected to impact performance. Disabling interrupts simplifies the design of nested kernel operations because they can execute atomically: they do not need to contend with the possibility of being interrupted. However, long-running mediation functions may need to run with interrupts enabled—we leave supporting this feature as future work.

PerspicuOS must also ensure that the *WP-bit* is set whenever either a trap occurs or if the outer kernel directly invokes the *WP-bit* disable instruction and subsequently manages to execute an interrupt prior to the second interrupt disable instruction. This is necessary because an attacker could feed inputs to a mediated function that causes it to generate a trap; if the handler runs in the outer kernel, it would be running with write-protection disabled. PerspicuOS protects against these attacks by isolating the x86-64 interrupt handler table [2], enforcing invariant I12, and configuring it to send all interrupts and traps through the nested kernel trap gate first—depicted in Figure 1; the nested kernel trap gate sets the *WP-bit* before transferring control to an outer kernel trap handler, following a similar loop as the exit gate starting at assembly label "1" in Figure 3, thus enforcing invariant I11.

3.6.3 Nested Kernel Stack

To enforce invariant I13, PerspicuOS includes separate stacks for the nested kernel. Upon entry to the nested kernel, PerspicuOS saves the existing outer kernel stack pointer and switches to a preallocated nested kernel stack, as shown in Figure 2. When exiting the nested kernel, PerspicuOS restores the original outer kernel stack pointer (Figure 3).

3.6.4 Ensuring Write Mediation

By mapping the nested kernel code into the same address space as the outer kernel, PerspicuOS gains in efficiency on privilege switches; however, the outer kernel can directly jump to instructions that modify the protected state. For example, the outer kernel can target the instruction that writes to PTP entries, thus, bypassing the vMMU mediation. However, such a write will fail with a protection trap because the jump would have bypassed the entry gate, which is the only way to turn off the system-wide write protections enforced by the *WP-bit* (Figure 2). In this way, PerspicuOS ensures that either mediation will occur or the system will detect a write violation.

3.7 Privileged Register Integrity

While the protections in Section 3.5 prevent the outer kernel from directly modifying privileged registers (e.g., *CR0*, *CR3*, *IDTR*), it is possible for the outer kernel to jump to instructions within the nested kernel that configure these registers. To protect against this, the nested kernel unmaps pages containing these instructions from the virtual address space when the outer kernel is executing and maps them only when needed. Invariants I6, protecting *CR3*, and I12, protecting *IDTR*, are enforced using this method because direct modification of these registers can allow the outer kernel to instantly gain control.

While this works for most privileged registers, it does not work for *CR0* (to enforce I8) because the entry and exit gates must toggle write protections, and therefore the instruction to disable *CR0* must be mapped into the same address space as the outer kernel. Therefore, the outer kernel could load a value into the RAX register and jump to the instruction in the entry and exit gates that move RAX into *CR0*. Ideally, the entry and exit gates would use bit-wise OR and AND instructions with immediate operands to set and clear the *WP-bit* in *CR0*. Unfortunately, the x86-64 lacks such instructions; it can only copy a value in a general purpose register into a control register [2]. Note that the *protected instruction* "mov %REG, *%CR0*" is only mapped at three code locations, the entry, exit, and trap gates.

Entry gates do not require verification of the value loaded to *CR0* because the purpose of the entry gate is to disable the *WP-bit*; in contrast, exit and trap gates return control-flow to the outer kernel after modifying *CR0*. The exit and trap gates must therefore ensure that the *WP-bit* is enabled. To do so, PerspicuOS inserts a simple check and loop in the exit gate to ensure that the value of *RAX* has the *WP-bit* enabled, thus ensuring invariant I8. Since these are the only instances of writes to *CR0* in the code, PerspicuOS ensures that outer kernel attacks cannot bypass write-protections by using these instructions.

In the x86-64 architecture, paging is enabled when the processor is in either protected mode with paging enabled (both *PG-bit* and *PE-bit* set) or long mode (PG, PE, PAE, and the LME bits set). To handle the situation where the outer kernel disables the *PG-bit* (regardless of whether the CPU is in long or protected mode), PerspicuOS configures the MMU so that the virtual address of the entry gate matches a physical address containing code that traps into the nested kernel. Therefore, enforcing invariant I9 whenever the *PG-bit* is disabled.

If either the *PAE-bit* or *LME-bit* are disabled while the CPU is in long mode a general protection fault occurs. Because the bits are not updated but instead a trap occurs, the write-protections continue to be enabled and do not require any other solution. According to the Intel Architecture Reference Manual, the *PE-bit* cannot be disabled unless the *PG-bit* is also disabled [2], which is handled by the previously described solution.

3.8 Allocating Protected Data Structures

PerspicuOS presents the intra-kernel write-protection interface as described in Section 2.4 for allocating and updating write protected data structures. PerspicuOS establishes a predefined ELF memory region to protect global statically-defined data structures. Kernel developers declare write-protected data structures with a C macro that uses a special compiler directive to notify the linker to allocate the object into the specified region. The macro then registers the object into the write descriptor table along with the precomputed bounds and generates both the `nk_wd` and pointer to the region. PerspicuOS provides for dynamic allocation via the interface as described in Section 2.4. The shadow process list example uses this interface, which is described in Section 4.1.3.

One of the primary challenges of implementing the nested kernel write protection services is to devise a method for conquering the protection granularity gap [51], specifically the issue of protecting data co-located on pages with non-protected data. The nested kernel interface can fully support in-place protections but would result in poor performance: each unprotected object would require a trap and emulate cycle. Therefore, we modify the linker script to put this protected ELF region onto its own set of separate pages so that only write-protected data is placed in the region. At boot time, pages belonging to this protected ELF section are write-protected via MMU configuration to ensure the *Nested Kernel Property* for each of these data structures.

3.9 Mediation Functions

In an ideal nested kernel implementation, mediation functions would not be in the TCB. This would keep the TCB small regardless of the number of policies and would allow policies to be mutally distrusting. However, to simplify implementation, and to ensure that the mediation functions are executed prior to writes, mediation functions are incorporated into PerspicuOS's TCB. In our evaluation of the write protection interface, we present a set of predefined trusted mediation functions (which, like the mediation functions in an ideal design, do not write to nested kernel memory).

3.10 Implementation

We implemented PerspicuOS in FreeBSD 9.0. We replaced all instances of writes to PTPs to use the appropriate nested kernel API function and inserted validation checks as Section 3.4 describes. We modified the trap handlers to check for and enable the *WP-bit*; however, we did not implement the IDT and IDTR protections. We believe that these will not impact performance as modern OS kernels rarely modify the IDT and IDTR. We ported and implemented nested kernel calls for each function in Table 2. These calls perform the requested operation and verify that the value in the register is correct. We ported all instances of writes to MSRs to ensure the NX bit is always set in *EFER*; however, we did not fully implement no-execute page permissions in the PTPs. We do not believe these will negatively impact performance as the nested kernel already interposes on all MMU updates and sets other protection bits accordingly. We also implemented an offline scanner for the kernel binary; we have applied this to the entire core kernel but not to dynamically loaded kernel modules (this is a minor matter of engineering).

Our current implementation uses coarse-gained synchronization even though our evaluation is on a uni-processor. It uses a single nested kernel stack with a lock to protect it from concurrent access. We did not implement protections for DMA writes or enforce nested kernel control on SMI events; however, we do not believe they will negatively impact performance because these are rare events under normal operation. Last, we did not fully implement all features to enforce Invariant I6; however, we did implement code that updates a PTE and flushes the TLB to simulate mapping and unmapping the code that modifies *CR3*. We believe this faithfully represents the performance costs of the full solution.

4. Enforcing Intra-Kernel Security Policies

The nested kernel architecture permits kernel developers to employ fundamental design principles such as least privilege and complete mediation [41]. In this section, we explore several intra-kernel security policies enabled by the nested kernel. Our examples demonstrate the nested kernel's ability to combat key mechanisms used by well-known classes of kernel malware such as rootkits [27].

We emphasize that our use cases do not completely solve specific high-level security goals (such as preventing rootkits from evading detection). However, they demonstrate specific key elements for complete solutions. Developing complete solutions is part of our ongoing work.

4.1 Nested Kernel Write Mediation Policies

The nested kernel provides kernel developers with the ability to prevent or monitor memory writes at run-time. We illustrate three write-protection policies that this interface can enforce; each can be used for multiple security goals.

4.1.1 Write-Once Data

Several kernel data structures are written to only once, when they are initialized. Other structures are initialized to default values and are only changed once during operation (e.g., the system call table). Our interface can protect these data with very low overhead.

As such, PerspicuOS implements a simple, byte-granularity, *write-once* policy within the nested kernel. It is enforced by maintaining a bit-vector with one bit per byte, initialized to zeroes. When `nk_write` is called, it uses a mediation function that checks whether each bit is set for the memory to be modified; if all the bits are clear, it writes the data and marks those corresponding bits as being written.

We apply the write-once policy to protect the FreeBSD system call table by allocating the table within nested kernel-protected pages and selecting the *write-once* policy, guaranteeing that it can never be overwritten by malware after initialization. This application defends against kernel malware that "hook" system call dispatch by overwriting entries in the system call table to invoke exploit code [27], and could be extended to protect other key kernel code pointers.

4.1.2 Append-Only Data

Operating systems also have append-only data structures such as circular buffers and event logs. These data structures reside in ordinary kernel memory and are vulnerable to kernel exploits, making them unreliable for forensics use.

To protect such data structures, PerspicuOS implements an *append-only* write policy within the nested kernel. It is enforced by maintaining a "tail" pointer to a list structure within the nested kernel. Each call to `nk_write` increments the tail pointer to ensure that writes never overwrite existing data within the region. A stricter policy could ensure that no gaps exist between successive writes. A full solution must also be able to securely write the log to disk when full, which our prototype does not yet do.

We used this policy to implement a system call event logger that records system call entry and exit events in a statically allocated, append-only buffer. System call recording has been a popular target in both research systems [22, 23, 32, 36] and security monitoring applications [19, 21, 26, 33, 35, 44, 52]. However, these systems are susceptible to attack [49]. By protecting the log buffer, we ensure that rootkits cannot hide traces of malicious system call events and strengthen security staffs' ability to conduct forensics investigations after breakins. Further effort is required to write the logs out to another media for long term storage, and to defend against an attacker that spoofs security events.

4.1.3 Write Logging

A rootkit's primary goal is to hide itself and malicious processes and files. Therefore, they often modify kernel data such as network counters, process lists, and system event logs [27]. Some of these data are challenging to protect due to being co-located within large kernel data structures; others cannot be protected by simple write-once and append-only policies. However, the ability to reliably *monitor* writes to such data enables detection of all malicious modifications.

Therefore, we implement a general *write-logging* mechanism that records (and can later reconstruct) all writes to selected data structures. All calls to `nk_write` for a memory region declared with this policy record the range of addresses modified and the values written into the memory. Again, this buffer must be periodically written to disk.

As an example use case of the interface, we use *write-logging* to detect rootkits that attempt to hide processes by corrupting FreeBSD's process list data structure: `allproc`. Instead of logging writes to `allproc` directly, we created a shadow *allproc* data structure that exactly mirrors the original list. Each shadow list entry contains a pointer back to the corresponding `allproc` entry, and any updates to the `allproc` list structure (e.g., unlinking a node) are also performed on the shadow list. More importantly, to fully hide the presence of a particular process from the kernel, the rootkit must use `nk_write` to remove the shadow entry from the shadow list (which is logged).

The logging of shadow list writes enables effective forensics. Security monitors can easily reconstruct the list updates and identify the prior existence of hidden processes. Moreover, we modified the `ps` program to query the shadow list instead of the `allproc` list so that the `ps` program can detect the presence of hidden processes.

4.2 System Protection Policies

The nested kernel architecture can also realize several system security properties because it controls all virtual memory mappings in the system. One example is *lifetime kernel code integrity* (as Section 3.5 explains). This single use case effectively thwarts an entire class of kernel malware (namely code injection attacks). In addition to code integrity, PerspicuOS also marks memory pages as non-executable by default and enables superuser mode execution prevention of user-mode code and data. Even if commodity kernels use these hardware features, they cannot prevent malware from disabling them. PerspicuOS, in contrast, enables these protections *and* prevents malicious code from disabling them. PerspicuOS can also be used for any type of security monitor that inserts explicit calls into source code to ensure that the monitor both executes and is isolated from the untrusted code.

5. Evaluation

We evaluate PerspicuOS by investigating the impact on the TCB, FreeBSD porting effort, *de-privileging* scanner, and performance overheads.

We evaluated the overheads of PerspicuOS on a Dell Precision T1650 workstation with an Intel® Core$^{\text{TM}}$ i7-3770 processor at 3.4 GHz with 8 MB of cache, 16 GB of RAM, and an integrated PCIE Gigabit Ethernet card. Experiments requiring a network used a dedicated Gigabit ethernet network; the client machine on the network was an Acer Aspire Revo R3700 with an Intel® Atom$^{\text{TM}}$ D525 processor at 1.8 GHz with 2 GB of RAM. We evaluate five systems for each of our tests: the *original* (unmodified) FreeBSD system, the base PerspicuOS, and each of our three use cases: append-only, which is used for system call entry and exit recording; write-once, used for the system call table protection; and write-log, used for the shadow process list. The baseline for the syscall use case was the original FreeBSD modified so that it is logging system call entry and exit events.

5.1 Trusted Computing Base and Kernel Porting

The nested kernel requires porting existing functionality in a commodity kernel to use the nested kernel operations. Our port of FreeBSD to the nested kernel architecture modified 52 files totalling $\sim$1900^{+} LOC changed, including comments. The vast majority of deleted lines were to configuration or build system files—ignoring these, only $\sim$100^{-} LOC were eliminated in the port. Code modifications were measured using Git change logs. We measure the number of lines in the nested kernel with the SLOCCount tool [53]: the implementation consists of $\sim$4000 C SLOC and $\sim$800 assembly SLOC; the scanner was implemented in 248 python SLOC.

5.2 Code Scanning Results

To evaluate the feasibility of eliminating all instances of *protected instructions* from the outer kernel, we scanned our compiled kernels and subsequently used manual methods to eliminate all unaligned *protected instructions*. We found a total of 40 implicit instructions for writing to *CR0* (2) and *wrmsr* (38). Most of these instances are due to constants embedded in the code used for relative addressing; therefore, we eliminated them by adjusting alignments, rearranging functions, and inserting *nop*s. A few were due to particular sequences of arithmetic expressions; these were addressed by replacing them with equivalent computation. Finally, a small number of constants in the outer kernel code contained implicit instructions. These were addressed by replacing the each constant with two others that were dynamically combined to create the equivalent value.

5.3 Privilege Boundary Microbenchmark

To investigate the impact of different privilege crossings against the nested kernel architecture approach, we developed a simple microbenchmark that evaluates the round trip cost into a null function for each privilege boundary: syscall, nested kernel call, and VMM call (hypercall).

Privilege Boundary	Time (μsecs)	Time / NK Call
NK Call	0.1390	1.00x
Syscall	0.08757	0.62x
VMCALL	0.5130	3.69x

Table 3. Privilege Boundary Crossing Costs.

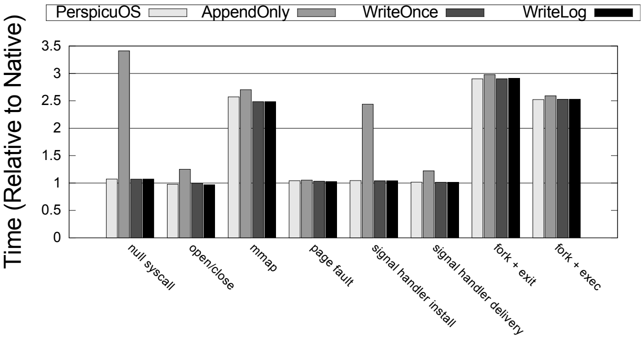

Figure 4. LMBench results.

For the syscall boundary experiment we used the *syscall* instruction to invoke a special system call added to the kernel that immediately returns. The VMM boundary cost experiment is performed using a guest kernel consisting solely of *VMCALL* instructions in a loop executing within a VMM modified to resume the guest immediately after this instruction traps to the VMM. The nested kernel cost experiment uses an empty function wrapped with the entry and exit gates as described in Section 3.6.1.

The microbenchmark performs each call one million times and reports total elapsed time. Each microbenchmark configuration was executed 5 times with negligible variance, and the computed average time per call is reported.

Our results, shown in Table 3, indicate that a nested kernel call is approximately 3.69 times *less* expensive than a hypercall, thus motivating the performance benefits of implementing the nested kernel architecture at a single supervisor privilege level. User-mode to supervisor-mode calls are faster than nested kernel calls, which take approximately 1.59 times as long.

5.4 Micro-benchmarks

To evaluate the effect that PerspicuOS has on system call performance, we ran experiments from the LMBench benchmark suite [30]. Figure 4 shows the results for our four systems relative to the original FreeBSD. In most cases, our systems are, at most, 1.25 times slower relative to the baseline (unmodified) FreeBSD kernel. `mmap`, `fork+exit`, and `fork+exec`, however, exhibit higher execution time overheads of approximately 2.5 to 3 times. This is because these benchmarks stress the vMMU with several consecutive calls to set up new address spaces. Upon investigation, we

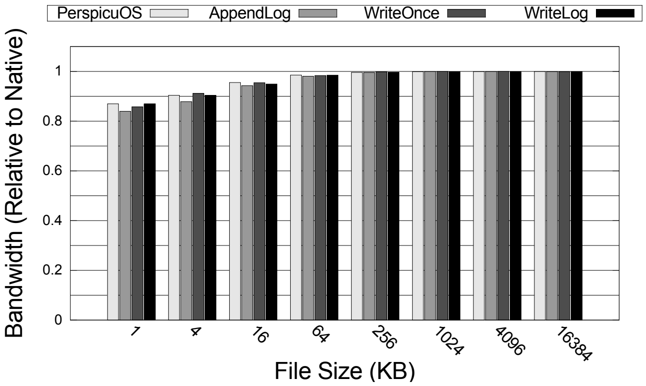

Figure 5. SSHD Average Bandwidth.

identified a small set of functions that were responsible for most of this behavior, and preliminary experiments showed a reduction by more than 60% when converting these to batch operations. In the future, we plan to extend the nested kernel interface to allow for batch updates to the vMMU in order to reduce overheads for these operations.

We also observe that the write-once and write-log policies incur the same overheads as the base PerspicuOS system, whereas the append-only policy used for system call entry and exit recording incurs higher overheads. In fact, the worst *relative* overhead for this system is the `null syscall` benchmark; it occurs because each null system call makes two nested kernel operations calls.

5.5 Application Benchmarks

To evaluate the overheads on real applications, we measured the performance of the FreeBSD OpenSSH server, the Apache httpd server running on the PerspicuOS kernel, and a kernel compile. We opted to use network servers as they exercise kernel functionality more heavily than many compute bound applications and are therefore more likely to be impacted by kernel overhead.

OpenSSH Server: For the OpenSSH experiments, we transferred files ranging from 1 KB to 16 MB in size from the server to the client. We transferred each file 20 times, measuring the bandwidth achieved each time. Figure 5 shows the average bandwidth overhead, relative to native, for each file size transferred. The maximum bandwidth reduction is 20% for 1 KB files. Transferring files above 64 KB in size has less than 2% reduction in bandwidth.

Apache: For the Apache experiments, we used Apache's benchmark tool `ab` to perform 10000 requests using 32 concurrent connections over a 1Gbps network for file sizes ranging from 1 KB to 1 GB. We performed this experiment 20 times for each file size, and present the results in Figure 6. The experiment results reveal negligible, if any, overheads that are within the standard deviation error.

Kernel Build: The kernel build experiment cleaned and built a FreeBSD kernel from scratch for a total of 3 runs (the variance was virtually negligible). The worst run times are shown in Table 4. The results show an overhead of about 2.6% for the base PerspicuOS, system call table, and process list configurations; and an overhead of 3% for the system call entry/exit case.

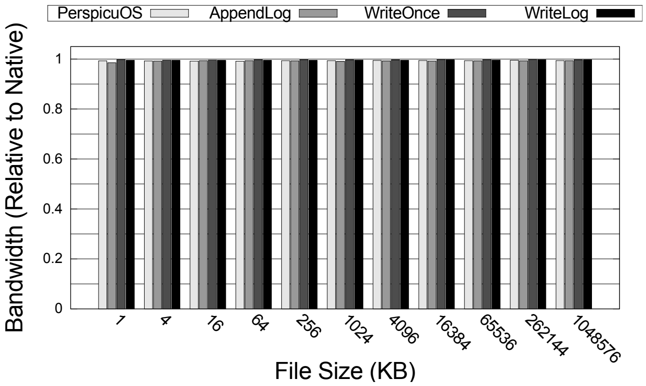

Figure 6. Apache average bandwidth.

PerspicuOS	AppendOnly	WriteOnce	WriteLog
2.6%	3.0%	2.6%	2.7%

Table 4. Kernel Build Overhead over Native.

6. Future Work

There are several directions for future work. First, we plan to investigate methods for removing mediation functions from the TCB. Mediation functions do not need to write to protected memory and could be excluded from the TCB by running them with write-protections enabled. Second, although the nested kernel provides a sufficient interface to protect data structures, techniques are needed to ensure that policy-enforcing code stores all critical data within the nested kernel. Third, we plan to formally verify PerspicuOS's design to improve assurance of its correctness.

Additionally, we will explore applications of PerspicuOS. For example, moving the kernel memory allocator into the nested kernel could protect the kernel from memory safety attacks that overwrite allocator meta-data [5]. Additionally, we could move the access control functionality into the nested kernel, thereby ensuring that attacks on the operating system kernel cannot subvert its access controls.

7. Related Work

The core contributions of this work include a new OS organization, the nested kernel architecture, for providing privilege separation and isolation within a single monolithic kernel, and a unique method for implementing it on commodity hardware with PerspicuOS.

Operating System Organizations. Several alternative operating system designs provide privilege separation and memory isolation, including microkernels [3, 9, 28], ExoKernels [8, 17], and separation kernels [39]. OSes written in type safe languages also provide inherent security improvements [4, 9, 42]. While these approaches isolate kernel components and mediate access to critical data structures, they completely abandon commodity OS design.

MMU Protections. The nested kernel architecture isolates the MMU by modifying the outer kernel so that MMU updates can be mediated, and exports an interface similar to those of related efforts including Xen [7], SVA-OS [13–15, 20] and `paravirtops` [54]. Although the interface is similar, the nested kernel architecture employs different *p*MMU policies to protect and virtualize the MMU, as well as introduces *de-privileging* to isolate the nested kernel. SecVisor employs similar MMU policies to enforce kernel code integrity [43] as PerspicuOS; however, SecVisor uses special nested paging hardware support that uses implicit traps on certain hardware events, which is both external to the kernel and has higher costs per invocation than PerspicuOS.

Intra-Kernel Memory Isolation. SILVER [55] and UCON [56] specify policy frameworks (similar to mandatory access control) to enforce access control policies on internal kernel objects using VMM hardware. SILVER exports an access control service that is used by the operating system to specify principals and object ownership access policies through the memory allocator, which are then enforced by the VMM. In contrast, PerspicuOS uses the x86-64 *WP-bit* to provide a memory isolation mechanism on which SILVER access control polices could be overlaid.

Nooks [46] provides lightweight protection domains for kernel drivers and modules. Nooks uses the hardware MMU to create protection domains and changes hardware page tables when transferring control between the core kernel and the kernel driver. Although Nooks provides reliability guarantees, it does not consider isolation from malicious entities, and therefore is susceptible to attack.

Compiler Based Intra-Kernel Memory Isolation. Several previous efforts [10, 18] employ software fault isolation (SFI) and control-flow integrity (CFI) to isolate kernel components. These systems utilize heavy weight compiler instrumentation in addition to address translation policies to isolate kernel components. LXFI [29] uses programmer annotations to specify interface policy rules between kernel extensions and the core kernel and inserts run-time checks to enforce these rules. In contrast, PerspicuOS does not require compiler-based enforcement mechanisms, alleviates the need for kernel control-flow integrity, and removes the core kernel from the TCB.

Hypervisor Based Isolation. Both SVA [15] and HyperSafe [50] employ the MMU and the *WP-bit* to prevent privileged system software from making errant changes to page tables. However, these approaches require control flow integrity, and furthermore the HyperSafe work claimed that using the WP approach on a monolithic kernel would be too challenging due to shared code and data pages.

Several approaches deploy security monitors to protect and record certain kernel events: each has drawbacks. Several such approaches place the monitor in the same TCB as the untrusted code, leaving them vulnerable to attack [31, 37, 47]. Other systems, namely Lares [35] and the In-VM monitor SIM [44], place the monitor in a VMM (using nested paging support) to provide integrity guarantees about the isolation and invocation of the security monitor. These systems suffer from high performance costs [35] or assume integrity of the code region [44]. VMM-based monitors must also address VMM introspection problems: the monitor does not understand the semantics of kernel data structures [19, 24]. In PerspicuOS, security monitors are isolated from the monitored system, can be invoked much more efficiently via direct nested kernel operations instead of expensive VMM hypercalls, and completely avoid the VMM introspection problem.

KCoFI [13], SecVisor [43], and NICKLE [38] provide kernel code integrity. SecVisor and NICKLE also ensure that only authorized code runs in the processor's privileged mode. PerspicuOS enforces the same policies, but also includes a novel memory isolation mechanism.

8. Conclusion

This paper presents the nested kernel architecture, a new OS organization that provides important security benefits to commodity operating systems that was retrofitted to an existing monolithic kernel. We show that the two nested kernel architecture components, the nested kernel and the outer kernel, can co-exist in the highest hardware protection level in a common address space without compromising the isolation guarantees of the system. The nested kernel architecture can efficiently support useful write-mediation policies, such as write-once and append-only, which OS developers can use to incorporate new security policies with very low performance overheads. More broadly, we expect that the nested kernel architecture can improve OS security by enabling OS developers to incorporate richer security principles like complete mediation, least privilege, and least common mechanism, for selected OS functionality.

Acknowledgments

The authors would like to thank Audrey Dautenhahn for her editorial services, and Maria Kotsifakou, Prakalp Srivastava, and Matthew Hicks for refining our ideas via technical and writing discussions. Our shepherd Peter Druschel and the anonymous reviewers provided valuable feedback that greatly enhanced the quality of this paper. This work was sponsored by ONR via grant number N00014-12-1-0552 and supported in part by ONR via grant number N00014-4-1-0525, MURI via contract number AF Subcontract UCB 00006769, and NSF via grant number CNS 07-09122.

References

[1] AMD64 architecture programmers manual volume 2: System programming. Manual, Advancd Micro Devices, 2006.

[2] Intel 64 and IA-32 architectures software developers manual. Manual 325384-051US, Intel, June 2014.

[3] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'10, pages 93–112, Altanta, GA, USA, 1986. USENIX Association.

[4] M. Aiken, M. Fhndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[5] argp and Karl. Exploiting UMA, FreeBSD's kernel memory allocator. *.:: Phrack Magazine ::.*, 0x0d(0x42), Nov. 2009.

[6] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 82–91, Washington, DC, USA, 2010. IEEE Computer Society.

[7] 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*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[8] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazires, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

[10] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, SOSP '09, pages 45–58, New York, NY, USA, 2009. ACM.

[11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[13] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, Washington, DC, USA, 2014. IEEE Computer Society.

[14] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014. ACM.

[15] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 83–100, Berkeley, CA, USA, 2009. USENIX Association.

[16] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM.

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[18] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[19] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[20] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM.

[21] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, Aug. 1998.

[22] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive application-level record-replay for replay parallelism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 193–206, New York, NY, USA, 2013. ACM.

[23] N. Honarmand and J. Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 445–456, Piscataway, NJ, USA, 2014. IEEE Press.

[24] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. SoK: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 605–620, Washington, DC, USA, 2014. IEEE Computer Society.

[25] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 957–972, Berkeley, CA, USA, 2014. USENIX Association.

[26] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM.

[27] J. Kong. *Designing BSD Rootkits*. No Starch Press, San Francisco, CA, USA, 2007.

[28] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[29] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 115–128, New York, NY, USA, 2011. ACM.

[30] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[31] Microsoft. Kernel patch protection: frequently asked questions (windows drivers), 2007.

[32] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM.

[33] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, Feb. 2006.

[34] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.

[35] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society.

[36] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas. QuickRec: Prototyping an intel architecture extension for record and replay of multithreaded programs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 643–654, New York, NY, USA, 2013. ACM.

[37] C. Ries. Defeating windows personal firewalls: Filtering methodologies, attacks, and defenses. Technical report, 2005.

[38] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[40] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, July 1974.

[41] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[42] T. Saulpaugh and C. A. Mirho. *Inside the JavaOS operating system*. Addison-Wesley Reading, 1999.

[43] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[44] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM.

[45] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[46] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, Feb. 2005.

[47] A. Tereshkin. Rootkits: Attacking personal firewalls. In *Proceedings of the Black Hat USA 2006 Conference*, 2006.

[48] I. Unified EFI. Unified extensible firmware interface specification: Version 2.2d, November 2010.

[49] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[50] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[51] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[52] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy*, SP '99, pages 133–145, Oakland, California, USA, May 1999. IEEE Computer Society.

[53] D. Wheeler. SLOCCount, 2015. http://www.dwheeler.com/sloccount/.

[54] C. Wright. Para-virtualization interfaces, 2006. http://lwn.net/Articles/194340.

[55] X. Xiong and P. Liu. SILVER: Fine-grained and transparent protection domain primitives in commodity OS kernel. In S. J. Stolfo, A. Stavrou, and C. V. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, number 8145 in Lecture Notes in Computer Science, pages 103–122. Springer Berlin Heidelberg, Jan. 2013.

[56] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, pages 71–80, New York, NY, USA, 2007. ACM.