Rethinking Operating System Design: Asymmetric Multiprocessing for Security and Performance

Scott Brookes
Thayer School of Engineering at Dartmouth College
14 Engineering Dr
Hanover, NH 03755
scott.l.brookes.th@dartmouth.edu

Stephen Taylor
Thayer School of Engineering at Dartmouth College
14 Engineering Dr
Hanover, NH 03755
stephen.taylor@dartmouth.edu

ABSTRACT

Developers and academics are constantly seeking to increase the speed and security of operating systems. Unfortunately, an increase in either one often comes at the cost of the other. In this paper, we present an operating system design that challenges a long-held tenet of multicore operating systems in order to produce an alternative architecture that has the potential to deliver both increased security and faster performance. In particular, we propose decoupling the operating system kernel from user processes by running each on completely separate processor cores instead of at different privilege levels within shared cores. Without using the hardware's privilege modes, virtualization and virtual memory contexts enforce the security policies necessary to maintain process isolation and protection. Our new kernel design paradigm offers the opportunity to simultaneously increase both performance and security; utilizing the hardware facilities for inter-core communication in place of those for privilege mode switching offers the opportunity for increased system call performance, while the hard separation between user processes and the kernel provides several strong security properties.

CCS Concepts

•Computer systems organization → Multicore architectures; •Security and privacy → Virtualization and security;

1. INTRODUCTION

Multicore systems have become ubiquitous in every corner of the modern world, providing the foundation for virtually all modern technology. For many applications, high security *and* high performance are both critical. As a result, the performance of computer systems is increasing at breakneck speed and security research is more important than ever. However, the two efforts are largely conducted independently of (and sometimes opposed to) one another. Often, an increase in security implies a decrease in performance, and vice versa. Given the highly developed state of modern systems, it is rare for any incremental change to the status quo to increase security without adding work, thereby decreasing performance. In this paper, we challenge one of the earliest and most fundamental design choices that led to the current status quo of multicore operating system design in order to arrive at a system that we believe can increase both security and performance, rather than trading one for the other.

We begin by describing the path that led to the current accepted standard in operating system design, illustrating the issues with that design, and introducing an alternative design paradigm to address these issues in Section 2. In Section 3, we discuss the various components of the alternative design in greater detail. Section 4 provides a brief survey of related and background literature. In particular, it examines previous attempts to mitigate privilege escalation, previous work studying asymmetric multiprocessing, and a selection of prior attempts to redefine traditional operating system design. Finally, Section 5 discusses possible next steps, future work, and methods for evaluating the merits of the proposed design.

The contribution of this work is the detailed description of an alternative design paradigm for multicore operating systems that offers opportunities for improvement in both performance and security. In particular, the design offers:

- Complete virtual memory isolation and "sandboxing" of every user application.
- Device drivers with the security of user-space encapsulation *and* the performance of kernel modules.
- Hardware-enforced secure contexts available for application-specific use.
- Real-time "watchdog" security monitors in the kernel for intrusion detection in applications.
- Fine-grained security policies enabled by per-core virtualization and separation of the kernel and the application.
- Decreased system call performance overhead.

2. MOTIVATION AND OVERVIEW

The fundamental role of an operating system is to provide an appropriate context for process execution, multiplex processes' access to the hardware, and protect processes from

NSPW '16, September 26-29, 2016, Granby, CO, USA
© 2016 ACM. ISBN 978-1-4503-4813-3/16/09...$15.00
DOI: http://dx.doi.org/10.1145/3011883.3011886

each another. The current state of the art operating system architecture is the product of many seminal [59, 20, 16, 7, 65, 60] and hundreds of subsequent research efforts and accomplishes all of these tasks. Current systems provide each application with a *virtual address space* in which the kernel manages the required context. The kernel provides the application with functionality for manipulating hardware by including its own code in the virtual address space of each process[1]. In order to protect processes from one another, the kernel does not share virtual address spaces between applications and it restricts access to the shared kernel functionality using hardware-provided CPU *privilege levels*.

Unfortunately, some characteristics of the current kernel design lead to security and performance issues. The popular return-to-user (ret2usr) style attack [35] is enabled by sharing the virtual address space between the kernel and the application, despite the CPU privilege levels. Additionally, the frequent interrupts used to change privilege levels between the kernel and application generate substantial performance overhead.

The current relationship between the operating system and a user application is largely a historical artifact. In the early days of computing, with just one processing unit, only one program could run at any given time. There needed to be a way to pass control from one program to another (in this case, from the application to the kernel), and some notion of differing privileges between the two programs. To accomplish the first task, system designers simply included both "programs" in a single computing context. For the second, the hardware provided the privilege mode switch triggered by an interrupt (an `INT` instruction on x86). When the processor reached this instruction, it would save the state of the current operating context to a known location, change the CPU's privilege mode, and direct execution to a predefined kernel entrance routine. Later, when the kernel wanted to resume execution of the application, it would issue a special return (`IRET` on x86), at which point the hardware would demote privilege and transfer execution to a kernel-defined location in the application's code. More recently, the x86 architecture defined `SYSENTER` and `SYSEXIT` as alternatives to `INT` and `IRET` for the system call (syscall) interface.

With the advent of multicore processors, the need to share hardware no longer forced this design onto kernel developers. However, since each core still provided the functionality of the unicore processor, it became standard practice to simply take the established formula and replicate it, instantiating one instance per core. The only special care needed was that no two cores would execute the same code at the same time. Originally, this was accomplished using a simple "kernel lock," a software mechanism that restricted access to the kernel to only one processor at a time [41]. More recently, many kernels have narrowed the granularity of their locked paths so that two or more fully independent paths through the kernel may be used simultaneously by two or more different cores [64]. Overall, this scheme of employing a single mechanism on each available core is known as *Symmetric MultiProcessing (SMP)*.

We suggest that the decision to *replicate* the unicore operating system design onto each core in multicore hardware imposed unnecessary limits on the security and performance attainable on modern systems. In this paper, we revisit that decision and explain how to *replace* the unicore operating system design with one that leverages modern multicore hardware to address some of the weaknesses found in modern designs. The unique utilization of hardware resources that we present in this paper fundamentally changes how systems provide security to their different components. While microkernels redistribute system components to increase the security offered by monolithic kernels, they do so using the exact same hardware abstractions to differentiate user- and kernel-space. In contrast, our paradigm changes the underlying hardware abstractions used for coordination and translation between different components of the system. In fact, any current kernel, whether monolithic or micro, could be implemented using our design. Even cutting edge virtualization-based security efforts use the same hardware design patterns that operating systems have been using for years [57]. As argued in [10] this "turtles all the way down" approach to security is not sufficient. In our design, virtualization is used as a tool for the kernel to protect itself, rather than as an abstraction that replicates the classic kernel design paradigm in a new software layer.

Our design explores the idea of using *Asymmetric MultiProcessing (AsMP)* on modern x86 hardware. Its main goal is to divorce the virtual address space of the application from that of the kernel. The distinct, decoupled operating environments provided on each processor allow for the kernel and the application to use completely different hardware resources, rather than sharing a common processor core. *Inter-Processor Interrupts (IPIs)* will take the role of the standard `INT/SYSENTER` and `IRET/SYSEXIT` functionality for system calls[2]. The application and the kernel will no longer share hardware, so the hardware-provided CPU privilege mode switch is not needed, and neither program has to relinquish its own resources to message or signal the other. Additionally, virtualization will be utilized to implement fine-grained security rules on a per-core basis to further police the software.

This paradigm shift interrupts commonly deployed privilege escalation mechanisms by defining privilege by a physical location on-chip rather than traditional processor modes. As a result, more advanced virtualization-based defenses can be employed because each core is responsible for *either* the kernel *or* an application - never both. Additionally, the fact that the kernel and the application do not share hardware resources means that they can operate simultaneously. This creates opportunities for techniques such as using kernel to perform watchdog state checks on applications while they run or applications utilizing truly asynchronous system calls by signaling the kernel but continuing to run while the kernel services its request. The combination of all of these techniques and their derivative features will deliver strong separation between the kernel and an application, fast IPI-based control transfers and messaging, advanced virtualization-based and application-specific security features, and new opportunities such as asynchronous system calls, kernel-based watchdog processes, and application-specific hardware-secured subcontexts.

[1]The few examples of systems that use "strong" rather than "weak" separation between kernel memory and user memory include the 4G/4G split Linux patch [49], 32-bit XNU [36], and certain systems using the hardware facilities provided by SPARC V9 hardware [46]

[2]Future work that seeks to implement this design will explore polling as a possible alternative to IPIs as suggested by an anonymous NSPW reviewer.

2.1 Threat Model

This work assumes that an adversary has physical or remote user-level access to the system after it has been initialized. He does not have access to the system during initialization and may not influence the boot process; issues such as trusted boot are beyond the scope of our work. The attacker hopes to escalate his privilege in order to gain access to protected data or code using some privilege escalation *mechanism*, such as return-to-user (ret2usr) attacks [35], rather than a particular privilege escalation *vulnerability*, such as [2, 1, 47]. The attacker may have offline access to the source and/or binary code of the kernel and any application running on the system, and arbitrary control of any code/data to which his privilege allows access.

3. ASMP KERNEL DESIGN PARADIGM

Figure 1 illustrates an overview of our design. Core number, rather than the traditional CPU privilege level (ring), differentiates kernel- from user-space. Core-specific security rules in the virtualization layer and virtual memory isolation for ring 0 processes recover the protection offered by the traditional ring 3 user-space. Interprocessor interrupts provide a mechanism for implementing system calls across cores. With ring 3 no longer used for isolating user-space, the application can use the hardware protection associated with ring 3 to provide its own secure sub-contexts.

This section explores the features of our design in detail. The main factors contributing to increased security are divorcing kernel- and user-space, utilizing user-space drivers and deploying fine-grained per-core virtualization-based security policies. The increase in performance will come from a faster mechanism for signaling from an application to the kernel, less overhead in driver implementations, and more efficient parallelization of computation.

3.1 Separating Kernel and User Cores

The primary characteristic of the proposed design is that the kernel and the user application do not share a virtual address space and are each executed at ring 0, but on independent cores. This arrangement will have several implications on the security of the system.

3.1.1 Ring 3 (and more) Protection in Ring 0

The weak separation between kernel- and user-space in traditional operating system design allows for even the most strictly "sandboxed" processes to be used as a stepping stone for compromising the whole system because privilege escalation compromises the kernel. Since the kernel needs to manage all of system memory, it maintains virtual mappings to all memory; a process that shares its virtual address space with the kernel can perform arbitrary reads and writes with sufficient privilege escalation.

When not shared with the kernel, the application's virtual address space does not need to contain mappings to all of system memory. Consequently, the application cannot inspect or modify memory unless the kernel has given it access, even with arbitrarily high privilege. Naively, this means that the application can safely be run in ring 0. However, additional steps are required to recreate the full protections offered by ring 3 while running in ring 0.

In ring 0, the process is a supervisor on-core and can read from or write to any memory mapped into its address space. In order to stop a process from modifying its page tables to create mappings for arbitrary system memory, we will omit virtual mappings to the application's page tables from the application's virtual address space. In order to modify its own page tables, the page tables need to contain a virtual mapping to themselves; without it, the application cannot modify the memory used by the memory management unit to define its own virtual address space. This ensures that the process is permanently and completely isolated in virtual memory and enables the process' page tables to deny access to memory-mapped devices.

The virtualization layer also helps to recreate ring 3 protection by denying inappropriate hardware accesses like reading from and writing to processor control registers or executing typically privileged instructions. The end result is an application execution context with the same protections offered by ring 3, but without the crown jewels of the system hiding behind a CPU privilege level bit.

3.1.2 Hardware Control from Applications

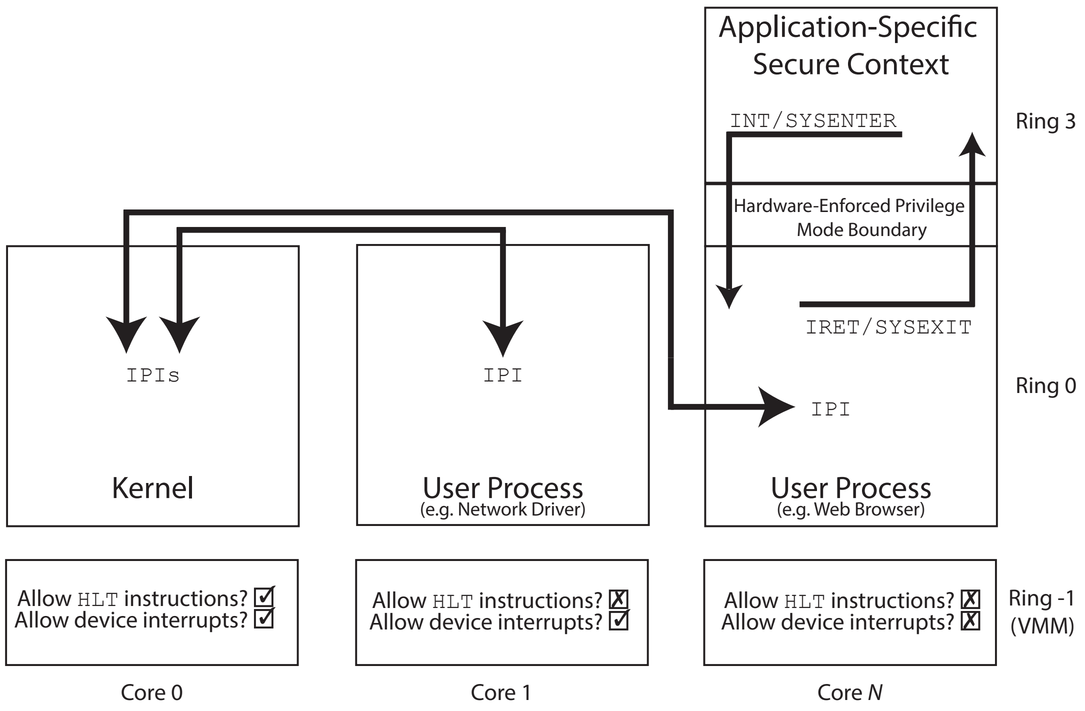
User-space drivers [26] minimize the amount of potentially third party and frequently buggy [25, 15, 53] code that runs with privilege. Unfortunately, this comes at a significant performance cost. Device drivers need to interact with hardware, for which they need privilege that they do not have in ring 3. Therefore, they must frequently interact with the kernel to accomplish their tasks, significantly degrading performance, increasing the kernel's code base, and widening the interface between user- and kernel-space.

Although we've shown that applications can be run in ring 0 with privileges less than or equal to those of ring 3, we can actually increase security by providing drivers with privileges greater than those available in ring 3. Setting up the proper operating context for a driver (as suggested in the virtualization layer on the core being used for a network driver in Figure 1) will allow the driver to be implemented in a single context. This is an improvement over traditional "user-space" drivers that need the kernel to do work on their behalf. It keeps the kernel's code base small and interface narrow.

3.1.3 Application-Specific Secure Contexts

By moving user-space from ring 3 on a shared core to ring 0 on a user-specific core, we've introduced yet another opportunity to increase security. In particular, ring 3 is no longer being used by the system. Instead, applications can (with some kernel support) use ring 3 to provide their own secure contexts.

For an example of how this feature might be used, consider a web browser. Although a browser strives to protect web pages from one another and protect itself from web pages, cross-site scripting [63] is just one example of a common attack vector in which a single web page can compromise an entire web browser. However, if the browser is running in ring 0, it could manage individual web pages in ring 3 the same way that traditional operating systems manage user processes. When the user opens a new web page, the browser could ask the kernel for a fresh address space that contains itself in ring 0 and a clean context for the new web page in ring 3. The kernel needs to handle this task, but the browser could use a local scheduler implementation (separate from the main scheduler in the kernel) to cycle through these different contexts without kernel intervention during its own scheduled time quantum.



Application-Specific Secure Context
INT/SYSENTER
Ring 3
Hardware-Enforced Privilege Mode Boundary
IRET/SYSEXIT
IPIs
IPI
IPI
Ring 0
Kernel
User Process
(e.g. Network Driver)
User Process
(e.g. Web Browser)
Allow HLT instructions? ☑
Allow device interrupts? ☑
Allow HLT instructions? ☒
Allow device interrupts? ☑
Allow HLT instructions? ☒
Allow device interrupts? ☒
Ring -1
(VMM)
Core 0
Core 1
Core N

Figure 1: System design using the proposed paradigm. CPU privilege level (ring) 0, traditionally used for the kernel context, contains the main code for each core: either the kernel or a user processes. The virtualization layer, commonly referred to as ring -1, protects the hardware from user processes. With ring 3 no longer being used by the system, individual applications can use the hardware to provide their own secure sub-contexts.

3.1.4 Watchdog Kernel Processes

In current operating systems, the kernel is invoked strictly when an application needs work done on its behalf. In other words, the kernel is never running without a specific task to handle. In contrast, with the asymmetric multiprocessing solution, the kernel will be running on a dedicated core whether or not it is servicing a specific process' request. This creates an opportunity: time during which the kernel is executing but has no specific task to complete. One possible way to utilize this opportunity is to implement a watchdog routine [17, 44] in the kernel. This routine could monitor system and application invariants in real time. For example, it could hash a process' code to check for code patching, monitor the stack looking for ROP payloads, or profile system components to detect unfamiliar configurations. In the event that an anomaly is detected, the application can be suspended pending further action.

The watchdog routine in this implementation is doing its work with CPU cycles that would otherwise be wasted on the kernel core. These cycles could also be used for other tasks in the interest of either performance or security. One example of the former is pre-computing values that are likely to be requested by applications in future system calls. In the case of the latter, the kernel could use these cycles as a third line of defense (after virtual memory isolation and virtualization) to restrict user processes being run in ring 0. For example, if the kernel wants the application to alert it in the event of a processor fault, the watchdog process can verify that the process does not modify the fault handlers mapped into its context.

3.2 Fine-Grained Virtualization

One advantage of the proposed design is the ability to exercise each core's virtualization hardware independently to impose fine-grained security rules on each core based on the software that that particular core will be executing. Previously, we discussed why this is necessary to recreate ring 3 protections for user processes in ring 0. Moving beyond necessity to benefit, this section examines a virtualization-based security project and shows how it could be further strengthened with the new design paradigm.

ExOShim [11] is a thin layer of virtualization underneath the kernel used to mark all memory associated with kernel code as *execute-only* using the Extended Page Table (EPT) functionality provided by modern Intel VT-x hardware [30]. This mitigates the risk of kernel-level memory disclosures that could facilitate reverse-engineering or the construction of kernel ROP or JIT-ROP payloads. Additionally, ExOShim does not accept any inputs or tolerate any permissions violations so that it can provide this protection for the lifetime of the system, even in the face of kernel compromise.

Figure 2 illustrates how the physical frames of memory corresponding to kernel code are marked execute-only in the EPT, denying kernel-level memory disclosure vulnerabilities. However, all other memory is marked with the most liberal possible permissions in the EPT: read, write, and execute. This is because the applications running on the processor may need to use this memory. Therefore, the virtualization layer relies on the kernel to maintain the appropriate memory management and access control for this memory. Note that this is consistent with most hypervisors; only the least

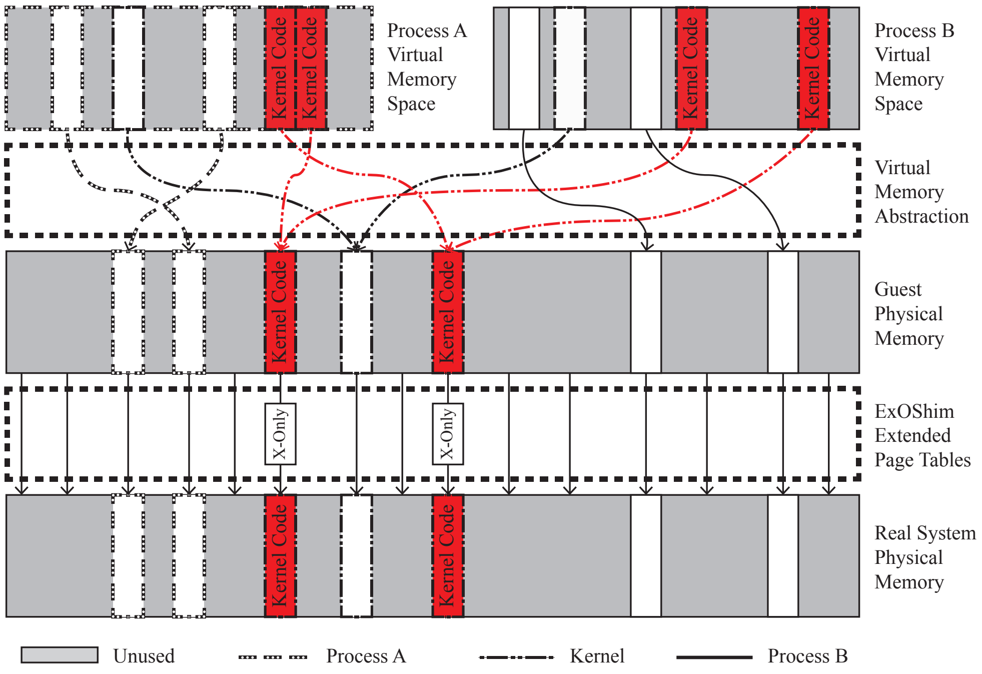


Figure 2: Overview of the Protections Provided by ExOShim [11]. All kernel code pages are marked execute-only in the hypervisor's EPTs. All other memory is marked with the most liberal permissions to allow the kernel to manage memory for individual applications.

restrictive security rules can be deployed for the lifetime of a virtual machine. This is an unfortunate byproduct of SMP; the hypervisor cannot enforce stricter security rules because it cannot predict which core will need which set of rules.

Unfortunately, the permissive rules applied to this memory allows for the possibility of a ret2usr [35] or ret2dir [34] style attack. Intuitively, this is because the security rules that ExOShim applies to the kernel are not complete. It enforces a rule that *all kernel code is execute-only*. However, it does not enforce the desirable rule that *the kernel may execute only kernel code*. This rule would eliminate the possibility of the processor executing any non-kernel code with kernel privilege, but the requirement that application code must be able to utilize this memory from this core denies the possibility of applying such a strong rule in the virtualization layer. This rule would be better implemented with ExOShim than with hardware extensions such as Intel's SMEP [24]. SMEP can be disabled in the event of kernel compromise, SMEP bypass techniques have already been demonstrated [58, 33], and SMEP cannot mitigate the threat of a ret2dir style attack [34] while ExOShim can.

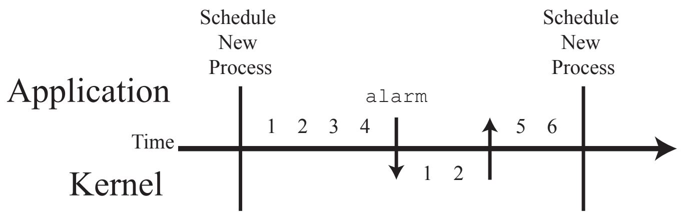
In the proposed design, however, the kernel runs on its own core; no application code will run on this core. This means that the ret2usr attack is defeated directly: there is no user-space to return to in the kernel context. Additionally, the kernel core's virtualization layer can enforce even stronger security rules than the ExOShim prototype presented in [11]. In addition to marking all memory corresponding to kernel code as execute-only, it will mark all other memory as non-executable. This will preserve the protection against memory disclosure vulnerabilities while also denying ret2dir style attacks by prohibiting the execution of any non-kernel code from the kernel core.

3.3 Performance Implications

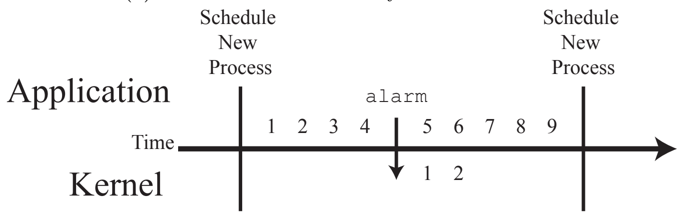
One possible issue facing the performance of the proposed architecture is the symmetrical nature of hardware resources in SMP chips. In particular, caches and shared memory or peripheral busses are optimized for symmetric use by all cores. Using the cores asymmetrically may result in suboptimal performance of these hardware facilities which could only be resolved at the hardware development stage.

The more obvious performance concern is that the proposed design suggests reserving one processor core for the kernel, decreasing the total number of cores available for simultaneous operation of applications by one. Applications that are optimized to make heavy concurrent use of all available cores will see a decrease in performance with the fewer resources available to them. However, Amdahl's Law [4] suggests that there is a limit to how many processor cores can decrease the overall time required for an application. Therefore, in the future (as hardware continues to scale and offer more and more processors) removing any constant number of cores from those available will not significantly decrease the performance of any particular application. In fact, in a future implementation of this architecture deployed on a system with a very high number of available processor cores, the design could reserve one core per independently locked path in the kernel and still have many cores available for active processes.

Additionally, the proposed design offers several opportunities to increase the performance of an application that will be immediately evident in most cases. These performance gains may even help to offset the cost of removing a core from specialized highly-concurrent processes.



(a) With the Traditional Syscall Mechanism



(b) With the Proposed IPI-Based Mechanism

Figure 3: Application's work completed in a given time quantum. With the proposed IPI-based syscall mechanism, the clock cycle following the call to the kernel belongs to the application, not to the kernel. This is not the case with traditional system call interfaces; even syscalls that claim to be "asynchronous" suspend the application while it is delivering its request to the kernel.

3.3.1 *Simultaneous Application and Kernel Execution*

Recall that the proposed design will have the kernel and application operating simultaneously on different processor cores. This opens the door for new classes of system call interfaces. In particular, the application does not necessarily need to stop execution while it makes a syscall. Figure 3 contrasts the work that an application can accomplish with a traditional syscall and with the proposed IPI-based design. It illustrates that when the application requests work from the kernel (in this case, setting an "alarm") with the traditional interface, it must wait for the kernel to perform the requested action prior to continuing its work[3]. In the proposed design, however, it can continue its work immediately because the application and the kernel are not sharing hardware. As a result, the application is able to complete more work in a given time quantum.

Note that this truly asynchronous system call interface is only possible if the kernel is not sharing hardware with the application. As such, many additional research questions exist before a full implementation of this feature can be realized. For example, if the kernel fails to perform the requested task, it must alert the process; this is nontrivial if the process has changed state since the time of the syscall. Additionally, the process needs a way to know that the kernel is prepared to handle a second syscall after the first has been dispatched. Despite these unanswered questions, the possibility for this type of system call mechanism has promising implications for system performance.

[3]Note that this is true even in the case of the "asynchronous system calls" [13] used in some current operating systems. These interfaces may provide the capability for the process to work while the kernel is servicing its request, but the process must still relinquish its hardware to the kernel *while it is making the request.*

3.3.2 *IPIs vs.* `INT/IRET`

In addition to the possibility for doing additional work per unit time by running the kernel and application simultaneously, using IPIs to handle control transfer from the application to the kernel is faster than using the `SYSENTER/SYSEXIT` or `INT/IRET` mechanisms. Using the traditional mechanisms, the hardware goes through context switching routines in which it saves and restores the appropriate states. In the proposed design, this type of context switching is not necessary. In the case of a regular system call, the operating context of the user process is maintained on the user's core; not sacrificing the hardware to the kernel means no state saving is required. Similarly, the kernel does not need its state restored because it is operating on its own hardware which maintains its own state.

In the case where context switching is required, namely during scheduling, some additional work will be done to preform the required state saving. The method to best implement this additional task will be explored in the future implementation of a prototype implementing this design, but is likely to require some modification to user applications within the system call library.

3.3.3 *Removing Drivers from the Kernel*

In Section 3.1.2 we explained how our design allows us to give device drivers both privileged access to hardware and full user-space encapsulation. We suggested that our design could reveal faster performance than the equivalent driver in user-space as traditionally defined because the driver can access hardware from its own context rather than invoking the kernel.

In addition to the performance increase over traditional user-space driver implementations, our method also has a possible performance advantage over kernel-level drivers employed in monolithic kernels. Without sufficiently fine-grained locking, even a kernel-level driver locks other cores out of (at least some part of) the kernel while it is servicing interrupts. In contrast, drivers loaded with privileged access to the hardware but not into the kernel can service interrupts as quickly as traditional kernel-level drivers, but without blocking other cores from entering the kernel.

This effect is similar to that reported in [21] when the network driver was isolated to its own virtual machine. In that case, letting a network-specific VM handle all network interrupts without interrupting the main kernel resulted in a performance increase, despite the added overhead of inter-VM communication. Analogous results were also found in [8, 51]

4. RELATED WORK

There is a substantial body of work that aims to thwart privilege escalation using design-, compile-, load-, and/or run-time techniques. Additionally, many efforts have explored the idea of asymmetric multiprocessing for performance purposes. Finally, some previous work suggests a new paradigm for operating system design. A brief summary of major work in each of these areas follows.

4.1 Privilege Escalation Mitigation

Often assisted by the weak separation of kernel- and user-space, all of the most popular kernels have been compromised by "rootkits" that give the attacker the highest level of privilege (i.e. "root") [66, 37, 56]. Typically, these attacks

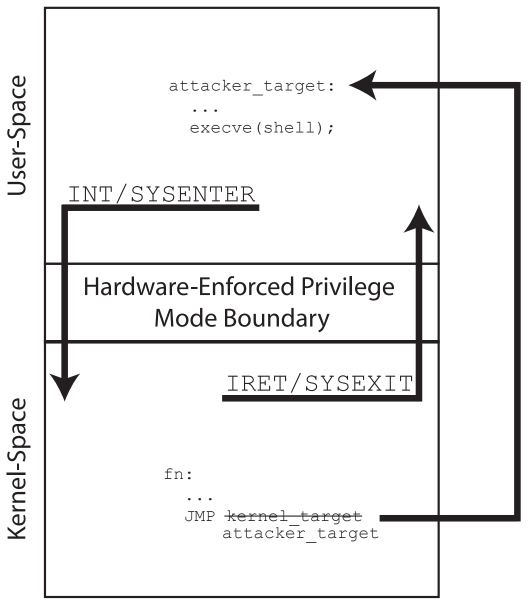


Figure 4: Return-to-User Privilege Escalation Attack

fall into three main categories: kernel code implants [42], kernel-mode return oriented programming (ROP) [14, 62, 29], and ret2usr attacks [35]. We have published a survey of techniques used to mitigate the risks of each of these attacks [12]. Since the strong separation between kernel- and user-space in our proposed operating system design directly defeats ret2usr attacks, we will summarize existing methods used to defeat this attack on modern 64-bit x86 architectures.

A return-to-user attack is directly enabled by weak kernel- and user-space separation. In this attack, illustrated in Figure 4, a user controlled target associated with some kernel-code branch is set to an address in the normal user-space code. The compromised branch creates a path of execution that leaves kernel-code and enters user-code without changing the CPU privilege level from supervisor mode to user mode. This attack results in the execution of user-controlled code with kernel-level privileges. Although hardware extensions such as Intel's SMEP [24] aim to mitigate this threat, these extensions are only slowly being adopted by operating systems and SMEP bypass techniques have already been demonstrated [58, 33]. Additionally, SMEP cannot mitigate the threat of a ret2dir style attack [34].

SecVisor [61] uses physical memory virtualization to mark only one of kernel- and user-space executable at a time. When a violation of security rules is detected, the protections are swapped and execution resumed only if the CPU has indeed changed privilege level. This defeats ret2usr attacks by ensuring a processor mode switch whenever the control flow jumps between kernel- and user-space as shown in Figure 5a. Additionally, SecVisor enforces standard W$\oplus$X rules on all kernel code pages that the user has approved. This mitigates the possibility of a kernel code implant by verifying that all executable kernel code is non-writable and has been approved for execution by the user.

The Secure Virtual Architecture (SVA) [19] is a set of architecture independent instructions that allow an operating system to interact with hardware. A kernel is ported to use these instructions, similar to porting a kernel to any new hardware architecture. Offline, an SVA compiler produces SVA byte-code from the kernel source code. This compiler has advanced features to provide memory safety and control-flow integrity at compile-time, similar to "safe" programming languages such as Java. The byte-code is distributed to users and executed on top of a virtualized SVA interpreter that performs the final step of translating to native target-dependent machine code.

Kernel Control Flow Integrity (KCoFI) [18] leverages the mechanics of the SVA implementation discussed previously, but offers only control flow integrity. Specifically, KCoFI ensures that function calls always enter at the beginning of some function's code, and that all returns from a particular function target the location of a possible call site. By verifying all kernel mode branches at run-time, KCoFI manages to deny each of the three primary privilege escalation techniques.

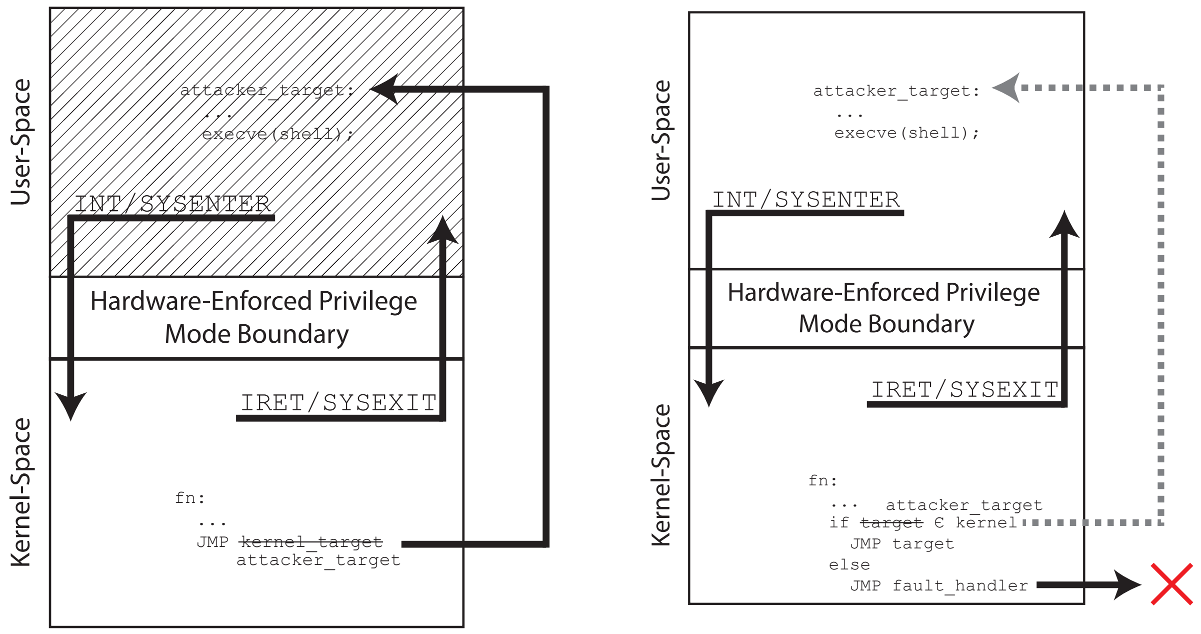
kGuard [35] aims to deny ret2usr attacks by inserting guards on the kernel's control-flow at compile time as shown in Figure 5b. On the x86 platform, the `call`, `jmp`, and `ret` instructions are all vulnerable to being hijacked in order to redirect kernel execution into user-controlled code. kGuard places an inline check before each of these instructions. kGuard also includes a compile-time code diversification mechanism that makes it difficult for the attacker to bypass its inline code checks.

4.2 Asymmetric MultiProcessing

Some research has investigated the possible merits of asymmetrical multiprocessing compared to SMP. Allocating specific software tasks to specific hardware resources is already commonly used within computing systems. Hardware units such as network cards or graphics processing units (GPUs) demonstrate how matching specific software tasks with specialized hardware can greatly increase the performance of normal computation.

In fact, many multiprocessors are built with cores that have heterogeneous performance. These processors allow many low-performance cores to lower the heat, power, and cost of the chip and provide large degrees of parallelization while a few high-performance cores provide valuable service for high-cost serial computation. Examples include the "Cell" microprocessor architecture used in the Sony Playstation and the Apple A10 Fusion chipset used in the iPhone 7. Research that examines the application of asymmetrical multiprocessing in these types of processors is plentiful [50, 5, 48]. However, research that investigates asymmetric application of software to symmetric multicore hardware resources is particularly relevant.

Early work surrounding the introduction of multiprocessors explored a master-slave relationship between processors, with the operating system running only on the master core. This is summarized in [23], where Enslow writes that "[a]lthough the master-slave type of system is simple, it is usually quite inefficient in its control and utilization of the total system resources." Fortunately, this assessment is unlikely to persist in the face of the speed and quantity of modern multiprocessors. Early attempts had limited hardware resources and could not afford to isolate a single core for the kernel alone; the kernel shared its core with general purpose processes. This resulted in delayed servicing of slave core requests and interruption of software on the mas-



(a) Using SecVisor (b) Using kGuard

Figure 5: Selected Techniques to Defeat ret2usr Privilege Escalation Attacks

ter core during slave core requests. With the low cost and high speed of processor cores modern systems, these issues are unlikely to plague the proposed design.

The master-slave paradigm is also used in [32] to implement a scheme designed to easily allow a uniprocessor operating system implementation to manage software on a multicore processor. In particular, it provides lightweight kernel implementations that record system calls made by applications on other cores, and a daemon that scans for these records and requests the appropriate computation from the kernel on the main core. This work is similar in its concept, but because it was designed to shoehorn a uniprocessor kernel onto a multicore processor, it does not realize the security or performance benefits that we believe are possible with AsMP.

The Twin-Linux project [31] examined utilizing symmetric multiprocessor cores asymmetrically by running two completely independent instances of the Linux operating system on the same CPU - each using its own subset of the processor's cores. Their work provides a thorough illustration of the flexibility of x86 IPIs and processor cores to be used in a capacity beyond their traditional use in commodity operating systems.

AsyMOS [51] assigns cores of a symmetric multicore processor to specific tasks such as network communication or disk I/O. These cores run the appropriate device driver and a Lightweight Device Kernel (LDK) that implements only the functionality that those drivers might need. The prototype described demonstrated improved performance over traditional SMP operating systems. Similarly, Corey [8] explores using specific cores to do application-specific kernel-intensive work on behalf of an important process in parallel to the application itself.

4.3 Previously Proposed Paradigm-Shifts

Some approaches, like ours, attempt to offer similar security benefits by redefining the entire paradigm rather than simply patching the existing status quo. In particular, microkernels, exokernels, and unikernels provide case studies for the possibility of an alternative kernel design.

The idea of a microkernel departs from the standard monolithic kernel architecture by emphasizing a small codebase for the operating system kernel. There have been several examples of microkernels presented in the literature such as Mach [3], Minix [27], L4 [55], QNX [28], Bear [52], and many others. All microkernels aim to keep the source code minimal in order to decrease the likelihood of vulnerabilities [54]. Additionally, small code bases allow for the possibility of using formal analysis and formal verification techniques [6, 39]. In order to keep the microkernel small, core functionality such as device drivers are migrated into user level processes. Unfortunately, this means that microkernels struggle to offer the same levels of performance as monolithic kernels.

The ExoKernel [22] suggests redefining what the kernel is entirely. Rather than providing abstractions that the application developer can use to interact with hardware, the ExoKernel provides only the thinnest possible layer necessary to manage the multiplexing of hardware resources. Although offering more security for a system overall, the ExoKernel has not become common-place, largely because it complicates the job of the application development significantly. Many of the tasks that a secure kernel can provide to protect all processes, such as virtual memory management, become the responsibility of the application developer. Even when this is handled in a library, controlling access to shared resources without a centralized authority is particularly challenging. The proposed design could actually augment the capabilities of an ExoKernel architecture, since applications would have native access to the hardware from ring 0.

Unikernels trade flexibility for security and performance by running a single process within a single address space [43]. Eliminating the requirement to support multiple processes and/or multiple users simplifies the code base required to implement a unikernel and reduces the overhead required to complete a single unit of useful work. Several examples have been deployed alongside virtualization technologies in cloud applications [9, 38, 45]. Despite their proven usefulness for providing fast, highly focused applications, unikernels don't, in isolation, provide protection from most of the attack vectors discussed in this paper. Additionally, in order to support the multiple-user multiple-job paradigm that conventional applications require to operate effectively, they require a hypervisor for scheduling and other process-management type tasks. This is an obvious instance of the often criticized "turtles all the way down" approach to system security [10, 57].

5. NEXT STEPS

The design paradigm we've described has the potential to substantively increase both performance and security. In order to measure its validity, we will develop a prototype kernel that employs it. The platform for our prototype is a research microkernel called Bear [52]. It is very small, but still supports full 64-bit multicore Intel hardware, including the Intel VT-x and VT-d virtualization features. Its small size and full feature set make it a perfect candidate for the type of aggressive kernel redesign necessary to implement a prototype of our design.

Developing the prototype will involve many challenging research and design questions. Utilizing Intel's SMP architecture to implement an unconventional AsMP design involves several challenges including:

- Using IPIs in a considerably different way from their standard uses in the bootup process, interrupt propagation, and cache coherency.
- Using the virtualization layer on a given core to stop malicious or confused privileged user processes from inappropriately manipulating hardware.
- Using the virtualization layer and IPIs to force processes to comply with the commands of the kernel. Without the kernel sharing a core, a failure to police delinquent processes might lead to spamming the kernel with syscalls or refusal to cooperate when the kernel wants to switch processes on a core.
- Bootstrapping user applications on cores without a loaded kernel.
- Merging the virtualization rules required in the proposed design with existing type-1 or type-2 hypervisors to support running this operating system in a cloud, or deploying commercial virtualization products on top of this system.
- Verifying the identity of each core during intercore communication.

We will measure the prototype's success with an estimate of its complexity, a detailed security study, and a thorough performance evaluation.

Executable size and lines of source code provide an estimate of the prototype's complexity. A small prototype is expected to be more secure, since the number of vulnerabilities in a project is directly related to its size [54]. Security is notoriously difficult to measure, but a detailed security study will consider the prototype's capabilities against known attack vectors and speculate on its possible resiliency to previously unknown "zero day" attacks.

Evaluating the performance impact of the proposed design is more straight-forward. Bear comes packaged with a suite of benchmark tests that perform CPU and memory intensive software tasks, including an adaptation of the `malloc` benchmark test presented in [40]. Additionally, the industry-standard AIM9 benchmark suite has been ported to Bear. The performance of these test suites will be compared between our prototype and the standard Bear build in order to estimate the overall performance impact of the design. We will also conduct micro-benchmarks to investigate specific system tasks instead of overall system throughput.

6. CONCLUSION

Over the past several decades, few have attempted to change the traditional operating system design methods. We believe that symmetric multiprocessing is imposing unnecessary limits on the security and performance of operating systems. In this paper, we've shown an alternative design paradigm based on asymmetric multiprocessing to offer possibilities for increased security and performance.

Our design abandons the traditional weakly-separated kernel and user virtual address spaces in favor of a strongly separated kernel- and user-space. In particular, this approach will execute the kernel on one core and applications on other cores, with the two pieces of software never sharing hardware. System calls and other communication between the kernel and the application will be conducted using IPIs instead of the traditional `INT/IRET` or `SYSENTER/SYSEXIT` methods. Finally, the processor's virtualization layer will be utilized on a per-core basis to protect the hardware from malicious processes.

We expect an implementation of our design to offer several contributions to the current state of the art. Divorcing virtual address spaces will enable complete sandboxing of each user application. Redefining "user-space" will allow for device drivers with the security of user-space encapsulation and the performance of kernel modules, and will make the ring 3 hardware protection mechanisms available for use within applications. Simultaneous operation of the kernel and applications introduces the possibility for real-time kernel watchdog security monitors for intrusion detection in applications. The asymmetric multiprocessing paradigm allows for uniquely fine-grained security policy enforcement enabled by per-core virtualization. Finally, the increased concurrency and faster system call interface suggest increased overall performance.

We plan to build a prototype kernel that uses the design presented in this paper. In order to measure its success, the prototype will be evaluated in terms of its complexity, its security, and its performance. We hope that this design and any future prototype implementation will provide a valuable case study in the sparse world of alternative operating system designs.

Acknowledgments

Many thanks to Jason Dahlstrom, Steve Kuhn, and especially Rob Denz and Martin Osterloh for support during the development of the ideas described in this paper; and to Ross Philipson, Sergey Bratus and Michael Locasto for agreeing to provide feedback on a draft of the paper. Finally, thanks to the NSPW community including, but not limited to, our anonymous reviewers and our shepherd, Matt Bishop.

This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-11-2-0257. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

7. REFERENCES

[1] CVE-2013-2094, May 2013.

[2] CVE-2016-0728, January 2016.

[3] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. pages 93–112, 1986.

[4] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, May 2005.

[6] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Formal Verification of a Microkernel Used in Dependable Software Systems. In B. Buth, G. Rabe, and T. Seyfarth, editors, *Computer Safety, Reliability, and Security*, volume 5775 of *Lecture Notes in Computer Science*, pages 187–200. Springer Berlin Heidelberg, 2009.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[8] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[9] A. Bratterud, A.-A. Walla, P. E. Engelstad, K. Begnum, et al. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257. IEEE, 2015.

[10] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. VM-based Security Overkill: A Lament for Applied Systems Security Research. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW'10, pages 51–60, New York, NY, USA, 2010. ACM.

[11] S. Brookes, R. Denz, M. Osterloh, and S. Taylor. ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*, ICCWS'16, pages 56–66, April 2016.

[12] S. Brookes and S. Taylor. Containing a Confused Deputy on x86: A Survey of Privilege Escalation Mitigation Techniques. *IJACSA*, April 2016.

[13] Z. Brown. Asynchronous System Calls. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 81–85, 2007.

[14] E. Buchanan, R. Roemer, S. Savage, and H. Shacham. Return-Oriented Programming: Exploitation without Code Injection, 2008.

[15] 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.

[16] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM.

[17] P. Crawford. Linux Watchdog Daemon - Overview. http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-background.html, January 2016.

[18] 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, May 2014.

[19] 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.

[20] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, 1968.

[21] R. Denz. *Securing the Cloud with Utility Virtual Machines.* PhD thesis, Thayer School of Engineering at Dartmouth College, July 2016.

[22] 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.

[23] P. Enslow, Jr. Multiprocessor Organization – a Survey. *ACM Comput. Surv.*, 9(1):103–129, Mar. 1977.

[24] S. Fischer. Supervisor Mode Execution Protection. NSA Trusted Computing Conference and Exposition, 2011.

[25] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proceedings of the 20th Conference on Large Installation System Administration*, LISA '06, pages 12–12, Berkeley, CA,

USA, 2006. USENIX Association.

[26] J. N. Herder. *Towards a true microkernel operating system*. PhD thesis, Vrije Universiteit Amsterdam, February 2005.

[27] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A Highly Reliable, Self-repairing Operating System. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[28] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[29] R. Hund, T. Holz, and F. C. Freiling. Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

[30] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*, June 2014.

[31] A. Joshi, S. Pimpale, M. Naik, S. Rathi, and K. Pawar. Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system. In *Proceedings of the Ottawa Linux Symposium*, 2010.

[32] S. Kagstrom, L. Lundberg, and H. Grahn. A novel method for adding multiprocessor support to a large and complex uniprocessor kernel. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 60–, April 2004.

[33] keegan. Attacking Hardened Linux Systems with Kernel JIT Spraying, June 2011.

[34] 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.

[35] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[36] D. Keuper. XNU: a security evaluation, December 2012.

[37] J.-J. Khalife. MS15-010/CVE-2015-0057 win32k Local Privilege Escalation, December 2015.

[38] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv-optimizing the operating system for virtual machines. In *2014 usenix annual technical conference (usenix atc 14)*, pages 61–72, 2014.

[39] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, Feb. 2014.

[40] C. Lever and D. Boreham. Malloc() Performance in a Multithreaded Linux Environment. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 56–56, Berkeley, CA, USA, 2000. USENIX Association.

[41] R. Lindsley and D. Hansen. Bkl: One lock to bind them all. In *Ottawa Linux Symposium*, page 301, 2002.

[42] A. Lineberry. Malicious Code Injection via/dev/mem. 2009.

[43] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[44] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[45] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. Enabling fast, dynamic network processing with clickos. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 67–72. ACM, 2013.

[46] R. McDougall and J. Mauro. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education, 2006.

[47] metasploit. Chkroot Local Privilege Escalation, November 2015.

[48] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, 2008.

[49] I. Molnar. 4G/4G split on x86, 64 GB RAM (and more) support, July 2003.

[50] T. Morad, U. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 5(1):14–17, Jan 2006.

[51] S. Muir and J. Smith. AsyMOS-an asymmetric multiprocessor operating system. In *Open Architectures and Network Programming, 1998 IEEE*, pages 25–34, Apr 1998.

[52] C. Nichols, M. Kanter, and S. Taylor. Bear – A Resilient Kernel for Tactical Missions. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 1416–1421, Nov 2013.

[53] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding Collateral Evolution in Linux Device Drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 59–71, New York, NY, USA, 2006. ACM.

[54] R. K. Pandey and V. Tiwari. Article: Reliability Issues in Open Source Software. *International Journal of Computer Applications*, 34(1):34–38, November 2011. Full text available.

[55] D. Potts, S. Winwood, and G. Heiser. Design and Implementation of the L4 Microkernel for Alpha Multiprocessors, 2002.

[56] rebel. issetugid() + rsh + libmalloc osx local root, July 2015.

[57] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and Virtue. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 4:1–4:6, Berkeley, CA, USA, 2007. USENIX Association.

[58] D. Rosenburg. SMEP: What is it, and How to Beat it on Linux., June 2011.

[59] J. H. Saltzer and M. D. Schroeder. The protection of

information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[60] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, (2):38–47, 1996.

[61] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *SIGOPS Oper. Syst. Rev.*, 41(6):335–350, Oct. 2007.

[62] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[63] K. Spett. Cross-site scripting. Technical report, SPI Labs, 2005.

[64] A. Starke. Locking in os kernels for smp systems. Citeseer, 2006.

[65] K. Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, Aug. 1984.

[66] K. Way. Lastore-Daemon in Deepin 15 results in privilege escalation, February 2016.