Denali: A Scalable Isolation Kernel

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble
The University of Washington
{andrew,mar,gribble}@cs.washington.edu

Abstract

The Denali project provides system support for running several mutually distrusting Internet services on the same physical infrastructure. For example, this would enable a developer to push dynamic content into third party hosting infrastructure such as content distribution networks. To accomplish this, we propose a new kernel architecture called an isolation kernel *to isolate untrusted applications. An isolation kernel is a simple, thin software layer that runs directly on hardware (and hence below operating systems), whose function is to subdivide a physical machine into a set of fully isolated protection domains. Isolation kernels resemble virtual machine monitors in that they expose a virtualized hardware interface to a set of virtual machines. Unlike VMMs, however, isolation kernels do not attempt to precisely emulate the underlying physical architecture. By selectively modifying the hardware architecture, we enable our system to scale up to 1000's of virtual machines on commodity hardware. In this paper, we describe a set of design principles that govern isolation kernels, briefly discuss a prototype isolation kernel, and present future work and applications of isolation kernels.*

1. Introduction

The Internet is dramatically changing the way we think about application deployment. Instead of installing shrink-wrapped software on their PCs, users are increasingly relying on infrastructural services such as Hotmail and MapQuest. This "Internet services" model offers several powerful advantages: software distribution is simplified, upgrades and bug-fixes can be applied immediately, and services are always on and accessible from any capable device. The attractiveness of this model has resulted in significant industrial and research attention into Internet service frameworks, including .NET.

Although Internet services have compelling advantages, the introduction of a new service currently requires a large investment in infrastructure and administration. We believe this high barrier to entry stifles innovation, especially when the service must run in many locations across the wide-area (as would be necessary to enable dynamically generated content in content delivery networks, for example). For the Internet services model to succeed, we believe that the ownership and management of physical Internet service infrastructure is best handled by third party providers such as ISPs, and that mechanisms should be established to allow Internet service authors to "push" new services into this infrastructure in a safe manner.

Because not all services warrant their own dedicated hardware, services will need to be multiplexed on the same physical machines, as is currently done for virtual web site management. Unfortunately, Internet services contain active code rather than static data, which raises serious trust and security issues: infrastructure providers cannot trust hosted Internet services, and services will not trust each other. Accordingly, there is a need to provide strong isolation between services, both to enforce security and to control services' resource consumption.

In this paper, we propose a software construct called an *isolation kernel*, whose purpose is to multiplex physical hardware across many mutually untrusting Internet services by containing each service within an isolation domain. Although there have been attempts to address security and performance isolation within a monolithic OS [2, 10], we believe these issues are best addressed *under* a monolithic OS. An isolation kernel is a thin software layer which virtualizes the underlying hardware, in much the same fashion as a virtual machine monitor (VMM). Unlike a VMM, an isolation kernel does not attempt to precisely emulate the underlying physical architecture, because there are significant simplicity, scalability, and performance benefits to be gained by modifying it, as we will argue later in this paper.

An isolation kernel is similar in many respects to other small-kernel architectures, such as virtual machine monitors [12], hypervisors [4], microkernels [1], and exokernels [8]. In the next section of this paper, we outline some of the guiding design principles of isolation kernels, and describe how our work differs from other small-kernel architectures. Next, we describe the architecture and implementation of Denali, our prototype isolation kernel for the x86 architecture. Finally, we sketch out our future directions for our research.

2. Design Principles and Choices

In this section of the paper, we present a number of principles and design choices that guided us while architecting the Denali isolation kernel. These principles were motivated by technological trends, as well as characteristics of

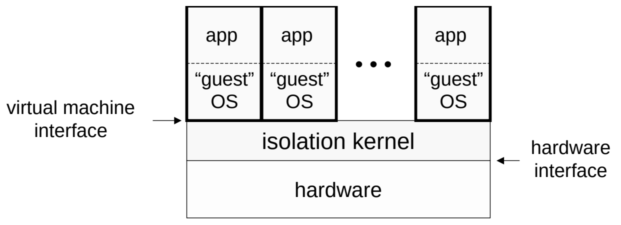

Figure 1. An isolation kernel. An isolation kernel is a thin software layer which exposes a virtual machine abstraction, and which prevents direct sharing across VMs.

the applications that we wish to support.

Simplicity promotes security: A long history of vulnerabilities suggests that conventional operating systems are poorly suited to isolating and containing untrusted code. We argue that this is a fundamental consequence of two architectural characteristics of OSs, rather than simply being a matter of avoidable implementation flaws.

First, an OS exposes high-level abstractions, rather than low-level resources, and enforces protection at the same layer as the exposed high-level abstractions. For example, files serve as an abstract container for persistent data, as well as a unit of access control. The high level at which protection is enforced gives rise to "layer-below attacks" [13], in which an attacker accesses a resource below the layer of abstraction. For example, forcing a core-dump of an application may allow an attacker to bypass virtual memory protection to inspect program state. The complexity of OSs, which is largely due to the complexity of implementing high-level abstractions, makes it extremely difficult to anticipate and protect against layer-below attacks.

Second, a modern OS typically exposes a wide API. The original version of UNIX had only 33 system calls; today, Windows has over 3400 system calls [17]. This wide API runs contrary to the principle of economy of mechanism [16], which states that security is achieved by limiting the number of access paths available to untrusted code. Monolithic OSs are constantly evolving: each new release contains millions of lines of new code and countless new features. This leads to insecurity since new code is known to contain more bugs than older, more stable code [6].

We believe than an isolation kernel must be implemented as a thin software layer that exposes a narrow API and runs directly on hardware; this is similar to small-kernel architectures such as virtual machine monitors or microkernels. Rather than providing complex high-level abstractions, an isolation kernel exposes and protects low-level hardware resources. This greatly simplifies the implementation of the isolation kernel, promoting security and avoiding layer-below attacks. Because an isolation kernel defers the implementation of abstractions to higher software layers (Figure 1), pressure to accumulate new features and widen its exposed interface is alleviated.

Performance is no longer the primary issue: In the past, small-kernel architectures have been considered inefficient when compared to monolithic kernels. However, technology advances have made raw performance less of an issue, and in many cases it has become reasonable to trade away performance in return for reliability, security, or manageability.

Additionally, recent results have demonstrated that the performance penalty of small kernels is not prohibitive. The Disco virtual machine monitor reported performance penalties of no more than 16% over a range of applications [5]. Our early experience with Denali is even more encouraging, as described in Section 3.

Sharing is infrequent: Operating systems have customarily provided efficient mechanisms for sharing data between applications, such as fast IPC and shared file systems. In our application domain, however, we expect sharing to be infrequent, since Internet services are designed and operated by independent users. This suggests that we can afford to have a high cost of sharing across isolation domains, if in return we strengthen isolation.

The emphasis of conventional OSs on providing data sharing mechanisms has a harmful effect on their ability to provide isolation. For example, in most OSs, all applications see the same global file system name space regardless of whether they want to share data. If the file system access control policy is not perfectly configured, then malicious applications can find hard-to-spot ways of reading or modifying other users' data. Attackers have used symbolic links to trick unsuspecting applications into reading or modifying privileged system files [1]. Although this particular vulnerability affects conventional OSs, we believe that small-kernel architectures that encourage fine-grained data sharing would be susceptible to the same class of vulnerabilities. This includes microkernel systems, which support sharing through user-mode file servers [1], and Exokernel systems which support sharing by downloading protection policy into the kernel [8].

Virtual machine monitors like Disco [5] and VM/370 are better suited to isolation because they disallow direct sharing. Each virtual machine on a VMM is confined to a private, virtual namespace: virtual physical memory pages, virtual disk blocks, and so forth. The default way to share data on such a system is to send the data over a (virtual) Ethernet segment. Thus, applications that desire complete isolation can simply ignore all network traffic, or use firewall techniques to filter out suspicious traffic.

Zipf's law implies a need to scale: Measurement studies of web documents, web servers, DNS names, and other network services show that popularity distributions tend to be Zipfian [3]. Based on this, we expect that the popularity distribution for Internet services will also be driven by Zipf's law.

Zipfian distributions are heavy-tailed, meaning that a

[1]Refer to CERT vulnerability notes: VU#356323, VU#747736, and VU#426273.

non-trivial fraction of requests go to a large set of unpopular services. Individually, these services are accessed infrequently, motivating the desire to multiplex many of them on a single computer for reasons of affordability and manageability. However, the unpopular services collectively constitute a large fraction of the total requests. Previous studies of web cache performance have demonstrated that unpopular objects tend to drag down overall system performance [3].

Our goal in Denali is to support both popular and unpopular services. To support the unpopular, we want to be able to host a large number of services (hundreds, if not thousands) on a commodity PC. To make this feasible, we must use main memory as a cache of active services, using disk as backing store for the majority of services that are idle. Thus, our system must support rapid swapping of services off disk to mitigate the negative results in [3].

Transparency is a non-goal, and is potentially harmful: One consideration for small-kernel architectures is how to support legacy software, including OS code. Virtual machine monitors are distinct in that they can provide transparent backwards compatibility for legacy operating systems [5, 12] by precisely emulating the underlying physical architecture. Although transparency is useful for supporting legacy software, it is independent from the issue of enforcing isolation, and we believe there are compelling reasons to allow the interface exposed by an isolation kernel to differ from the physical architecture on which the kernel runs.

Some physical architectures are not strictly virtualizable [11]. Non-virtualizable architectures (such as x86) can be virtualized using binary re-writing techniques, however this requires significant complexity to handle a small set of infrequently used instructions [14]. Similarly, some components of the hardware or firmware are rarely used by applications, but must be emulated to maintain backwards compatibility with legacy OSs. Examples of this include the x86 segmentation hardware and the BIOS.

The Denali isolation kernel exposes an interface that is similar to the underlying hardware architecture, but with several strategic modifications. In the next section, we describe our architecture modifications, which we use to enhance scalability and performance, while simplifying the implementation of the isolation kernel and the services that run on it.

Of course, giving up backwards compatibility carries the disadvantage that we must modify legacy OS code to run on our architecture. On the other hand, breaking backwards compatibility has provided us with the opportunity to redesign the guest operating system to be virtualization-aware. In the following section, we describe how we have used this ability inside Denali.

3. Denali: A Scalable Isolation Kernel

The Denali kernel aims to provide strong isolation and high scalability for Internet services. To achieve isolation, the kernel exposes a virtualized hardware interface — virtual I/O devices, virtual physical memory, etc. — to a set of virtual machines. Each virtual machine (VM) contains a guest OS, which contains a customary set of OS abstractions (such as TCP/IP sockets and threads), as well as one or more applications. In this respect, the Denali kernel resembles a virtual machine monitor.

Unlike VMMs, however, we have made selective modifications to the underlying physical architecture to promote scalability, performance and simplicity of implementation.

3.1. Architectural Modifications

One barrier to running a large number of concurrently active virtual machines is idle loops inside guest OSs. To prevent wasting CPU resources, Denali exposes an idle instruction, which allows a guest OS to relinquish control of the CPU[2]. The VM idles until an external interrupt arrives that requires processing. To prevent a VM from idling forever, Denali exposes a virtual alarm clock, which is set by the guest OS to raise an interrupt after a given number of clock ticks.

Another barrier to scalability relates to the handling of timers. On most systems, the passage of time is indicated by a programmable interval timer, which generates an interrupt every few milliseconds. Emulating this behavior in an isolation kernel would require raising a virtual timer interrupt on each timer tick for each virtual machine, resulting in a large number of user/kernel crossings. Moreover, each clock tick would wake up idle virtual machines, thereby preventing unpopular services from being swapped out to disk.

Denali's approach is to only raise timer interrupts to the running virtual machine. For all other VMs, the kernel exposes a global clock that advances with the hardware clock. After a VM has been context switched, the kernel raises a "wakeup" interrupt to indicate that the guest OS should recalibrate timing-related routines against the global clock.

Denali's interrupt mechanism was designed to better support many concurrently executing VMs. As the number of VMs increases, it becomes likely that multiple virtual interrupts will arrive for a given VM while it is context-switched out. Denali delivers all pending interrupts to a virtual machine in a single batch rather than enforcing a strict serial ordering. Batching reduces the number of user/kernel crossings, and allows the system to piggyback "informational" interrupts on normal hardware interrupts.

The Denali architecture does not expose virtual memory management hardware; instead, virtual machines reside in a single, flat address space, much like conventional OS processes. As a result, guest OSs are directly linked against applications, similar to Exokernel library operating

[2]The Denali idle instruction is similar to the x86 hlt instruction, which places the processor in a halted state. However, the hlt instruction does not have a time limit, and therefore is only used after the processor has been idle for some period of time (usually hundreds of milliseconds). The Denali alarm clock mechanism allows for CPU sharing at a much finer granularity.

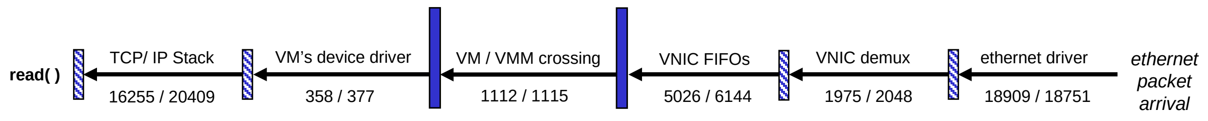

Figure 2. Packet processing overhead: This timeline illustrates the cost (in cycles) of packet reception, broken down across various functional stages. Each pair of numbers represents the number of cycles executed in that stage for 100 byte and 1400 byte packets, respectively. VNIC refers to the virtual NIC implementation.

systems [8]. We believe this simplification is justified, because a complex service that requires multiple protection domains can be structured to run in multiple virtual machines. Moreover, omitting virtual memory from the architecture improves performance by reducing TLB misses during guest OS context switches; this overhead was found to be significant during a recent measurement of VMWare's workstation product [18].

Denali exposes virtual I/O devices to virtual machines, but the interfaces to these devices have been drastically simplified relative to real hardware devices. For example, the virtual Ethernet device supports two operations: packet send and packet receive. The interface to real hardware devices is often more complex than necessary, which can lead to reduced performance during virtualization [18].

The Denali architecture greatly simplifies virtual machine initialization. There is no BIOS exposed, and all Denali virtual devices "power on" in a well-known boot state, eliminating the need for a guest OS to initialize devices. These changes dramatically reduce the complexity of both our isolation kernel (since it doesn't need to virtualize these architectural features), as well as the guest OSs themselves.

3.2. Implementation and Results

We have developed a prototype isolation kernel that runs directly on x86 hardware. Our implementation borrows device drivers and low-level support from the Flux OSKit [9]. However, the "core" of the kernel (scheduling, virtual device emulation, and paging) is entirely new. We have also constructed a prototype guest OS that contains a port of the BSD TCP/IP stack [7], full threading support, and a subset of the Posix API. We are currently investigating appropriate stable storage abstractions for our guest OS.

Our initial evaluation (on a 1.5 GHz Pentium IV uniprocessor with 1 GB RAM) has focused on networking applications, and has been very encouraging. A breakdown of the overhead associated with each network packet indicates that enforcing isolation through virtualization accounts for a relatively small fraction of the processing overhead (Figure 2). The physical device driver and guest OS TCP/IP stack represent the largest portion of per-packet overheads; for small and large packets, the physical device driver represents 43.3% and 38.4%, respectively, of the total receive overhead, while traversing of the TCP/IP stack accounted for an additional 37.3% and 41.8%, respectively.

Our current prototype can support an almost arbitrary

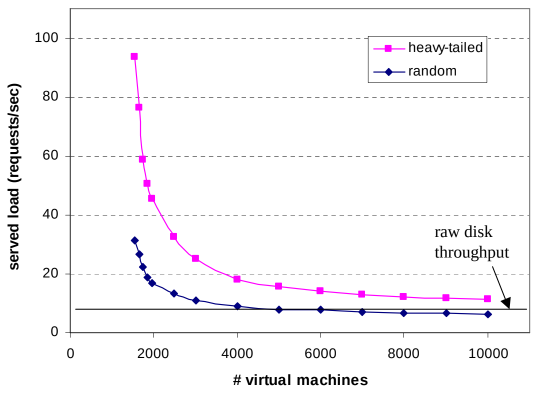

Figure 3. Web server throughput: This graph shows the aggregate serve load sustained across the set of web server VMs, each of which is delivering 2KB web pages. Denali utilizes the full disk swapping bandwidth for random requests. For heavy-tailed request distributions, Denali's performance surpasses raw disk performance by caching popular virtual machines in memory.

number of virtual machines by swapping inactive VMs to disk. Figure 3 demonstrates Denali's ability to scale up to 10,000 web server VMs. Requests were distributed across VMs according to two distributions: random and Zipfian (heavy-tailed) with $\alpha = .8$; all requests fetched a 2KB web document. For small numbers of VMs, Denali itself is not a performance bottleneck: the performance of the system is a function of the performance of the applications running on top of it. For example, Denali achieves an aggregate throughput of over 5,000 requests per second for up to 500 web server virtual machines (not shown on Figure 3). For large numbers of VMs, Denali's performance is disk-bound; our three disk subsystem can swap in 7 virtual machines per second. Denali's performance for random requests utilizes the full disk bandwidth; Denali's performance for heavy-tailed requests exceeds the raw disk bandwidth by caching popular virtual machines in memory.

4. Future Directions

Isolation kernels are relevant to a wide variety of application domains. In this section, we present avenues of future research for the Denali project. First, we discuss a set of mechanisms that could be added to the Denali isolation kernel to enhance performance or increase its functionality. We then outline several application domains to which isolation kernels can be applied.

4.1. Mechanisms

Performance isolation: Because an isolation kernel runs directly on the hardware and exposes (virtualized) hardware resources, it can precisely account for physical resources consumed by each VM. This fine-grained resource accounting should make it possible for an isolation kernel to enforce *performance isolation* between mutually distrusting applications, as well as security isolation.

Transparent sharing of resources: Isolation kernels achieve high aggregate system performance for large numbers of VMs by keeping popular services in memory while swapping unpopular services to disk. Leveraging copy-on-write techniques to transparently share code pages between different VMs may reduce the memory footprint of the popular services by eliminating redundant physical code pages, improving overall system performance by making it possible to keep a larger number of VMs in memory at a time. This technique should prove especially effective when many services use the same guest OS.

Checkpointing/cloning/migration: Because an isolation kernel is essentially an interposition layer between a virtual machine and the physical resources it consumes, an isolation kernel can observe and collect the full state of a VM. This state consists of the VM's memory footprint and virtual device state. Additionally, because virtual devices on two different physical machines will have the same interface, even if the underlying physical devices are different, a VM can potentially be migrated across heterogeneous physical machines. By leveraging these properties, an isolation kernel can provide checkpointing, cloning, and migration capabilities.

4.2. New Application Domains

Virtual clusters: Isolation kernels introduce the possibility of subdividing a physical cluster into several multiplexed virtual clusters, and dynamically growing or shrinking the amount of physical resources given to each virtual cluster. In this model, multiple virtual machines execute in a virtual cluster; each virtual cluster is mapped onto some number of nodes in the physical cluster. VMs from many virtual clusters can be multiplexed on a single physical machine to enable high resource utilization in the face of bursty request streams.

Virtual clusters inherit the scalability and availability properties of traditional clusters while acquiring new capabilities. As a virtual cluster's load increases, an isolation kernel can migrate VMs within that virtual cluster to physical machines with idle resources. Additionally, virtual clusters can achieve high availability in the face of physical node failures. Through checkpointing, VMs running on a failed node can be quickly restarted on a different physical node, restoring the membership of the virtual cluster to its original state even though the physical cluster's membership has changed.

Wide Area Infrastructures: Isolation kernels allow hosts to execute untrusted code, which should permit application authors to "push" new network services into third-party hosting infrastructure. For example, this should allow web service authors to push dynamic content generation code into caching and content delivery networks. A related application domain is wide-area network experimentation infrastructure (such as NIMI [15]); isolation kernels should allow multiple potentially untrustworthy experiments to be deployed and executed simultaneously on shared wide-area infrastructure.

Mobile devices: Mobile and wireless devices, such as Palm Pilots and iPAQs, are playing an increasingly important role in our computational infrastructure. Incorporating isolation kernels into these devices provides a mechanism for the safe download and execution of arbitrary code, permitting these mobile devices to acquire and run context-specific application code as they move between environments.

5 Conclusions

In this paper, we presented the design principles behind an *isolation kernel*, a simple, thin software layer that runs directly on hardware, whose function is to subdivide a physical machine into a set of fully isolated protection domains. We argued that several emerging applications would benefit from isolation kernels, as they either require the isolation of untrusted code, or the ability to precisely control and account for physical resources across competing software services. These emerging applications include pushing new Internet services into third party hosting infrastructure, deploying dynamic content generation code in content delivery networks, or pushing context-specific applications into mobile devices as they migrate across physical environments. We briefly described the design and implementation of the Denali isolation kernel, and presented experimental results that show that we can scale up to a very large number of simultaneous protection domains, as would be necessary for many of our targeted applications. Finally, we outlined several areas of future research.

References

[1] M. Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.

[2] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating system design and implementation*, Feb. 1999.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching, and Zipf-like distributions: Evidence, and implications, Mar 1999.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[5] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct. 1997.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, Oct. 2001.

[7] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March, 2001.

[8] D. Engler, M. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[9] B. Ford et al. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[10] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the sixth USENIX Security Symposium*, July 1996.

[11] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[12] R. Goldberg. A survey of virtual machine research. *IEEE computer magazine*, 7(6), June 1974.

[13] D. Gollmann. *Computer Security*. John Wiley and Son, Ltd., 1st edition, Feb. 1999.

[14] K. Lawton. Running multiple operating systems concurrently on an IA32 PC using virtualization techniques. `http://www.plex86.org/research/paper.txt`.

[15] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale internet measurement. *IEEE Communications Magazine*, 36(8):48–54, August 1998.

[16] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[17] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley and Sons, Aug. 2000.

[18] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual Usenix Technical Conference*, Boston, MA, USA, June 2001.