Bear – a Resilient Core for Distributed Systems

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

by

Colin Nichols

Thayer School of Engineering

Dartmouth College

Hanover, New Hampshire

January 2013

Examining Committee:

Chairman________________________
Stephen Taylor, Ph.D.

Member________________________
George Cybenko, Ph.D.

Member________________________
Andrew Campbell, Ph.D.

____
F. Jon Kull
Dean of Graduate Studies

Abstract

This paper describes Bear, a clean-slate, resilient operating system design intended to support military applications on scalable multi-processors. The system combines a minimalist micro-kernel with an associated hypervisor, and presents only a 120Kbyte attack surface on 64-bit x86 blade servers. MULTICS-like protections are strictly enforced through extended page tables and Intel VT-x extensions. The design utilizes multiple, overlapping, non-deterministic techniques to continually re-establish trust. This is achieved by dynamically regenerating core components of distributed computations and their underlying execution environment. The cumulative effect of this design style is to increase attacker workload by denying surveillance and persistence over time-scales consistent with tactical operations. Unlike traditional approaches to computer security, no attempt is made to detect intrusions: instead, we focus on continually validating, preserving, and re-establishing the ability of a mission to proceed.

Acknowledgements

The work draws its microkernel inspiration from the MINIX system and the many valuable insights that this system provides to operating system developers. The research also draws on the thoughts and work of a talented group of graduate students at Dartmouth, in particular Morgan Kanter who developed the hypervisor core, bootloader, and explored camouflage; Stephen Kuhn who explored kernel regeneration in the context of web-servers and forensics; Kathleen Mcgill who explored application level resiliency; and finally Michael Henson and Jason Dahlstrom who ported the Bear kernel to the ARM M3, A8, and A9 architectures. We sincerely thank all concerned for their help and participation.

This research is supported by the Defense Advanced Research Projects Agency (DARPA) as part of the CRASH program under contract FA8750-09-1-0213. 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.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	v
List of Figures	vi
List of Acronyms	vii
Introduction	1
Overview of the System Design	2
Protecting the Micro-kernel	3
Message-passing API	4
Attack Mitigation	5
Mitigation of Corrupted Device Drivers	6
Protecting the Hypervisor	9
DMA Protection	12
Quantifying the Attack Surface	15
Extending Trust to Distributed Applications	17
Conclusion	19
References	21

List of Tables

Table 1: Page Table Protection Configurations………………………………………. 4

Table 2 Kernel Comparison: Lines of Code……..………..……………………………..15

Table 3: Hypervisor Comparison: Lines of Code………..………………………………15

Table 4: The AIM9 Benchmark Suite ……………………………………………..… 16

Table 5: Performance Study Results…………………………………………………… 17

List of Figures

Figure 1: Threat Model for Intrusions with Remote Control…………………………..... 1

Figure 2: The Bear Operating System………………………………………………… 2

Figure 3: Page Table Entry for x86-64 Address Translation……………....................... 3

Figure 4: Virtual Address Space During Process Execution……………………………. 4

Figure 5: EPT provides the guest-physical address space (yellow), allowing the hypervisor to control physical memory without cooperation from the guest OS……….10

Figure 6: Guest-physical address space protections enforced via EPT…………………10

Figure 7: Intel VT-d modifies the standard computer architecture………………………12

Figure 8: A typical DMA transfer on a system without virtualization…………………..13

Figure 9: The NIC card using a driver-supplied GP address as a P address, corrupting memory……………………………………………………………………………..13

Figure 10: DMA transfer from guest-resident driver to NIC card with EPT and VT-d...14

Figure 11: EPT Workaround for DMA – kernel requests P address from hypervisor ...…………………………………………………………………………..14

Figure 12: Dynamic process regeneration………………………………………………15

List of Acronyms

Extended Page Tables	EPT
Read Only Memory	ROM
Internet Protocol	IP
Media Access Control	MAC
Message Passing Interface	MPI
Application Programming Interface	API
Video Graphics Array	VGA
Return Oriented Programming	ROP
GNU’s Not Unix	GNU
Basic Input/Output System	BIOS
Central Processing Unit	CPU
Peripheral Component Interconnect	PCI
Direct Memory Access	DMA
Commercial Off-The-Shelf	COTS
Input/Output Memory Management Unit	IOMMU
Network Interface Card	NIC
Light Detection and Ranging	LIDAR

Introduction

Current operating system designs have sought to utilize a *static* base of trust and extend trust into software through deliberate layering [1]. Unfortunately, a wide variety of vulnerabilities have appeared that undermine kernel security allowing attackers to implant code, hide, and persist at the highest levels of privilege [2]. The number of vulnerabilities is directly correlated with the size of the code base [3], indicating that there is substantial value in the intellectual process of *reducing the attack surface*.

Irrespective of the implant design, there are only two fundament use cases: performing a triggered effect autonomously, or conducting effects under remote control. The first case is of limited use and is analogous to any other general failure or error; it can, and routinely is, combated by skilled network administrators through diversity, gold-standard images, and/or spare equipment. The second, more interesting case, can be mitigated by denying or degrading remote control: increasing attacker workload to the point where there can be no significant impact on the time-scale of tactical operations.

The threat model for intrusions employing remote control is outlined in Figure 1. It may involves several steps including *surveillance* to determine if a vulnerability exists, use of an appropriate exploit or other access method, and privilege escalation to remove exploit artifacts and/or hide behavior. The implant then *persists* for a time sufficient enough to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems. Unlike the time to execute an exploit, the time spent in surveillance and persistence may range from minutes to *months or even years* depending upon the intended effect. Moreover, the presence of an intrusion may *never* be detected by network defenses but instead may be recognized due to either a deviation from expected behavior, or may be derived from intelligence sources.

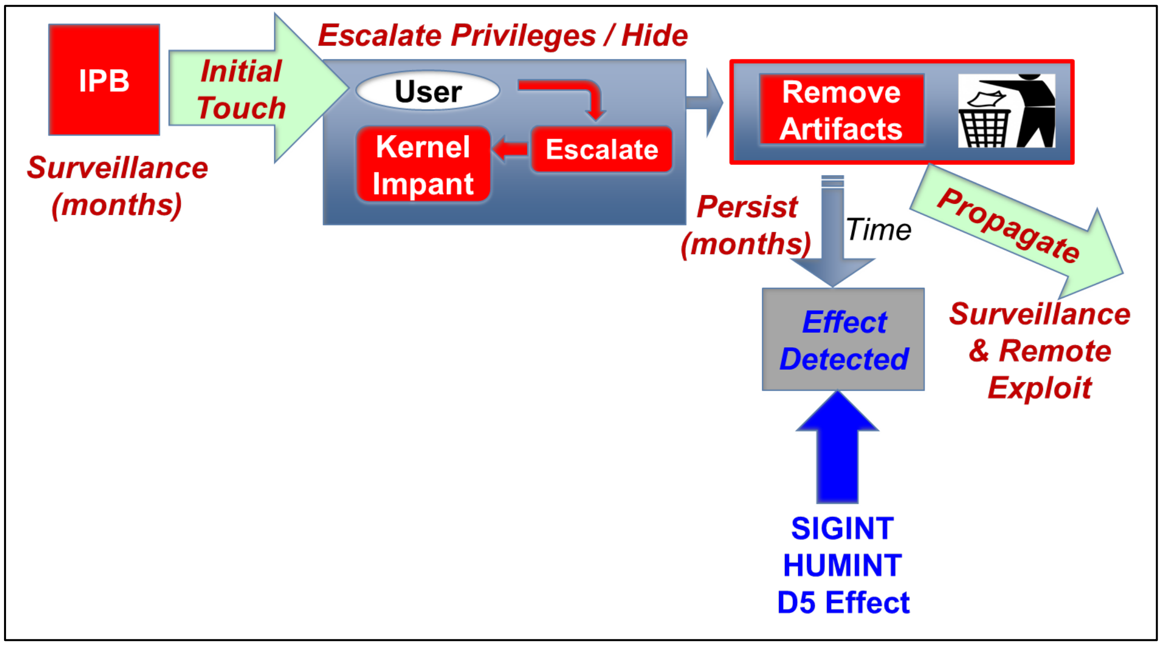

Figure 1: Threat Model for Intrusions with Remote Control

Overview of the System Design

Our approach assumes that adversaries will conduct *surveillance*, will be successful in gaining access, and will *persist undetected*. To mitigate the risks associated with remote control, we periodically discard the current kernel, user processes, and device drivers. They are replaced by new instances, bootstrapped in the background from read-only gold standards. The cumulative effect of this change in design style is to *increase attacker workload* by continually invalidating surveillance data and denying persistence over time-scales consistent with tactical missions. Unlike other approaches to computer security, no attempt is made to detect intrusions: instead, we focus on continually validating, preserving, and re-establishing the ability of a mission to proceed.

These concepts have been incorporated into a new, from-scratch operating system design -- ***Bear*** -- that operates on 64-bit, x86 multi-core blade servers. The system is depicted in Figure 2 and is composed of a minimalist *micro-kernel* with an associated *hypervisor* that share code extensively to reduce the attack surface.

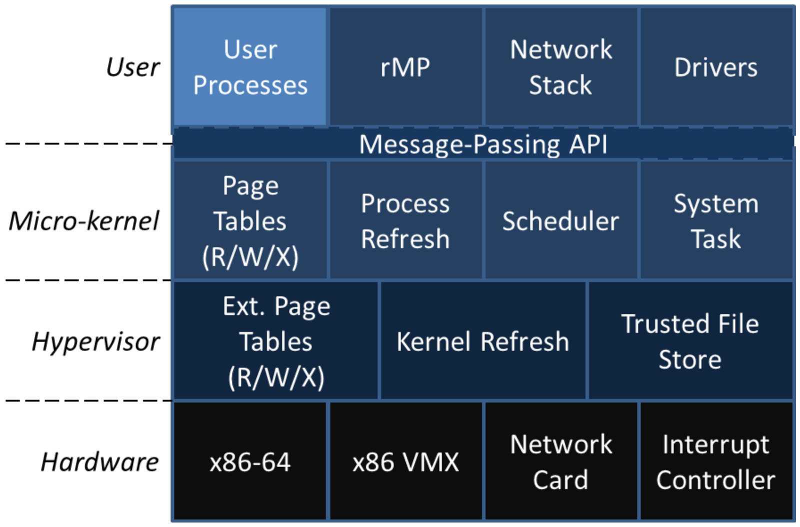

Figure 2: The Bear Operating System

The core functions of scheduling user processes and protecting them from each other are handled by the micro-kernel. All processes and layers are hardened by strictly enforcing MULTICS-style read, write, and execute protections [4] using 64-bit x86 address translation hardware. This calculated reduction in versatility is unlikely to impact military applications but explicitly removes vulnerabilities associated with code execution from the heap or stack.

All potentially contaminated user processes, device drivers and services are executed with user–level privileges and are strictly isolated from the micro-kernel via a message-passing interface. The system task, executing with kernel privileges, mediates between processes and the kernel to implement the interface. Unlike a conventional rendezvous mechanism [5], this asynchronous, buffered design provides a single uniform treatment of system calls, inter-process, and inter-processor communication. The interface also supports distributed computing through an MPI-like [6] programming model that maps processes to processors using a user level demon, *rMP*.

To prevent persistence in compromised device drivers and services, the micro-kernel randomly and non-deterministically regenerates them from gold-standard images resident in a *trusted read-only file store*. This store is currently realized through a file system accessible only from the kernel and hypervisor; however, it could alternatively be realized via read-only memory (ROM) or via an out-of-band, write-enabled channel to flash on new hardware. Unlike the MINIX re-incarnation process [5], regeneration is carried out without regard to the perceived fault or infection status. User processes can also be refreshed through pre-arranged or designated schedules; for example, every few hours, at night, or just prior to a tactical mission.

To prevent persistence in the micro-kernel, it is also non-deterministically refreshed from a gold-standard image in the trusted file store, by the hypervisor. Unlike traditional hypervisors, which are intended to support a general virtual machine execution environment [7, 8, 9], this minimalist hypervisor is designed to support *only* the operations required to bootstrap a new micro-kernel and change its network properties (e.g. IP & MAC address) so as to invalidate an adversary's surveillance data. The current running and bootstrapping instances of the micro-kernel are isolated in hardware through extended page tables, implemented with Intel VT-x extensions. Similarly, the network card is isolated through a mapping scheme based on Intel VT-d extensions.

Protecting the Micro-kernel

The micro-kernel architecture leverages the latest x86-64 address translation hardware to provide isolation and MULTICS-style read, write, and execute (R/W/X) privileges for processes. Recent x86-64 processors no longer support segmentation, but they do feature control bits that enable the kernel to allow or deny reading, writing, and execution of a particular memory page. This is achieved using three protection bits in x86-64 page table entries, shown in Figure 3.

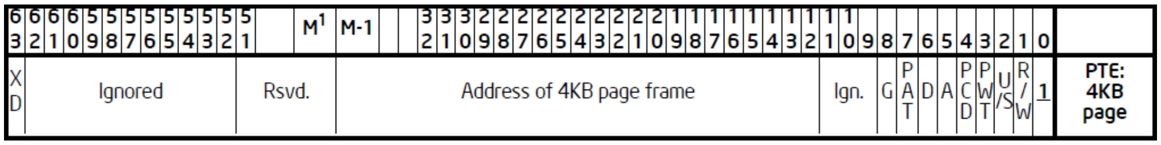

Figure 3: Page Table Entry for x86-64 Address Translation.

To isolate user processes from the kernel, the kernel clears the user/supervisor bit (U/S, bit 2) on its own pages. If any user process attempts to read, write, or execute code in these pages, the processor traps to the kernel. Bear enforces MULTICS-style protections for process memory using the read/write (R/W) and execute disable (XD) bits. When a process is loaded, the bits are set so that the process text (code) is readable/executable. Conversely, process data and stack are designated readable/writeable. These decisions yield the protected address space illustrated in Figure 4. The corresponding permission bit configurations are also shown below in Table 1.

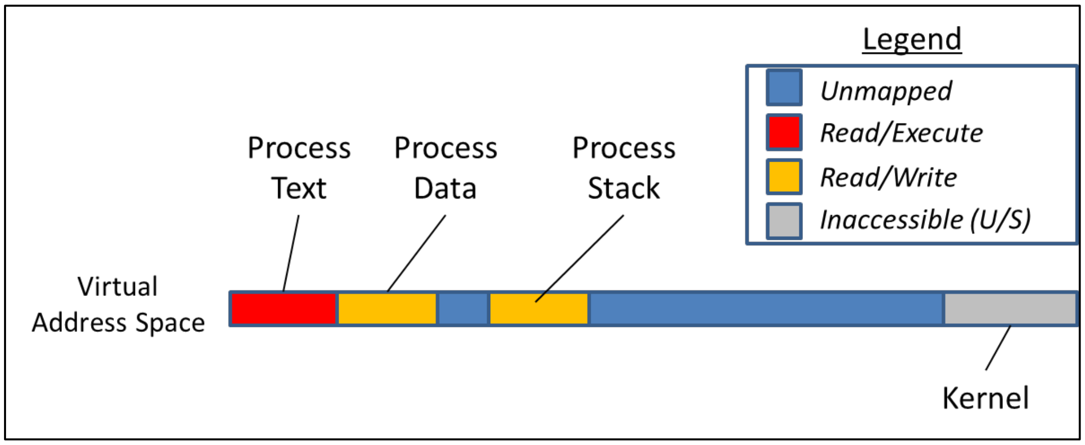

Figure 4: Virtual Address Space During Process Execution.

Memory Type	U/S Bit Value	R/W Bit Value	XD Bit Value
User Process Text	1	0	0
User Process Data	1	1	1
User Process Stack	1	1	1
Kernel Text	0	0	0
Kernel Data	0	1	1
Kernel Stack	0	1	1

Table 1: Page table protection configurations.

Message-Passing API

The memory space of each process is strictly isolated from that of other processes and the micro-kernel by page protections. All processes interact via a simple MPI-like asynchronous message-passing interface [6]. This allows the same isolation ideas to be used for inter-process communication within the same processor, across multiple processors, and between user processes and the kernel. The interface provides only two asynchronous, blocking, communication primitives:

- **msgsend(dest, &sendbuffer, size)** – send a message from *sendbuffer* of length *size* bytes to process *dest*.
- **msgrecv(src, &recvbuffer, size, &status)** – receive a message from process *src* (or ANY process) into *recvbuffer* of length *size*; *status* is a structure designating the *source* of the message and its *length*, messages that are larger than *size* are truncated.

Both primitives are realized using software interrupts that isolate user-processes from the micro-kernel. All messages are buffered in the kernel at the *receiver*. The *msgsend* operation causes a process to be blocked until a message is sent (i.e. injected into the kernel, if the receiver is at the same host, or the network if it is on a remote host). Return from this primitive allows the *sendbuffer* to be re-used. The *msgrecv* operation causes a process to be blocked until a message is transferred into the *recvbuffer* from the kernel. System calls, such as -- fork(), exec(), and exit() -- are implemented by sending a message to a designated *system task (c.f. Figure 1)* which is capable of modifying kernel data structures (e.g. pages, scheduling queue's etc); distributed computing is achieved by forwarding messages to a remote host via a mapping process *rMP* (c.f. *Figure 1).*

The micro-kernel leverages *user-space separation of privilege* to minimize kernel size. In this approach, device drivers are given only the access rights needed to operate. Thus they require *no kernel intervention* other than startup in order to execute. This allows system calls serviced by user-space processes – the network stack, the filesystem, and so on – to operate entirely in user-space. Borrowing terms used by MINIX, the system call *policies* remain in user-space, but are joined by the system call *mechanisms,* in the form of *entire* device drivers. Consequently, a significant amount of privileged code is excised from the kernel, creating a small attack surface with few entry points.

It is instructive to contrast this approach with that used in MINIX: a number of user-level tasks service system calls. These tasks – such as the process manager, filesystem, info server, and so on – enforce system call *policies* and carry out bookkeeping, but they do not contain the actual *mechanisms* to carry out a system call. That is left up to either drivers or the kernel. However, even drivers are reliant on kernel code to perform their functions, and they have their *own* set of system calls that are directly serviced by the kernel. The result is a small reduction in kernel code and data, but a significant increase in complexity.

Currently, Bear provides three user-level processes that service system calls. The network process handles network connectivity and BSD-style socket calls, while the keyboard and VGA processes handle user I/O calls directly. The keyboard and VGA processes are stopgap solutions that are being used to bootstrap the system; they will eventually be replaced by a secure shell implementation that will be the only method for interacting with the system. A simple network file-system client is currently under development, which will provide the only file storage mechanism available to user processes.

Attack Mitigation

Despite efforts to insulate the kernel from user processes, there are still methods to get code into the kernel memory space. For instance, while carrying out inter-process communication, the kernel may buffer user data in kernel memory-space. Furthermore, a hardware implant could potentially inject code directly into kernel memory. Once kernel memory is contaminated, an attacker need only find a method to divert kernel execution to this code.

Bear's treatment of kernel memory is designed to expressly deny this avenue of attack and increase attacker workload. At all levels, Bear enforces the policy that no memory region may be both *writeable and executable* simultaneously. In the Bear kernel, there are four classes of buffers: those created by the kernel's small-memory allocator, those created by the kernel's large-memory (page) allocator, static buffers in the kernel binary, and temporary buffers located on the kernel stack. The small-memory allocator is used to dynamically allocate space for data structures within the kernel (e.g., message buffers, process structures, hash tables, linked lists, etc.). All memory regions returned by the small-memory allocator are protected from execution by the XD bit in the kernel page tables. The large-memory allocator provides free pages (or multiple pages) for process or kernel use. If used by the kernel, pages from this allocator are protected from execution via the XD bit in the kernel page tables. Static buffers in the binary and dynamic buffers on the kernel stack are similarly protected from execution via the XD bit in the kernel page tables. Thus, no buffers have both write and execute permissions enabled.

It is well-known that robust memory protections are not enough to secure a system from return-oriented programming (ROP) even in the presence of non-executable buffers [10]. These attacks leverage small sections of the code already resident in memory, known as gadgets [11]. The payload of a ROP exploit is a series of specially-crafted return addresses, which link together gadgets to perform whatever action the attacker desires. ROP exploit development is facilitated by a large codebase, such as GNU Libc (glibc) [11].

To increase the difficulty of crafting these attacks, we emphasize the reuse of *common data structure* abstractions throughout kernel and hypervisor so as to reduce the attack surface. Generic implementations of common data structures, including a linked list and hash table, were created with flexibility in mind. Application-specific data is always stored in these structures through the use of opaque void pointers, and application-specific functionality is added through the use of function pointers in the API. The result is lean, robust, multi-purpose code; for example, the function for removing a process from the scheduler is also the function for removing an element from a hash table.

Mitigation of Corrupted Device Drivers

Unfortunately, device drivers are a frequent source of vulnerability [12]; they are always resident and often developed by third-party vendors, whose priorities are fast turnaround, inter-operability and performance, rather than security. Recall that the Bear micro-kernel refreshes each device driver at nondeterministic intervals. This allows the kernel to *operate through* attacks, preserving trust while denying the attacker the ability to persist over tactically relevant timescales.

The upper and lower bound on the duration of a device driver instance is configurable, and could be set higher or lower based on threat or mission deadlines. Driver refresh is achieved by interrupting the driver, freeing its memory, and re-allocating new resources for its replacement. The kernel then loads the driver's gold-standard image from a protected, read-only store. As a result, compromised drivers are not able to persist over long time-scales. Once driver regeneration is complete, the kernel schedules the driver, and normal operation is resumed. Although the hardware state is lost, this is not typically detrimental to a system functioning. In a server environment, it may involve a few dropped packets, but these will be re-transmitted by normal protocols. Down-time associated with refreshing the driver could be minimized by creating the new driver process in the background using underutilized computing cores, although this has not yet been necessary.

The main objectives for driver design in Bear are to protect the operating system from corruption, encapsulate the device driver using hardware mechanisms, and facilitate on-the-fly refresh of the drivers. Putting the driver in an isolated user-level process and utilizing process refresh techniques accomplishes most of these goals. Unfortunately, a compromised device driver has unique hardware resources at its disposal that open up avenues of attack not available to most user processes.

Traditionally, the x86 architecture provides four rings (or levels) of privilege, numbered 0 through 3. Processes on the outside ring are the least-privileged and have no access to critical functionality, while the innermost ring has full privileges. For obvious reasons, user processes usually reside in the outermost ring 3, and the operating system resides in ring 0. When considering where to put device drivers, rings 1 and 2 appear to be likely candidates. Unfortunately, upon close inspection of hardware support for rings 1 and 2, it was discovered that ring 0 is *not* truly protected from code running in the intermediate rings. Intel's memory management unit only supports two access levels – user (ring 3) and supervisor (rings 0, 1, and 2). Thus, code running in rings 1 and 2 has exactly the same memory access privileges as the kernel. This violates one of Bear's primary design principles – namely, complete isolation of device driver code from the kernel.

After searching, we discovered a workaround that restricts rings 1 and 2 to *read-only kernel access*. There is a processor control bit that allows ring 0 code to ignore the read/write control of a given part of memory, and this bit may only be modified from ring 0. By setting the kernel memory area to be read-only, ring 1 or 2 code would be unable to modify kernel code or data. Upon entry to the kernel by interrupt or exception, the processor control bit would be flipped to allow modification of the kernel data. Unfortunately, this workaround has several problems. Giving read access to device drivers is not ideal; additionally, the workaround would disable hardware write-protection for kernel code while in kernel mode, leaving the door open to code corruption. Instead, we chose to place device drivers in ring 3. Rings 1 and 2 actually provide few meaningful benefits compared to ring 0. In contrast, ring 3 provides complete isolation from the kernel through hardware mechanisms.

On modern commercial off-the-shelf (COTS) hardware, drivers often rely on several overlapping mechanisms to communicate with a device: interrupts, port I/O, memory-mapped I/O, and direct memory access (DMA). Peripheral devices use interrupts to signal to the driver that they need attention; the request is then usually serviced through one (or a combination) of the other methods. Port I/O uses special CPU instructions to access a "port address space" that is completely separate from main memory. At boot time, peripheral devices are mapped into the port address space by the BIOS. In memory-mapped I/O, the BIOS instead maps peripheral devices directly over main memory; accordingly, device registers can be read and written to with regular load/store CPU instructions. DMA dispenses with need for CPU intervention altogether by giving devices direct read/write access to physical memory.

The x86-64 processors have several mechanisms to allow operating systems to monitor peripheral I/O. The *I/O permission bitmap* controls access to individual I/O ports and thus individual hardware peripherals. The bitmap may only be modified by code at privilege level 0 (i.e., the operating system kernel). Any attempts to access blocked ports or to modify the bitmap while at another privilege level will trigger an exception that may be caught by the kernel. The offending driver could then be discarded and refreshed, or some other action taken. Additionally, the x86 paging structures allow the operating system to control memory-mapped I/O. Peripheral devices are mapped in at the *physical* address layer; meanwhile, all CPU code accesses memory at the *virtual* layer. The operating system controls the mapping between virtual to physical layers, meaning it can expose or hide memory-mapped peripherals at will. Accessing a "hidden" physical address is prohibited by address translation hardware, and code executing above privilege level 0 is unable to modify the virtual-to-physical mapping without kernel intervention.

PCI resources present a unique challenge for driver encapsulation and isolation. All devices on the PCI bus share a configuration space that provides device enumeration and basic device communication via port I/O. Currently, Intel's hardware mechanisms are too coarse-grained to allow access to a single PCI device; they allow either all or none. Thus, a malicious or unstable driver process could disrupt the function of other hardware resources via the PCI configuration registers. This issue could be resolved by trapping to the kernel and validating all accesses to PCI configuration ports, since they are at well-known locations. It would be straightforward to enforce device-level separation using this method; however, doing so would incur the overhead of a trap on every PCI configuration-space access. For some drivers, this could incur small but non-negligible overhead.

More alarming is the lack of control over DMA. Until recently, drivers were able to command a device to read/write to *any physical address* via DMA. In most modern PCs, only the number of address lines on the bus limits a peripherals access to memory. Thus, on most machines, devices can read or write to *any address*. Mechanisms to limit DMA access have recently become available in COTS hardware. The centerpiece of device protection is the input/output memory management unit (IOMMU), which provides a layer of address translation and access control between devices and physical memory. On Intel platforms, the IOMMU is part of a larger set of device virtualization technologies known as VT-d. Unfortunately, our available hardware does not support this technology; however, an alternative interface between the kernel and hypervisor was implemented, and a complete IOMMU solution could be added to the hypervisor with little or no system design modifications. In order to correctly configure an IOMMU, the hypervisor must know what memory addresses are being used for DMA. In Bear, the kernel relays this information to the hypervisor via the standard *vmcall* instruction. The details of this work-around are described below in the section "Protecting the Hypervisor."

To demonstrate the use of the user-level driver structure, three device drivers were written for Bear: a VGA terminal driver, a keyboard driver, and a network interface card (NIC) driver. At startup, the kernel modifies the drivers' privileges so they can access their respective hardware. Unlike traditional drivers, they do not have access to kernel code, kernel data, any other peripheral hardware interfaces, privileged instructions, or control registers. The *NIC driver process* encapsulates Broadcom's *bce* driver ported from BSD, augmented with a front-end that communicates via message passing. One driver process is spawned per NIC card present on the system (our blade servers have two each). The driver utilizes all four forms of device communication: interrupts, port I/O, memory-mapped I/O, and DMA. The kernel only intervenes for interrupts: interrupts are translated into messages and sent to the corresponding driver process. Similarly, the VGA and keyboard drivers encapsulate their respective hardware; we regard these as stopgaps until full SSH support is available.

Protecting the Hypervisor

Recall that the normal role of virtualization is to share the underlying hardware between multiple operating system instances. In contrast, the Bear hypervisor exists primarily to undermine network surveillance, deny persistence in the micro-kernel, and reestablish trust in the micro-kernel. Re-establishing trust is performed by periodically reloading the micro-kernel from gold-standard images located in the read-only store. This has the effect of expunging root-kits, bots, or other malware. Additionally, the hypervisor strives to utilize all available hardware mechanisms to provide protection for both itself and the kernel.

To mitigate the threat of well-timed attacks, the hypervisor refreshes the kernel at nondeterministic intervals. The upper and lower bound on the duration of a kernel instance is configurable, and could be set higher or lower based on the threat environment. To achieve kernel refresh, the hypervisor assumes control of the system, frees the memory associated with the previous kernel, and allocates resources for the next kernel. The hypervisor then loads the kernel binary and relinquishes control to the kernel, which boots and resumes normal operation. Due to its code size, the microkernel boots in less than 1 second; consequently, there was little reason to leverage multiple cores to perform booting in the background. Currently, the hypervisor loads the kernel binary from a standard SATA drive. Although sourcing a drive with a hardware write-protect switch would have been ideal, we were able to emulate write-protection through software. Although not as secure, it allowed us to verify read-only operation.

The hypervisor also provides *protection* for the kernel by leveraging extended page tables (EPT). EPT is a hardware address translation capability present in newer Intel CPUs (AMD has similar technology). EPT provides an extra layer of address translation that is transparent to the guest operating system. This allows a hypervisor to manage physical memory while giving the guest the illusion of physical memory access. EPT also allows the hypervisor to control what type of operations are allowed for a given memory region, opening the door to MULTICS-like protections on the kernel.

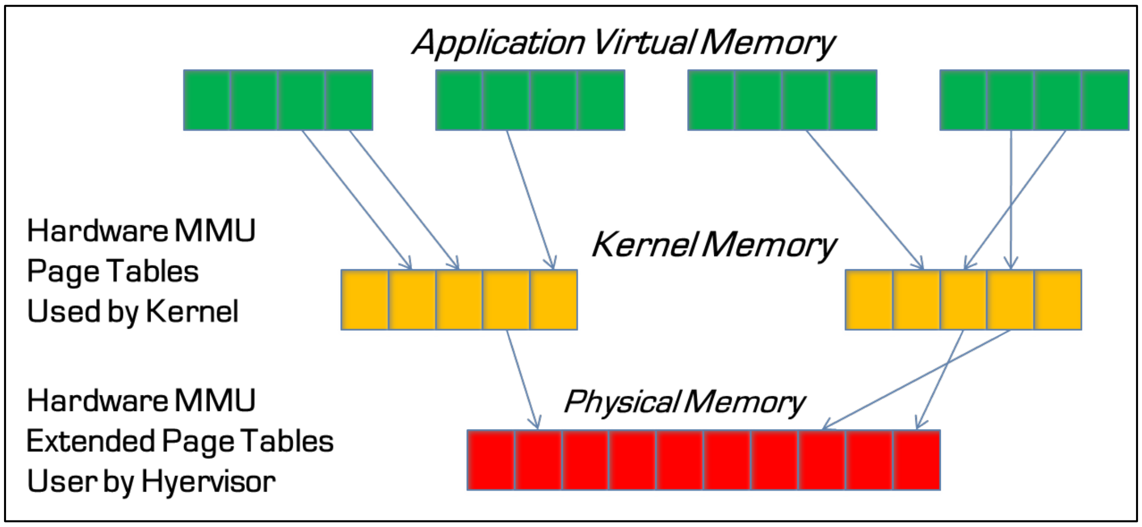
Application Virtual Memory
Hardware MMU Page Tables Used by Kernel
Kernel Memory
Hardware MMU Extended Page Tables User by Hyervisor
Physical Memory

Figure 5: EPT provides the guest-physical address space (yellow), allowing the hypervisor to control physical memory without cooperation from the guest OS.

Bear's hypervisor configures EPT to provide MULTICS-style read/write/execute controls on both the kernel code and static data. Thus, any attempt to patch the kernel or execute code located in a static buffer will result in a trap to the hypervisor. At that point, the hypervisor can refresh the kernel or take an alternative action, such as invoke forensic tools [13]. Hypervisor memory is inaccessible from the guest; it is not even mapped into the address space. Figure seven shows the guest-physical address space after configuration by the hypervisor.

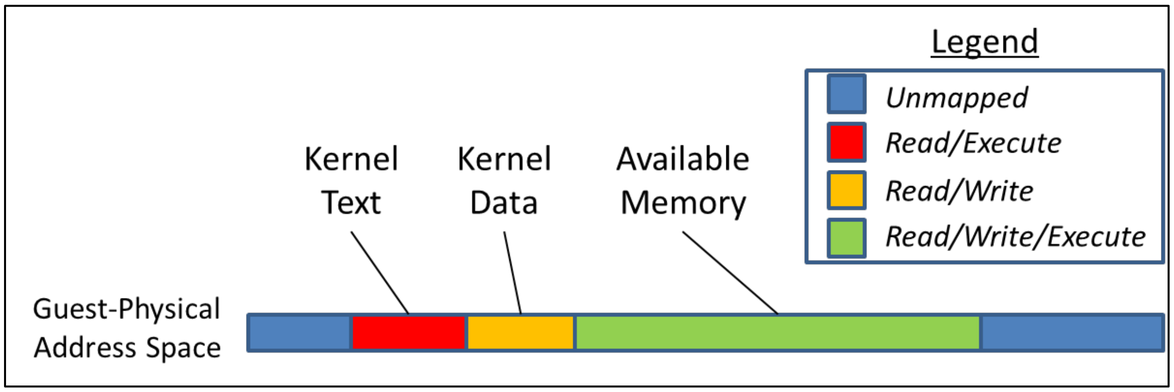

Figure 6: Guest-physical address space protections enforced via EPT.

During development, it was noted that a small change to EPT functionality could greatly improve the utility of execute protection: Almost all guest memory must stay marked as executable (and writeable) because at boot time it is unknown which pages will become kernel data and which will become user process text (code). However, user process memory can also be execute-protected via the kernel's page tables – little is gained by

EPT's double-coverage. If EPT's execute protection were limited to kernel operation (CPL=0) *only*, then *all* of the guest's available memory (in addition to kernel data) could be marked as no-execute in the EPT. In such an environment, operating a rootkit at kernel level would be exceedingly difficult.

Although the proposed EPT no-execute functionality could be emulated by the hypervisor, it would incur high overhead. The best solution would be hardware modification of EPT functionality by Intel. Were this implemented, normal rootkit operation would result in a trap to the hypervisor. Malware designers would have to craft an entire malware payload using return-oriented programming or some other method of circumventing execute-disable – this is no small task.

Currently, micro-kernel regeneration always uses the same micro-kernel image to deny persistence and re-establish trust. However, nothing prevents the hypervisor from non-deterministically varying the system configuration it brings up. In particular, each new micro-kernel instance may use a completely different micro-kernel image. Moreover, the presence of multiple NIC cards in the underlying hardware allows each new instance to non-deterministically choose an alternative network connection. These may be physically connected to completely different network segments, potentially behind different external firewalls and proxies. From a surveillance perspective, the operating system appears to be a completely different machine, running a different operating system, available for only a short period at different parts of the network. This invalidates surveillance data with every move, in the style of pioneering work conducted at BBN [14]. Our previous research has already demonstrated these forms *regeneration* and *network hiding* for the difficult end case associated with web servers, providing static pages, streaming, and stateful content [15].

Current hypervisors do not provide convenient support for dynamically switching network cards and introspection into connection information. Their role is to provide a general sharing mechanism for the underlying network hardware in much the same way as a bridge. The more simple multiplexing operations described here offer the opportunity not only to inspect traffic but also change its characteristics for the purpose of *deception*.

The traffic may project a completely different micro-kernel from that which is actually executing. Camouflage may also project known vulnerabilities and be associated with detection software. Our research group has already explored the concept of application-level deception in a proof of concept *camouflage* module that presents a false server fingerprint [16]. The camouflage has been demonstrated by disguising a Microsoft Exchange 2008 server running on Windows Server 2008 RC2 to appear as a Sendmail 8.6.9 server running on Linux 2.6. It was able to reliably deceive Nessus OS detection, Nmap OS detection and service detection, and RING OS detection into incorrectly identifying the Exchange server.

DMA Protection

In addition to regeneration and protection, the hypervisor must provide protected mechanisms for device communication, including DMA. Virtualization software has struggled with the problem of DMA for several years. Allowing guests to have access to hardware resources traditionally involved "giving away the store," since DMA could be used to inspect and patch the hypervisor. Until recently, the working solution was to plant a "thin" driver in the guest and block access to the actual device. The hypervisor would operate the real peripheral and redirect data into the guest. Although functional, this configuration introduces overhead and causes the hypervisor attack surface to balloon significantly; all supported devices have to include a driver in the hypervisor.

Intel and AMD have both independently addressed this issue; their solutions are **AMD-Vi** and **Intel VT-d**, respectively. These hardware standards both require an I/O memory-management unit (IOMMU), which adds a layer of VMM-controlled address translation between peripheral devices and main memory as shown in Figure 7. Thus, devices can only access memory ranges designated by the hypervisor.

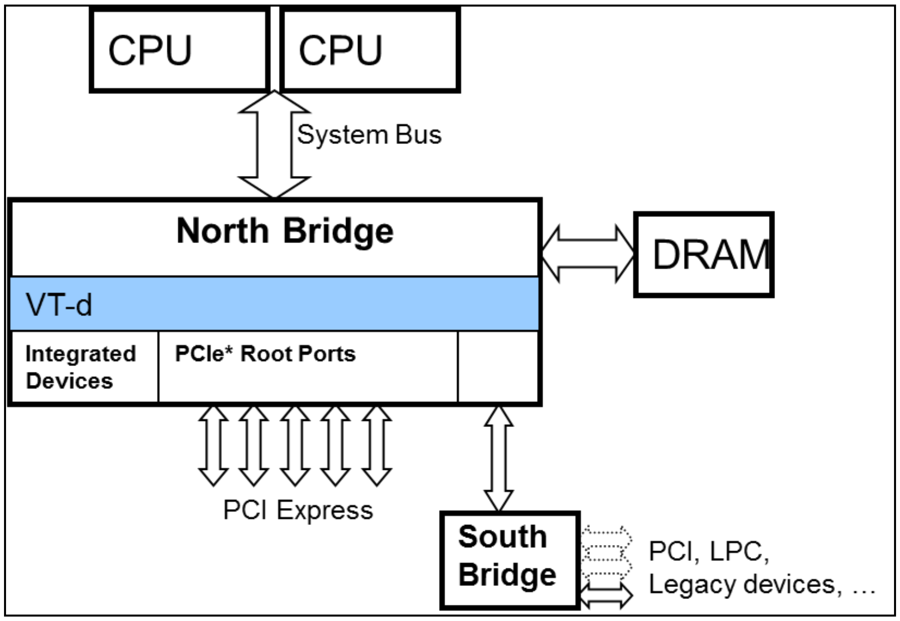

Figure 7: Intel VT-d modifies the standard computer architecture [17].

VT-d also solves an issue related to guest-resident device drivers arising from the address translation provided by EPT. To understand the issue, first consider a typical DMA transfer carried out by a device driver in a kernel running directly on hardware (no virtualization) as shown in Figure 8. The device driver has a pointer to a buffer that it needs to send to the device. The pointer is a virtual address, but the device can only read/write to physical addresses. So the driver leverages the kernel's knowledge of the page tables, and calls a kernel function to translate the virtual address to a physical address. This physical address is then passed to the device, and the device reads/writes directly from physical memory without intervention from the kernel.

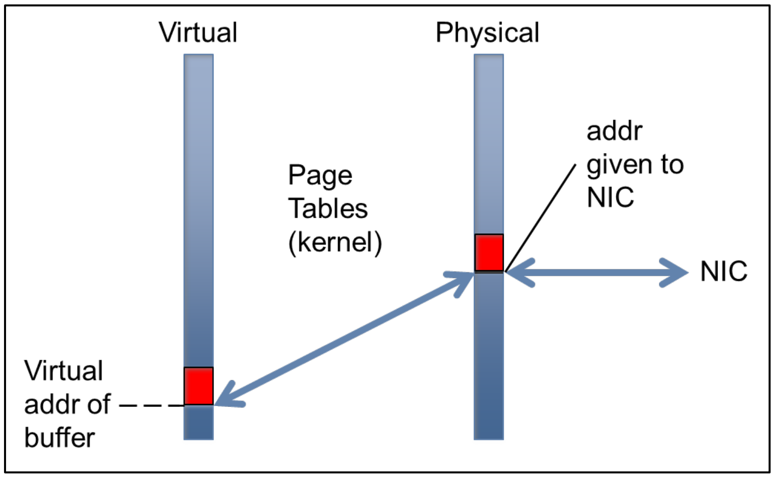

Figure 8: A typical DMA transfer on a system without virtualization.

EPT introduces the guest-physical address space, which allows the hypervisor to translate a guest-physical address to an arbitrary physical address. This causes devices and guest-resident drivers to have inconsistent views of memory as illustrated in Figure 9. The driver believes it has access to physical addresses, when in fact they are guest-physical addresses. Meanwhile, the device is unaware of any changes and uses guest-physical addresses provided by the driver to access the physical address space. This may lead to memory corruption via DMA.

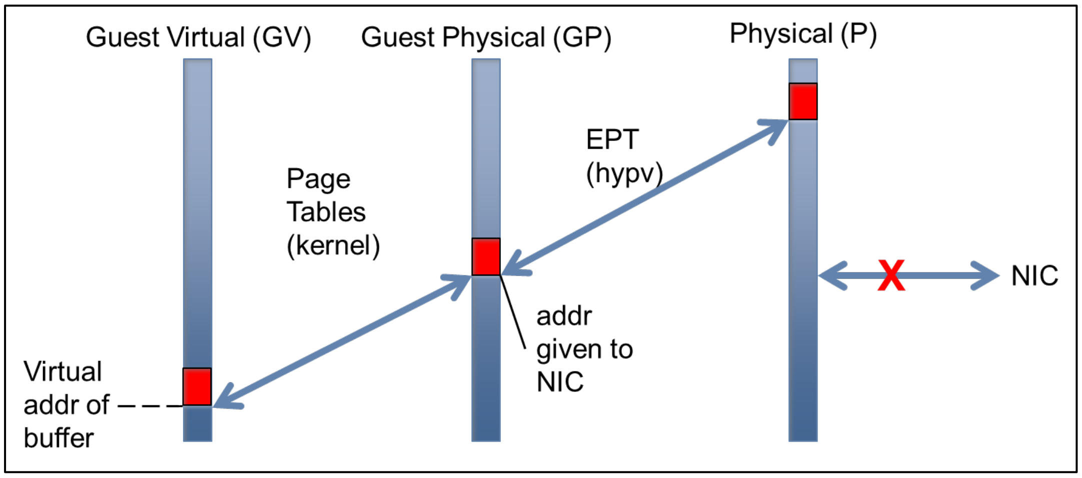

Figure 9: The NIC uses a driver-supplied GP address as a P address, corrupting memory.

VT-d provides the hardware necessary to solve this problem as illustrated in Figure 10. Essentially, VT-d provides a "device virtual" address space. The hypervisor can configure VT-d address translation to mirror EPT address translation, allowing devices to transparently use guest-physical addresses.

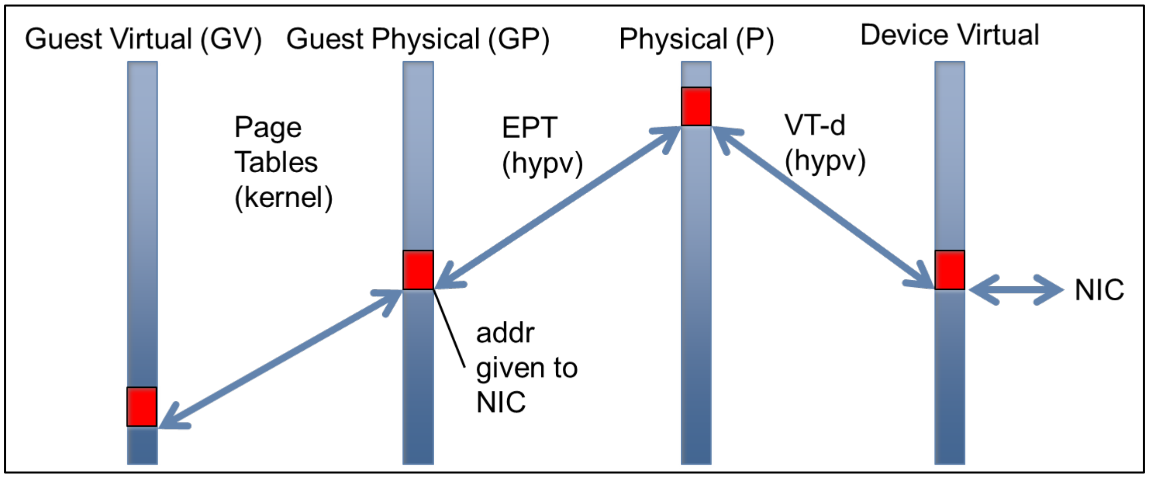

Figure 10: DMA transfer from guest-resident driver to NIC card with EPT and VT-d.

Unfortunately, the address translation features of VT-d are not present in our hardware and were not available at the time our servers were acquired. As a stopgap, we have implemented a workaround shown in Figure 11 that allows the same functionality without the full protection benefits. We leverage the kernel and the hypervisor to ensure that the driver passes on true physical addresses to the device. The kernel is able to translate a guest-physical address to a physical address using a call to the hypervisor (via the *vmcall* instruction). The kernel then provides the driver with a mapping from guest virtual addresses to physical addresses.

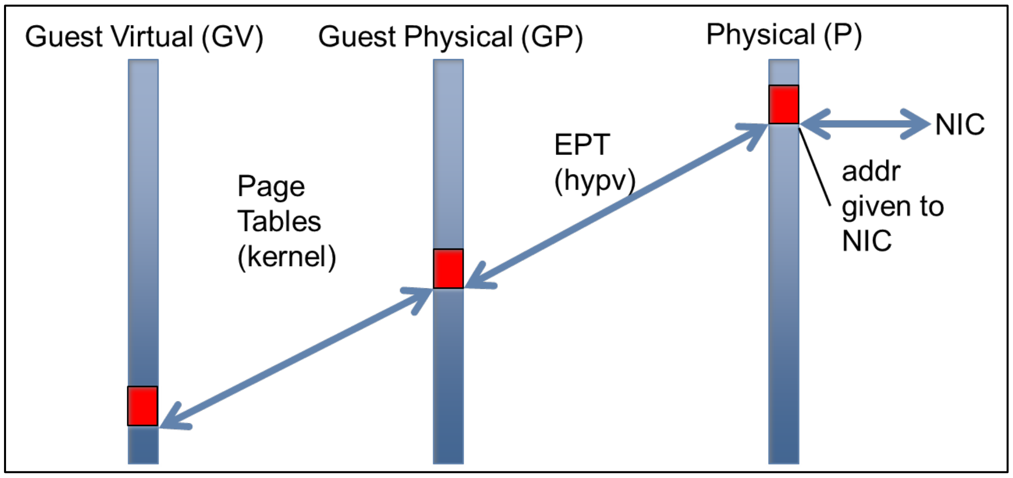

Figure 11: EPT Workaround for DMA– kernel requests P address from hypervisor.

In short, VT-d technology accomplishes two things: it allows a hypervisor to protect itself from devices, and it allows guests to transparently and safely control peripheral devices. Although our hardware does not have full support for VT-d, we have implemented a workaround that gives similar functionality. Although it does not have the hardware protection provided by EPT, it does allow a guest to control peripherals with almost no hypervisor interaction. Thus, we were able to remove device driver code from the hypervisor and place it at the less privileged user level.

Quantifying the Attack Surface

Recall that the number of potential vulnerabilities in a codebase is roughly proportional to the number of lines of code [3]; approximately 0.16 errors per thousand lines. The Bear kernel and hypervisor were designed to extensively share code in order to minimize the attack surface. As noted earlier, the two have considerable overlap in functionality. For example, memory management, PCI device auto-detection, and interrupt configuration must be performed at both levels. Accordingly, the Bear source code consists of self-contained modules that can be compiled and used in either the hypervisor or the kernel to provide these services. Furthermore, these code modules share generic implementations of well-known data structures, including a linked list and a hash table. These flexible implementations eliminate code redundancy.

To demonstrate the relative size of the Bear attack surface, we compare the number of lines of code (LOC) in the kernel and hypervisor with those of other state of the art systems. The lines of code were counted using the open-source code analyzer *cloc* using only C sources and assembly code. As a result, the Bear results are accurate while the other results represent a lower-bound. Collectively, the Bear hypervisor and micro-kernel combined offer *three orders of magnitude* less code than alternative solutions. This results into a small attack surface, aggressively applying the latest hardware protection mechanisms, with a small number of predicted vulnerabilities.

Kernel	Lines of Code
Bear Kernel	9,454 (7,399 shared code)
Linux Kernel	10,639,311
FreeBSD	3,707,252
MINIX 3.2.0	16,109

Table 2: Kernel Comparison: Lines of Code

Hypervisor	Lines of Code
Bear Hypervisor	8,701 (7,399 shared code)
Xen 4.1	262,191 (+ Dom0 kernel)
VMWare ESX	>150,000 (+ service terminal)

Table 3: Hypervisor Comparison: Lines of Code

The number of lines of code in the hypervisor and microkernel combined is 10,756, yielding an expected defect incidence of less than two errors for mature code. The corresponding attack surface for the micro-kernel executable image is 62.02Kbytes, the hypervisor is 54.78Kbytes, bringing the combined attack surface size to 116.8 Kbytes.

Despite the desire for secure systems, the reality is that no system will see practical use without acceptable performance. To establish a baseline, the Bear system was benchmarked against Ubuntu Linux using the standard AIM9 benchmarking suite to determine:

- How the un-optimized Bear system compares in performance with a highly optimized standard system.
- What impact the Bear Hypervisor has on performance, reflecting the core cost of resilience.

Obviously, our presumption is that Bear will be slower: the Linux kernel was released in 1991 and has been under continuous improvement and optimization ever since. In contrast, Bear is a research prototype developed primarily to explore resilience over the last two years. In addition, Bear uses a simple file system and a simple, slow disk driver as a stopgap measure until a more suitable read-only file store is integrated. Thus, any benchmark involving file operations are dominated by the disk driver's (lack of) performance.

The AIM9 suite is summarized in Table 3 and consists of 5 benchmark categories; the second column indicates the status of the port to Bear. The core benchmark routines employed are:

- **Add:** Conducts 4000 iterations, where each iteration consists of 1 million additions on short values, 1 million additions on integer values, and 1 million additions on long values. A total of approximately 12 Billion addition operations are executed. The **Mul** and **Div** benchmarks are functionally the same as Add but use multiplication and division.

- **Fork:** Conducts 1000 iterations, where each iteration performs one fork operation followed by exiting the child process.

- **Exec:** Conducts 1000 iterations, where each iteration performs one fork, followed by the execution of a small binary that returns 1. ***Note that this operation uses the file system.***

Benchmark Routine	Status	Bear Relevance
Integer Operations: Add / Mul / Div	✔	Context switching, Caching, Scheduling
Syscalls: Fork / Exec	✔	System Calls, Message Passing System
File System	✖	Awaiting NFS implementation
Network Stack	✖	Not yet ported
Numerical Operations	✖	Not yet ported

Table 4: The AIM9 Benchmark Suite

Bear was set to context switch every 10msec by default; therefore the Add benchmark, for example, involves approximately 15,000 context switches. Although this number is small compared to the number of addition operations, the benchmark would highlight adverse performance in memory management or interrupt handling. In contrast, Linux includes optimizations not present in Bear that allow processes to run for longer time slices based on the system state.

Routine	Ubuntu 12.04	Bear w/o Hypervisor	Bear w/ Hypervisor
Add	144 sec	150 sec	151 sec
Mul	250 sec	263 sec	261 sec
Div	1335 sec	1807 sec	1812 sec
Fork	0.3 sec	3.2 sec	3.2 sec
Exec	0.3 sec	3.2 sec	3.2 sec

Table 5: Performance Study Results

Our primary conclusion is that the additional overhead created by the hypervisor, our source of kernel resilience, is negligible. All 5 tests show that enabling the hypervisor does not lead to a significant performance loss – either in time or CPU cycles. Furthermore, Ubuntu 12.04 running on a Linux kernel 2.6.38-15-generic is *only* 5% faster than Bear on the core Add and Mul benchmarks. As expected, the optimizations present in Ubuntu result in better performance on other benchmarks. However, there exist simple optimizations to e.g., Bear's admittedly naïve implementations of fork and exec, that could substantially increase performance with little effect on attack surface.

Extending Trust to Distributed Applications

Modern computing is no longer centered on the single-machine, single-program paradigm. Falling under the broad moniker of cloud computing, an array new products and services now take advantage of the trend toward multi-core architectures and the low cost of COTS hardware to provide superior performance, efficiency, or convenience. However, this implies that large numbers of hardware nodes, the accompanying software stacks, and the communication paths between them are now all critical points of failure *and* vectors for attack. Moreover, the presence of a common operating system on every machine has the effect of amplifying vulnerabilities across the cloud. To circumvent this vulnerability amplification, we are investigating methods to add diversity through both source-to-source transformation and run-time translations. These changes will generate a large-number (>1 million) of semantically equivalent gold-standard images for the same small operating system source code. As a result, we would expect every instance of the Bear system to be unique and continually changing over time, mitigating the ability to use static code analysis to determine vulnerabilities. The core mechanism to discard the micro-kernel and re-establish trust allows the new instance to be a completely different runtime image.

The minimalist MPI-like message passing system, rMP, described previously is sufficient to provide system calls and inter-process communication. In addition, it has been used to express all three of the prevalent concurrent problem solving strategies that utilize *functional*, *domain*, and *irregular* decompositions. For the purpose of experimental evaluations, we have developed a suite of message-passing applications that exemplify these strategies involving numerical integration, iterative solution of partial differential equations, and a non-trivial LiDAR processing algorithm respectively [18]. However, the API also provides the necessary and sufficient functionality to implement process *replication* and *mobility* in the cloud. These capabilities are in turn sufficient to build distributed forms of *resource management* and *resilience*: processes are *replicated* and *dynamically regenerated* to assure that an application may proceed in the presence of malicious code or failures. The net impact of this approach is to allow the level of system assurance to be maintained, ensuring that a military mission may continue unabated [19].

Figure 12 illustrates how these ideas operate. At the user level, a concurrent application is expressed as processes that cooperate through *msgsend* and *msgrecv* message-passing primitives described earlier. A middle-ware layer implements a resilient view that replicates each process and organizes communication between the resulting *process groups*. Point-to-point communication among user processes is implemented by multi-cast communication between process groups. Individual processes within each group are mapped to different computers to ensure that a single attack or failure cannot impact an entire group.

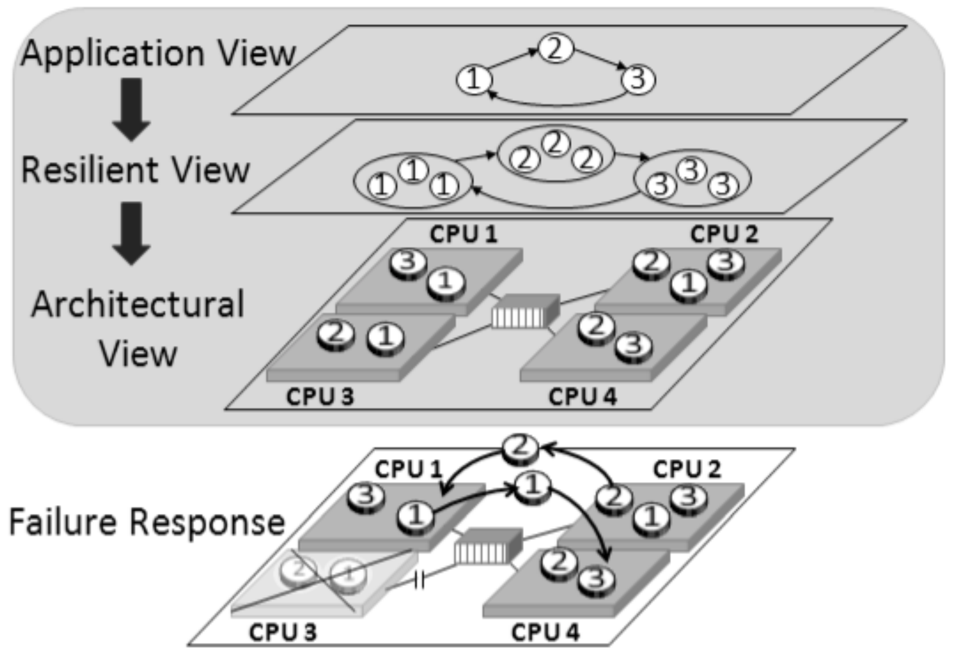

Figure 12: Dynamic process regeneration

The base of Figure 3 shows how the process structure responds to an attack or failure: An attack perpetrated against processor 3 causes processes 1 and 2 to fail or to portray inconsistencies in behavior or communication when compared to other replicas within their respective groups. These inconsistencies are detected either by behavioral alerts, communication timeouts, message comparison, or from external sources (e.g. SIGINT, Humint). Inconsistencies trigger automatic process regeneration: the consistent copies of processes 1 and 2 are used to dynamically regenerate a new replica and migrate it to alternate processors 4 and 1, respectively. As a result, the process structure is reconstituted, and the application continues operation with the same level of assurance.

The transparent realization of resilience on large-scale concurrent architectures necessitates an automatic approach to process scheduling. Our early work in this area resulted in a general algorithm for load balancing based on the *heat diffusion equation* [20] that can be implemented using the message passing API. This approach has several attractive properties: it uses a simple, fast, scalable algorithm involving only nearest neighbor communication; additionally, global progress and convergence are guaranteed through well-established mathematical analysis. The algorithm has been shown, through simulation, to simultaneously balance multiple independent load distributions over large-scale architectures, even with huge random load injections. Vector based extensions to the algorithm allow multiple resources (including communication, memory, and CPU load) to be balanced simultaneously [21].

Conclusion

Military systems have gained tremendously from the cost and flexibility benefits afforded by widespread adoption of commercial off the shelf (COTS) technology -- to the point where it is now difficult to imagine how we might operate, with similar levels of efficiency, using non-COTS methods. However, in times of tension, critical mission capabilities *must* continue to operate, even if major components of "the network" are unavailable and the systems upon which we rely are repeatedly compromised by error, fault, or malicious actions. It therefore behooves us to apply Occam's razor to pare back the layers of complexity that have been thrust upon us by commercial vendors, in light of the controlled environment in which DoD operates, to improve *resilience* and *increase attacker workload*.

Our approach is to use COTS subsystems, accepting their imperfections, but augmenting them with ideas from the fault-tolerance, distributed computing, and encryption communities. The research described in this paper explores how we might pursue this goal using three basic precepts:

- Don't trust what you have -- *validate*, *replicate and regenerate*,
- Don't advertise what you do – *hide and camouflage*, and
- Don't be predictable – instead be *mobile* and *non-deterministic*.

The Bear system uses overlapping regenerative techniques, combined at every layer of the system, from the user to the hardware. These methods deny surveillance by continually invalidating surveillance data, hiding in the network, and using camouflage. Persistence is denied by non-deterministically replacing, refreshing, replicating, and/or relocating components so as to continually re-establish trust. The methods can be incorporated individually, as independent modes through loadable modules, or collectively and continuously for critical missions.

The desire for field-upgradable hardware has opened a new dimension to malicious code in firmware and/or flash [22]. In consequence, there are some simple additions to COTS systems that are particularly valuable for improving resilience described here. These include *read-only memory* from which to draw *encrypted gold-standard images* that represent the code of final recourse, *removable links* on the primary write-lines to core flash components, and/or an *out-of-band network channel with associated micro-controller* to control flash updates and/or provide a forensic interface. The latter facility can be used to repeatedly re-flash devices when they are not in use or at designated system refresh times.

References

[1] W. A. Arbaugh, D. J. Farber, and J. M. Smith. "A secure and reliable bootstrap architecture." In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (SP '97). IEEE Computer Society, Washington, DC, USA, 65-. 1997.

[2] B. Blunden. *The Rootkit Arsenal: Escape and Evation in the Dark Corners of the System*. USA: Jones and Bartlett Publishers, Inc. 2009.

[3] Pandey and Tiwari, "Reliability Issues in Open Source Software." International Journal of Computer Applications, vol. 34 issue 1, pp. 34-38. 2011.

[4] Corbató, Fernando J., and Victor A. Vyssotsky. "Introduction and overview of the Multics system." Proceedings of the November 30--December 1, 1965, fall joint computer conference, part I. ACM, 1965.

[5] Tanenbaum and Woodhull, "Operating Systems: Design and Implementation," Prentice Hall, 2006.

[6] The MPI Forum. "MPI: A Message Passing Interface, version 2.2." Knoxville, TN: University of Tennessee. 2009.

[7] VMware, E. S. X. "Server: User's Manual." Version 1 (2011): 122-124.

[8] Matthews, Jeanna N., et al. Running Xen: a hands-on guide to the art of virtualization. Prentice Hall PTR, 2008.

[9] Habib, Irfan. "Virtualization with kvm." Linux Journal 2008.166 (2008): 8.

[10] c0ntex. "Bypassing Non-Executable-Stack During Exploitation with Return-to-libc." Open Security Group. 15 Nov. 2012 http://www.open-security.org/texts/4

[11] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti and E. Kirda, "G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries," in Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC), New York, 2010.

[12] Chou, Andy, et al. An empirical study of operating systems errors. Vol. 35. No. 5. ACM, 2001.

[13] Kuhn, Stephen, and Taylor, Stephen. "A Survey of Forensic Analysis in Virtualized Environments." Tech. rep., Dartmouth College, Hanover, New Hampshire, 2011.

[14] D Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic Approaches to Thwart Adversary Intelligence Gathering." DARPA Information Survivability Conference and Exposition, vol. 1 pp. 176-185, 2001.

[15] S. Kuhn and S. Taylor, “Increasing Attacker Workload with Virtual Machines”, submitted to MILCOM 2011. (Available as Thayer Technical Report TR11-002 at http://thayer.dartmouth.edu/tr/reports).

[16] M. Kanter and S. Taylor, Camouflaging Servers to Avoid Exploits, submitted to MILCOM 2011. (Available as Thayer Technical Report TR11-001 at http://thayer.dartmouth.edu/tr/reports)

[17] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1.” August 2012 edition.

[18] C. Nichols, S. Taylor, J. Keranen, G. Schultz. “A Concurrent Algorithm for Real-Time Tactical LIDAR.” *IEEE Aerospace Conference Proceedings*, 2011.

[19] K. McGill and S. Taylor, “Operating System Support for Resilience,” Submitted to IEEE Transactions on Reliability, (Available as Thayer Technical Report TR11-003 at http://thayer.dartmouth.edu/tr/reports).

[20] A. Heirich and S. Taylor "Load Balancing by Diffusion", Proceedings of 24th International Conference on Parallel Programming, vol 3 CRC Press pp 192-202, 1995. Outstanding Paper Award.

[21] J. Watts, and S. Taylor, "A Vector-based Strategy for Dynamic Resource Allocation", Journal of Concurrency: Practice and Experiences, 1998.

[22] Dell, Inc. “PowerEdge R410 replacement motherboard contains malware,” http://en.community.dell.com/support-forums/servers/f/956/t/19339458.aspx, Jul 10 2010.