Dune: Safe User-level Access to Privileged CPU Features

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, Christos Kozyrakis
Stanford University

Abstract

Dune is a system that provides applications with direct but safe access to hardware features such as ring protection, page tables, and tagged TLBs, while preserving the existing OS interfaces for processes. Dune uses the virtualization hardware in modern processors to provide a *process*, rather than a *machine* abstraction. It consists of a small kernel module that initializes virtualization hardware and mediates interactions with the kernel, and a user-level library that helps applications manage privileged hardware features. We present the implementation of Dune for 64-bit x86 Linux. We use Dune to implement three user-level applications that can benefit from access to privileged hardware: a sandbox for untrusted code, a privilege separation facility, and a garbage collector. The use of Dune greatly simplifies the implementation of these applications and provides significant performance advantages.

1 Introduction

A wide variety of applications stand to benefit from access to "kernel-only" hardware features. As one example, Azul Systems demonstrates significant speedups to garbage collection through use of paging hardware [15, 36]. As another example, process migration, though implementable within a user program, can benefit considerably from access to page faults [40] and system calls [32]. In some cases, it might even be appropriate to replace the kernel entirely to meet the needs of a particular application. For example, IBOS improves browser security by moving browser abstractions into the lowest OS layers [35].

Such systems require changes in the kernel because hardware access in userspace is restricted for security and isolation reasons. Unfortunately, modifying the kernel is not ideal in practice because kernel changes can be fairly intrusive and, if done incorrectly, affect whole system stability. Moreover, if multiple applications require kernel changes, there is no guarantee that the changes will compose.

Another strategy is to bundle applications into virtual machine images with specialized kernels [4, 14]. Many modern CPUs contain virtualization hardware with which guest operating systems can safely and efficiently access kernel hardware features. Moreover, virtual machines provide failure containment similar to that of processes—*i.e.,* buggy or malicious behavior should not bring down the entire physical machine.

Unfortunately, virtual machines offer poor integration with the host operating system. Processes expect to inherit file descriptors from their parents, spawn other processes, share a file system and devices with their parents and children, and use IPC services such as Unix-domain sockets. Moving a process to a virtual machine for the purposes of, say, speeding up garbage collection is likely to break many assumptions and may simply not be worth the hassle. Moreover, producing a kernel for an application-specific virtual machine is no small task. Production kernels such as Linux are complex and hard to modify. Yet implementing a special-purpose kernel with a simple virtual memory layer is also challenging. In addition to virtual memory, one must support a file system, a networking stack, device drivers, and a bootstrap process.

This paper introduces a new approach to application use of kernel hardware features: using virtualization hardware to provide a *process*, rather than a *machine* abstraction. We have implemented this approach for Linux on 64-bit Intel CPUs in a system called *Dune*. Dune provides a loadable kernel module that works with unmodified Linux kernels. The module allows processes to enter "Dune mode," an irreversible transition in which, through virtualization hardware, safe and fast access to privileged hardware features is enabled, including privilege modes, virtual memory registers, page tables, and interrupt, exception, and system call vectors. We provide a user-level library, libDune, to facilitate the use of these features.

For applications that fit its paradigm, Dune offers several advantages over virtual machines. First, a Dune process is a normal Linux process, the only difference being that it uses the `VMCALL` instruction to invoke system calls. This means that Dune processes have full access to the rest of the system and are an integral part of it, and

that Dune applications are easy to develop (like application programming, not kernel programming). Second, because the Dune kernel module is not attempting to provide a machine abstraction, the module can be both simpler and faster. In particular, the virtualization hardware can be configured to avoid saving and restoring several pieces of hardware state that would be required for a virtual machine.

With Dune we contribute the following:

- We present a design that uses hardware-assisted virtualization to safely and efficiently expose privileged hardware features to user programs while preserving standard OS abstractions.
- We evaluate three hardware features in detail and show how they can benefit user programs: exceptions, paging, and privilege modes.
- We demonstrate the end-to-end utility of Dune by implementing and evaluating three use cases: sandboxing, privilege separation, and garbage collection.

2 Virtualization and Hardware

In this section, we review the hardware support for virtualization and discuss which privileged hardware features Dune is able to expose. Throughout the paper, we describe Dune in terms of x86 CPUs and Intel VT-x. However, this is not fundamental to our design, and in Section 7, we broaden our discussion to include other architectures that could be supported in the future.

2.1 The Intel VT-x Extension

In order to improve virtualization performance and simplify VMM implementation, Intel has developed VT-x [37], a virtualization extension to the x86 ISA. AMD also provides a similar extension with a different hardware interface called SVM [3].

The simplest method of adapting hardware to support virtualization is to introduce a mechanism for trapping each instruction that accesses privileged state so that emulation can be performed by a VMM. VT-x embraces a more sophisticated approach, inspired by IBM's interpretive execution architecture [31], where as many instructions as possible, including most that access privileged state, are executed directly in hardware without any intervention from the VMM. This is possible because hardware maintains a "shadow copy" of privileged state. The motivation for this approach is to increase performance, as traps can be a significant source of overhead.

VT-x adopts a design where the CPU is split into two operating modes: *VMX root* and *VMX non-root* mode. VMX root mode is generally used to run the VMM and does not change CPU behavior, except to enable access to new instructions for managing VT-x. VMX non-root mode, on the other hand, restricts CPU behavior and is intended for running virtualized guest OSes.

Transitions between VMX modes are managed by hardware. When the VMM executes the `VMLAUNCH` or `VMRESUME` instruction, hardware performs a *VM entry*; placing the CPU in VMX non-root mode and executing the guest. Then, when action is required from the VMM, hardware performs a *VM exit*, placing the CPU back in VMX root mode and jumping to a VMM entry point. Hardware automatically saves and restores most architectural state during both types of transitions. This is accomplished by using buffers in a memory resident data structure called the VM control structure (VMCS).

In addition to storing architectural state, the VMCS contains a myriad of configuration parameters that allow the VMM to control execution and specify which type of events should generate VM exits. This gives the VMM considerable flexibility in determining which hardware is exposed to the guest. For example, a VMM could configure the VMCS so that the `HLT` instruction causes a VM exit or it could allow the guest to halt the CPU. However, some hardware interfaces, such as the interrupt descriptor table (IDT) and privilege modes, are exposed implicitly in VMX non-root mode and never generate VM exits when accessed. Moreover, a guest can manually request a VM exit by using the `VMCALL` instruction.

Virtual memory is perhaps the most difficult hardware feature for a VMM to expose safely. A straw man solution would be to configure the VMCS so that the guest has access to the page table root register, `%CR3`. However, this would place complete trust in the guest because it would be possible for it to configure the page table to access any physical memory address, including memory that belongs to the VMM. Fortunately, VT-x includes a dedicated hardware mechanism, called the *extended page table* (EPT), that can enforce memory isolation on guests with direct access to virtual memory. It works by applying a second, underlying, layer of address translation that can only be configured by the VMM. AMD's SVM includes a similar mechanism to the EPT, referred to as a nested page table (NPT).

2.2 Supported Hardware Features

Dune uses VT-x to provide user programs with full access to x86 protection hardware. This includes three privileged hardware features: exceptions, virtual memory, and privilege modes. Table 1 shows the corresponding privileged

Mechanism	Privileged Instructions
Exceptions	LIDT, LTR, IRET, STI, CLI
Virtual Memory	MOV CRn, INVLPG, INVPCID
Privilege Modes	SYSRET, SYSEXIT, IRET
Segmentation	LGDT, LLDT

Table 1: Hardware features exposed by Dune and their corresponding privileged x86 instructions.

instructions made available for each feature. Dune also exposes segmentation, but we do not discuss it further, as it is primarily a legacy mechanism on modern x86 CPUs.

Efficient support for exceptions is important in a variety of use cases such as emulation, debugging, and performance tracing. Normally, reporting an exception to a user program requires privilege mode transitions and an upcall mechanism (*e.g.,* signals). Dune can reduce exception overhead because it uses VT-x to deliver exceptions directly in hardware. This does not, however, allow a Dune process to monopolize the CPU, as timer interrupts and other exceptions intended for the kernel will still cause a VM exit. The net result is that software overhead is eliminated and exception performance is determined by hardware efficiency alone. As just one example, Dune improves the speed of delivering page fault exceptions, when compared to SIGSEGV in Linux, by more than 4×. Several other types of exceptions are also accelerated, including breakpoints, floating point overflow and underflow, divide by zero, and invalid opcodes.

User programs can also benefit from fast and flexible access to virtual memory [5]. Use cases include checkpointing, garbage collection (evaluated in this paper), data-compression paging, and distributed shared memory. Dune improves virtual memory access by exposing page table entries to user programs directly, allowing them to control address translations, access permissions, global bits, and modified/accessed bits with simple memory references. In contrast, even the most efficient OS interfaces [17] add extra latency by requiring system calls in order to perform these operations. Letting applications write their own page tables does not affect security because the underlying EPT exposes only the normal process address space, which is equally accessible without Dune.

Dune also gives user programs the ability to manually control TLB invalidations. As a result, page table updates can be performed in batches when permitted by the application. This is considerably more challenging to support in the kernel because it is difficult to defer TLB invalidations when general correctness must be maintained. In addition, Dune exposes TLB tagging by providing access to Intel's recently added process-context identifier (PCID) feature. This permits a single user program to switch between multiple page tables efficiently. All together, we show that using Dune results in a 7× speedup over Linux in the Appel and Li user-level virtual memory benchmarks [5]. This figure includes the use of exception hardware to reduce page fault latency.

Finally, Dune exposes access to privilege modes. On x86, the most important privilege modes are ring 0 (supervisor mode) and ring 3 (user mode), although rings 1 and 2 are also available. Two motivating use cases for privilege modes are privilege separation and sandboxing of untrusted code, both evaluated in this paper. Dune can support privilege modes efficiently because VMX non-root mode maintains its own set of privilege rings. Hence, Dune allows hardware-enforced protection within a process in exactly the way kernels protect themselves from user processes. The supervisor bit in the page table is available to control memory isolation. Moreover, system call instructions trap to the process itself, rather than to the kernel, which can be used for system call interposition and to prevent untrusted code from directly accessing the kernel. Compared to *ptrace* in Linux, we show that Dune can intercept a system call with 25× less overhead.

Although the hardware features Dune exposes suffice in supporting our motivating use cases, several other hardware features, such as cache control, debug registers, and access to DMA-capable devices, could also be safely exposed through virtualization hardware. We leave these for future work and discuss their potential in Section 7.

3 Kernel Support for Dune

The core of Dune is a kernel module that manages VT-x and provides user programs with greater access to privileged hardware features. We describe this module here, including a system overview, a threat model, and a comparison to an ordinary VMM. We then explore three key aspects of the module's operation: managing memory, exposing access to privileged hardware, and preserving access to kernel interfaces. Finally, we describe the Dune module we implemented for the Linux kernel.

3.1 System Overview

Figure 1 shows a high-level view of the Dune architecture. Dune extends the kernel with a module that enables VT-x, placing the kernel in VMX root mode. Processes using Dune are granted direct but safe access to privileged hardware by running in VMX non-root mode. The Dune module intercepts VM exits, the only means for a Dune pro-

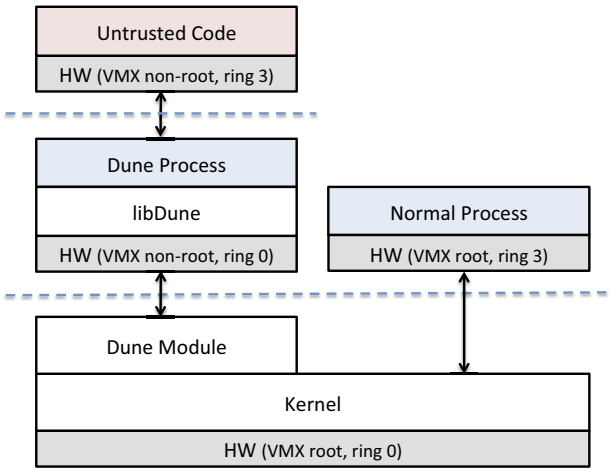

Figure 1: The Dune system architecture.

cess to access the kernel, and performs any necessary actions such as servicing a page fault, calling a system call, or yielding the CPU after a `HLT` instruction. Dune also includes a library, called *libDune*, to assist with managing privileged hardware features in userspace, discussed further in Section 4.

We apply Dune selectively to processes that need it; processes that do not use Dune are completely unaffected. A process can enable Dune at any point by initiating a transition through an *ioctl* on the `/dev/dune` device, but once in Dune mode, a process cannot exit Dune mode. Whenever a Dune process forks, the child process does not start in Dune mode, but can re-enter Dune if the use case requires it.

The Dune module requires VT-x. As a result, it cannot be used inside a VM unless there is support for nested VT-x [6]; the performance characteristics of such a configuration are an interesting topic of future consideration. On the other hand, it is possible to run a VMM on the same machine as the Dune module, even if the VMM requires VT-x, because VT-x can be controlled independently on each core.

3.2 Threat Model

Dune exposes privileged CPU features without affecting the existing security model of the underlying OS. Any external effects produced by a Dune-enabled process could be produced without Dune through the same series of system calls. However, by exposing hardware privilege modes, Dune enables additional privilege-separation techniques within a process that would not otherwise be practical.

We assume that the CPU is free of defects, although we acknowledge that in rare cases exploitable hardware flaws have been identified [26, 27].

3.3 Comparing to a VMM

Though all software using VT-x shares a common structure, Dune's use of VT-x deviates from that of standard VMMs. Specifically, Dune exposes a process environment instead of a machine environment. As a result, Dune is not capable of supporting a normal guest OS, but this permits Dune to be lighter weight and more flexible. Some of the most significant differences are as follows:

- Hypercalls are a common way for VMMs to support paravirtualization, a technique in which the guest OS is modified to use interfaces that are more efficient and less difficult to virtualize. In Dune, by contrast, the hypercall mechanism invokes normal Linux system calls. For example, a VMM might provide a hypercall to register an interrupt handler for a virtual network device, whereas a Dune process would use a hypercall to call *read* on a TCP socket.

- Many VMMs emulate physical hardware interfaces in order to support unmodified guest OSes. In Dune, only hardware features that can be directly accessed without VMM intervention are made available; in cases where this is not possible, a Dune process falls back on the OS. For example, most VMMs go to great lengths to present a virtual graphics card interface in order to support a frame buffer. By contrast, Dune processes employ the normal OS display service, usually an X server accessed over a Unix-domain socket and shared memory.

- A typical VMM must save and restore all state that is necessary to support a guest OS. In Dune, we can limit the differences in guest and host state because processes using Dune have a narrower hardware interface. This results in reductions to the overhead of performing VM entries and VM exits.

- VMMs place each VM in a separate address space that emulates flat physical memory. In Dune, we configure the EPT to reflect process address spaces. As a result, the memory layout can be sparse and memory can be coherently shared when two processes map the same memory segment.

Despite these differences, the Dune module could be considered a type-2 hypervisor [22] because it runs on top of an existing OS kernel.

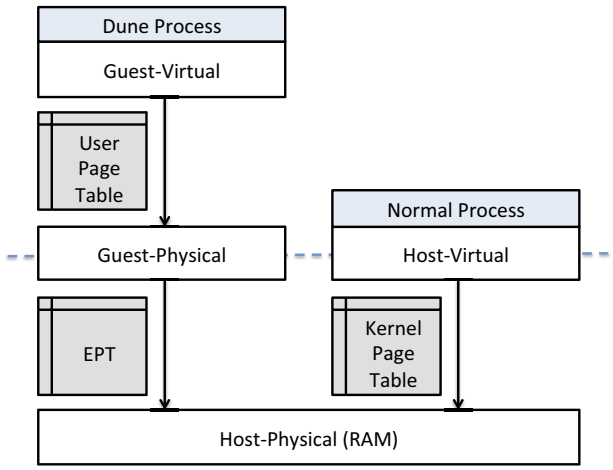

Figure 2: Virtual memory in Dune.

3.4 Memory Management

Memory management is one of the biggest responsibilities of the Dune module. The challenge is to expose direct page table access to user programs while preventing arbitrary access to physical memory. Moreover, our goal is to provide a normal process memory address space by default, permitting user programs to add just the functionality they need instead of completely replacing kernel-level memory management.

Paging translations occur in three separate cases in Dune, shown in Figure 2. One translation is specified by the kernel's standard page table. In virtualization terminology this is the host-virtual to host-physical (*i.e.,* raw memory) translation. Host-virtual addresses are ordinary virtual addresses, but they are only used by the kernel and normal processes. For processes using Dune, a user controlled page table maps guest-virtual addresses to guest-physical. Then the EPT, managed by the kernel, performs an additional translation from guest-physical to host-physical. All memory references made by processes using Dune can only be guest-virtual, allowing for isolation and correctness to be enforced in the EPT while application-specific functionality and optimizations can be applied in the user page table.

Ideally, we would like to match the EPT to the kernel's page table as closely as possible because of our goal to give processes using Dune access to the same address space they would have as normal processes. If it were permitted by hardware, we would simply point the EPT and the kernel's page table to the same page root. Unfortunately, two limitations make this impossible. First, the EPT requires a different binary format from the standard x86 page table. Second, Intel x86 processors limit the address width of guest-physical addresses to be the same as host-physical addresses. In a standard virtual machine environment this would not be a concern because any machine being emulated would have a realistically bounded amount of RAM. For Dune, however, the problem is that we want to expose the full host-virtual address space and yet the guest-physical address space is limited to a smaller size (*e.g.,* a 36-bit physical limit vs. a 48-bit virtual limit on many contemporary Intel processors). We note that this issue is not present when running in 32-bit protected mode, as physical addresses are at least as large as virtual addresses.

Our solution to EPT format incompatibility is to query the kernel for process memory mappings and to manually update the EPT to reflect them. We start with an empty EPT. Then, we receive an EPT fault (a type of VM exit) each time a missing EPT entry is accessed. The fault handler crafts a new EPT entry that reflects an address translation and permission reported by the kernel's page fault handler. Occasionally, address ranges will need to be unmapped. In addition, the kernel requires page access information, to assist with swapping, and page dirty status, to determine when write-back to disk is necessary. Dune supports all of these cases by hooking into an MMU notifier chain, the same approach used by KVM [30]. For example, when an address is unmapped, the Dune module receives an event. It then evicts affected EPT entries and sets dirty bits in the appropriate Linux page structures.

We work around the address width issue by allowing only some address ranges to be mapped in the EPT. Specifically, we only permit addresses from the beginning of the process (*i.e.,* the heap, code, and data segments), the mmap region, and the stack. Currently, we limit each of these regions to 4GB, allowing us to compress the address space to fit in the first 12GB of the EPT. Typically the user's page table will then expand the addresses to their original layout. This could result in incompatibilities in programs that use nonstandard portions of the address space, though such cases are rare. A more sophisticated solution might pack each virtual memory area into the guest-physical address space in arbitrary order and then provide the user program the additional information required to remap the segment to the correct guest-virtual address in its own page table, thus avoiding the possibility of unaddressable memory regions.

3.5 Exposing Access to Hardware

As discussed previously, Dune exposes access to exceptions, virtual memory, and privilege modes. Exceptions and privilege modes are implicitly available in VMX non-

root mode and do not require any special configuration. On the other hand, virtual memory requires access to the %CR3 register, which can be granted in the VMCS. We maintain a separate VMCS for each process in order to allow for per-process configuration of privileged state and to support context switching more easily and efficiently.

x86 includes a variety of control registers that determine which hardware features are enabled (*e.g.,* floating point, SSE, no execute, *etc.*) Although we could have permitted Dune processes to configure these directly, we instead mirror the configuration set by the kernel. This allows us to support a normal process environment; permitting many configuration changes would break compatibility with user programs. For example, it makes little sense for a 64-bit process to disable long mode. There are, however, a couple of important exceptions to this rule. First, we allow user programs to disable paging because it is the only method available on x86 to clear global TLB entries. Second, we give user programs some control over floating point hardware in order to allow for support of lazy floating point state management.

In some cases, Dune restricts access to hardware registers for performance reasons. For instance, Dune does not allow modification to MSRs in order to avoid the relatively high overhead of saving and restoring them during each system call. The FS and GS base registers are exceptions because they are not only used frequently but are also saved and restored by hardware automatically. MSR_LSTAR, which contains the address of the system call handler, is a special case where Dune allows read-only access. This allows a user process to map code for a system call handler at the existing address (by manipulating its page table) instead of changing the register to a new address and, as a result, harming performance.

Dune exposes raw access to the time stamp counter (TSC). By contrast, most VMMs virtualize the TSC in order to avoid confusing guest kernels, which tend to make timing assumptions that could be violated if time spent in the VMM is made visible.

3.6 Preserving OS Interfaces

In addition to exposing privileged hardware features, Dune preserves access to OS system calls. Normal system call invocation instructions will only trap within the process itself and do not cause a VM exit. Instead, processes must use VMCALL, the hypercall instruction, to make system calls. The Dune module vectors hypercalls through the kernel's system call table. In some cases, it must perform extra actions before calling the system call handler. For example, during an *exit* system call, Dune performs cleanup tasks.

Dune completely changes how signal handlers are invoked. Some signals are obviated by more efficient direct hardware support. For example, hardware page faults largely subsume the role of SIGSEGV. For other signals (*e.g.,* SIGINT), the Dune module injects fake hardware interrupts into the process. This is not only an efficient mechanism, but also has the advantage of correctly composing with privilege modes. For example, if a user process were running in ring 3 to sandbox untrusted code, hardware would automatically transition it to ring 0 in order to service the signal securely.

3.7 Implementation

Dune presently supports Linux running on Intel x86 processors in 64-bit long mode. Support for AMD CPUs and 32-bit mode are possible future additions. In order to keep changes to the kernel as unintrusive as possible, we developed Dune as a dynamically loadable kernel module. Our implementation is based partially on KVM [30]. Specifically, it shares code for managing low-level VT-x operations. However, high-level code is not shared with KVM because Dune operates differently from a VMM. Furthermore, our Dune module is simpler than KVM, consisting of only 2,509 lines of code.

In Linux, user threads are supported by the kernel, making them nearly identical to processes except they have a shared address space. As a result, it was easiest for us to create a VMCS for each thread instead of merely each process. One interesting consequence is that it is possible for both threads using Dune and threads not using Dune to belong to the same process.

Our implementation is capable of supporting thousands of processes at a time. The reason is that processes using Dune are substantially lighter-weight than full virtual machines. Efficiency is further improved by using virtual-processor identifiers (VPIDs). VPIDs enable a unique TLB tag to be assigned to each Dune process, and, as a result, hypercalls and context switches do not require TLB invalidations.

One limitation in our implementation is that we cannot efficiently detect when EPT pages have been modified or accessed, which is needed for swapping. Intel recently added hardware support for this capability, so it should be easy to rectify this limitation. For now, we take a conservative approach and always report pages as modified and accessed during MMU notifications in order to ensure correctness.

4 User-mode Environment

The execution environment of a process using Dune has some differences from a normal process. Because privilege rings are an exposed hardware feature, one difference is that user code runs in ring 0. Despite changing the behavior of certain instructions, this does not typically result in any incompatibilities for existing code. Ring 3 is also available and can optionally be used to confine untrusted code. Another difference is that system calls must be performed as hypercalls. To simplify supporting this change, we provide a mechanism that can detect when a system call is performed from ring 0 and automatically redirect it to the kernel as a hypercall. This is one of many features included in libDune.

libDune is a library created to make it easier to build user programs that make use of Dune. It is completely untrusted by the kernel and consists of a collection of utilities that aid in managing and configuring privileged hardware features. Major components of libDune include a page table manager, an ELF loader, a simple page allocator, and routines that assist user programs in managing exceptions and system calls. libDune is currently 5,898 lines of code.

We also provide an optional, modified version of libc that uses `VMCALL` instructions instead of `SYSCALL` instructions in order to get a slight performance benefit.

4.1 Bootstrapping

In many ways, transitioning a process into Dune mode is similar to booting an OS. The first issue is that a valid page table must be provided before enabling Dune. A simple identity mapping is insufficient because, although the goal is to have process addresses remain consistent before and after the transition, the compressed layout of the EPT must be taken into account. After a page table is created, the Dune entry *ioctl* is called with the page table root as an argument. The Dune module then switches the process to Dune mode and begins executing code, using the provided page table root as the initial `%CR3`. From there, libDune configures privileged registers to set up a reasonable operating environment. For example, it loads a GDT to provide basic flat segmentation and loads an IDT so that hardware exceptions can be captured. It also sets up a separate stack in the TSS to handle double faults and configures the GS segment base in order to easily access per-thread data.

4.2 Limitations

Although we are able to run a wide variety of Linux programs, libDune is still missing some functionality. First, we have not fully integrated support for signals despite the fact that they are reported by the Dune module. Applications are required to use *dune_signal* whereas a more compatible solution would override several libc symbols like *signal* and *sigaction*. Second, although we support pthreads, some utilities in libDune, such as page table management, are not yet thread-safe. Both of these issues could be resolved with further implementation.

One unanticipated challenge with working in a Dune environment is that system call arguments must be valid host-virtual addresses, regardless of how guest-virtual mappings are setup. In many ways, this parallels the need to provide physical addresses to hardware devices that perform DMA. In most cases we can work around the issue by having the guest-virtual address space mirror the host-virtual address space. For situations where this is not possible, walking the user page table to adjust system call argument addresses is necessary.

Another challenge introduced by Dune is that by exposing greater access to privileged hardware, user programs require more architecture-specific code, potentially reducing portability. libDune currently provides an x86-centric API, so it is already compatible with AMD machines. However, it should be possible to modify libDune to support non-x86 architectures in a fashion that parallels the construction of many OS kernels. This would require libDune to provide an efficient architecture independent interface, a topic worth exploring in future revisions.

5 Applications

Dune is generic enough that it lets us improve on a broad range of applications. We built two security-related applications, a sandbox and privilege separation system, and one performance-related application, a garbage collector. Our goals were simpler implementations, higher performance, and where applicable, improved security.

5.1 Sandboxing

Sandboxing is the process of confining code so as to restrict the memory it can access and the interfaces or system calls it can use. It is useful for a variety of purposes, such as running native code in web browsers, creating secure OS containers, and securing mobile phone applications. In order to explore Dune's potential for these types

of applications, we built a sandbox that supports native 64-bit Linux executables.

The sandbox enforces security through privilege modes by running a trusted sandbox runtime in ring 0 and an untrusted binary in ring 3, both operating within a single address space (on the left of Figure 1, the top and middle boxes respectively). Memory belonging to the sandbox runtime is protected by setting the supervisor bit in appropriate page table entries. Whenever the untrusted binary performs an unsafe operation such as trying to access the kernel through a system call or attempting to modify privileged state, libDune receives an exception and jumps into a handler provided by the sandbox runtime. In this way, the sandbox runtime is able to filter and restrict the behavior of the untrusted binary.

While we rely on the kernel to load the sandbox runtime, the untrusted binary must be loaded in userspace. One risk is that it could contain maliciously crafted headers designed to exploit flaws in the ELF loader. We hardened our sandbox against this possibility by using two separate ELF loaders. First, the sandbox runtime uses a minimal ELF loader (part of libDune), that only supports static binaries, to load a second ELF loader into the untrusted environment. We choose to use *ld-linux.so* as our second ELF loader because it is already used as an integral and trusted component in Linux. Then, the sandbox runtime executes the untrusted environment, allowing the second ELF loader to load an untrusted binary entirely from ring 3. Thus, even if the untrusted binary is malicious, it does not have a greater opportunity to attack the sandbox during ELF loading than it would while running inside the sandbox normally.

So far our sandbox has been applied primarily as a tool for filtering Linux system calls. However, it could potentially be used for other purposes, including providing a completely new system call interface. For system call filtering, a large concern is to prevent execution of any system call that could corrupt or disable the sandbox runtime. We protect against this hazard by validating each system call argument, checking to make sure performing the system call would not allow the untrusted binary to access or modify memory belonging to the sandbox runtime. We do not yet support all system calls, but we support enough to run most single-threaded Linux applications. However, nothing prevents supporting multi-threaded programs in the future.

We implemented two policies on top of the sandbox. Firstly, we support a null policy that allows system calls to pass through but still validates arguments in order to protect the sandbox runtime. It is intended primarily to demonstrate raw performance overhead. Secondly, we support a userspace firewall. It uses system call interposition to inspect important network system calls, such as *bind* and *connect*, and prevents communication with undesirable parties as specified by a policy description.

To further demonstrate the flexibility of our sandbox, we also implemented a checkpointing system that can serialize an application to disk and then restore execution at a later time. This includes saving memory, registers, and system call state (*e.g.,* open file descriptors).

5.2 Wedge

Wedge [10] is a privilege separation system. Its core abstraction is an *sthread* which provides *fork*-like isolation with *pthread*-like performance. An sthread is a lightweight process that has access to memory, file descriptors and system calls as specified by a policy. The idea is to run risky code in an sthread so that any exploits will be contained within it. In a web server, for example, each client request would run in a separate sthread to guarantee isolation between users. To make this practical, sthreads need fast creation (*e.g.,* one per request) and context switch time. Fast creation can be achieved through sthread *recycling*. Instead of creating and killing an sthread each time, an sthread is checkpointed on its first creation (while still pristine and unexploited) and restored on exit so that it can be safely reused upon the next creation request. Doing so reduces sthread creation cost to the (cheaper) cost of restoring memory.

Wedge uses many of Dune's hardware features. Ring protection is used to enforce system call policies; page tables limit what memory sthreads can access; dirty bits are used to restore memory during sthread recycling; and the tagged TLB is used for fast context switching.

5.3 Garbage Collection

Garbage collectors (GC) often utilize memory management hardware to speed up collection [28]. Appel and Li [5] explain several techniques that use standard user level virtual memory protection operations, whereas Azul Systems [15, 36] went to the extent of modifying the kernel and system call interface. By contrast, Dune provides a clean and efficient way to access relevant hardware directly. The features provided by Dune that are of interest to garbage collectors include:

- **Fast faults**. GCs often use memory protection and fault handling to implement read and write barriers.
- **Dirty bits**. Knowing what memory has been touched since the last collection enables optimizations and can be a core part of the algorithm.

- **Page table**. One optimization in a moving GC is to free the underlying physical frame without freeing the virtual page it was backing. This is useful when the data has been moved but references to the old location remain and can still be caught through page faults. Remapping memory can also be performed to reduce fragmentation.
- **TLB control**. GCs often manipulate memory mappings at high rates, making control over TLB invalidation very useful. If it can be controlled, mapping manipulations can be effectively batched, rendering certain algorithms more feasible.

We modified the Boehm GC [12] to use Dune in order to improve performance. The Boehm GC is a robust mark-sweep collector that supports parallel and incremental collection. It is designed either to be used as a conservative collector with C/C++ programs, or by compiler and run-time backends where the conservativeness can be controlled. It is widely used, including by the Mono project and GNU Objective C.

An important implementation question for the Boehm GC is how dirty pages are discovered and managed. The two original options were (i) utilizing *mprotect* and signal handlers to implement its own dirty bit tracking; or (ii) utilizing OS provided dirty bit read methods such as the Win32 API call *GetWriteWatch*. In Dune we support and improve both methods.

A direct port to Dune already gives a performance improvement because *mprotect* can directly manipulate the page table and a page fault can be handled directly without needing an expensive `SIGSEGV` signal. The GC manipulates single pages 90% of the time, so we were able to improve performance further by using the `INVPLG` instruction to flush only a single page instead of the entire TLB. Finally, in Dune, the Boehm GC can access dirty bits directly without having to emulate this functionality. Some OSes provide system calls for reading page table dirty bits. Not all of these interfaces are well matched to GC—for instance, SunOS examines the entire virtual address space rather than permit queries for a particular region. Linux provides no user-level access at all to dirty bits.

The work done on the Boehm GC represents a straightforward application of Dune to a GC. It is worth also examining the changes made by Azul Systems to Linux so that they could support their C4 GC [36] and mapping this to the support provided by Dune:

- **Fast faults**. Azul modified the Linux memory protection and mapping primitives to greatly improve performance, part of this included allowing hardware exceptions to bypass the kernel and be handled directly by usermode.
- **Batched page table**. Azul enabled explicit control of TLB invalidation and a shadow page table to expose a prepare and commit style API for batching page table manipulation.
- **Shatter/Heal**. Azul enabled large pages to be 'shattered' into small pages or a group of small pages to be 'healed' into a single large page.
- **Free physical frames**. When the Azul C4 collector frees an underlying physical frame, it will trap on accesses to the unmapped virtual pages in order to catch old references.

All of the above techniques and interfaces can be implemented efficiently on top of Dune, with no need for any kernel changes other than loading the Dune module.

6 Evaluation

In this section, we evaluate the performance and utility of Dune. Although using VT-x has an intrinsic cost, in most cases, Dune's overhead is relatively minor. On the other hand, Dune offers significant opportunities to improve security and performance for applications that can take advantage of access to privileged hardware features.

All tests were performed on a single-socket machine with an Intel Xeon E3-1230 v2 (a four core Ivy Bridge CPU clocked at 3.3 GHz) and 16GB of RAM. We installed a recent 64-bit version of Debian Linux that includes Linux kernel version 3.2. Power management features, such as frequency scaling, were disabled.

6.1 Overhead from Running in Dune

Performance in Dune is impacted by two main sources of overhead. First, VT-x increases the cost of entering and exiting the kernel—VM entries and VM exits are more expensive than fast system call instructions or exceptions. As a result, both system calls and other types of faults (*e.g.,* page faults) must pay a fixed cost in Dune. Second, using the EPT makes TLB misses more expensive because, in some cases, the hardware page walker must traverse two page tables instead of one.

We built synthetic benchmarks to measure both of these effects. Table 2 shows the overhead of system calls, page faults, and page table walks. For system calls, we manually performed *getpid*, an essentially null system call (worst case for Dune), and measured the round-trip latency. For page faults, we measured the time it took to fault in a pre-zeroed memory page by the kernel. Finally, for page table walks, we measured the time spent filling a TLB miss.

	getpid	page fault	page walk
Linux	138	2,687	35.8
Dune	895	5,093	86.4

Table 2: Average time (in cycles) of operations that have overhead in Dune compared to Linux.

	ptrace	trap	appel1	appel2
Linux	27,317	2,821	701,413	684,909
Dune	1,091	587	94,496	94,854

Table 3: Average time (in cycles) of operations that are faster in Dune compared to Linux.

Measuring TLB miss overhead required us to build a simple memory stress tool. It works by performing a random page-aligned walk across 2^{16} memory pages. This models a workload with poor memory locality, as nearly every memory access results in a last-level TLB miss. We then divided the total number of cycles spent waiting for page table walks, as reported by a performance counter, by the total number of memory references, giving us a cycle cost per page walk.

In general, the overhead Dune adds has only a small effect on end-to-end performance, as we show in Section 6.3. For system calls, the time spent in the kernel tends to be a larger cost than the fixed VMX mode transition costs. Page fault overhead is also not much of a concern, as page faults tend to occur infrequently during normal use, and direct access to exception hardware is available when higher performance is required. On the other hand, Dune's use of the EPT does impact performance in certain workloads. For applications with good memory locality or a small working set, it has no impact because the TLB hit rate is sufficiently high. However, for application with poor memory locality or a large working set, more frequent TLB misses result in a measurable slowdown. One effective strategy for limiting this overhead is to use large pages. We explore this possibility further in section 6.3.1.

6.2 Optimizations Made Possible by Dune

Access to privileged hardware features creates many opportunities for optimization. Table 3 shows speedups we achieved in the following OS workloads:

ptrace is a measure of system call interposition performance. This is the cost of a Linux process intercepting a system call (*getpid*) with *ptrace*, forwarding the system call to the kernel and returning the result. In Dune this is the cost of intercepting a system call directly using ring protection in VMX non-root mode, forwarding the system call through a `VMCALL` and returning the result. An additional scenario is where applications wish to intercept system calls but not forward them to the kernel and instead just implement them internally. `PTRACE_SYSEMU` is the most efficient mechanism for doing so since *ptrace* requires forwarding a call to the kernel. The latency of intercepting a system call with `PTRACE_SYSEMU` is 13,592 cycles. In Dune this can be implemented by handling the hardware system call trap directly, with a latency of just 180 cycles. This reveals that most of the Dune *ptrace* benchmark overhead was in fact forwarding the *getpid* system call via a `VMCALL` rather than intercepting the system call.

trap indicates the time it takes for a process to get an exception for a page fault. We compare the latency of a SIGSEGV signal in Linux with a hardware-generated page fault in Dune.

appel1 is a measure of user-level virtual memory management performance. It corresponds to the TRAP, PROT1, and UNPROT test described in [5], where 100 protected pages are accessed, causing faults. Then, in the fault handler, the faulting page is unprotected, and a new page is protected.

appel2 is another measure of user-level virtual memory management performance. It corresponds to the PROTN, TRAP, and UNPROT test described in [5], where 100 pages are protected. Then each is accessed, with the fault handler unprotecting the faulting page.

6.3 Application Performance

6.3.1 Sandbox

We evaluated the performance of our sandbox by running two types of workloads. First, we tested compute performance by running SPEC2000. Second, we tested IO performance by running lighttpd. The null sandbox policy was used in both cases.

Figure 3 shows the performance of SPEC2000. In general, the sandbox had very low overhead, averaging only 2.9% percent slower than Linux. However, the *mcf* and *ammp* benchmarks were outliers, with 20.9% and 10.1% slowdowns respectively. This deviation in performance can be explained by EPT overhead, as we observed a high TLB miss rate. We also measured SPEC2000 in VMware Player, and, as expected, EPT overhead resulted in very similar drops in performance.

We then adjusted the sandbox to avoid EPT overhead by backing large memory allocations with 2MB large

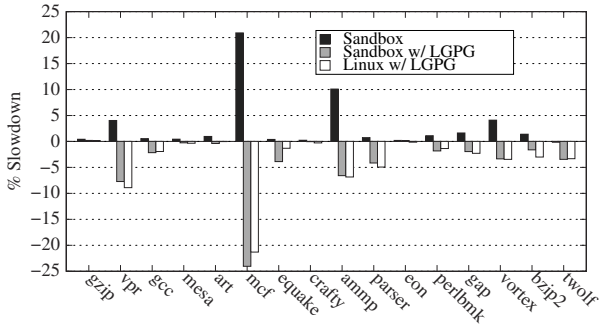

Figure 3: Average SPEC2000 slowdown compared to Linux for the sandbox, the sandbox using large pages, and Linux using large pages identically.

	1 client	100 clients
Linux	2,236	24,609
Dune sandbox	2,206	24,255
VMware Player	734	5,763

Table 4: Lighttpd performance (in requests per second).

pages, both in the EPT and the user page table. Supporting this optimization was straightforward because we were able to intercept *mmap* calls and transparently modify them to use large pages. Such an approach does not cause much memory fragmentation because large pages are only used selectively. In order to perform a direct comparison, we tested SPEC2000 in a modified Linux environment that allocates large pages in an identical fashion using libhugetlbfs [1]. When large pages were used for both, average performance in the sandbox and Linux was nearly identical (within 0.1%).

Table 4 shows the performance of lighttpd, a single-threaded, single-process, event-based HTTP server. Lighttpd exercises the kernel to a much greater extent than SPEC2000, making frequent system calls and putting load on the network stack. Lighttpd performance was measured over Gigabit Ethernet using the Apache *ab* benchmarking tool. We configured *ab* to repeatedly retrieve a small HTML page over the network with different levels of concurrency: 1 client for measuring latency and 100 clients for measuring throughput.

We found that the sandbox incurred only a slight slowdown, less than 2% for both the latency and throughput test. This slowdown can be explained by Dune's higher system call overhead. Using *strace*, we determined that lighttpd was performing several system calls per connection, causing frequent VMX transitions. However, VMware Player, a conventional VMM, experienced much greater overhead: 67% for the latency test and 77% for the throughput test. Although VMware Player pays VMX transition costs too, the primary reason for the slowdown is that each network request must traverse two network stacks, one in the guest and one in the host.

We also found that the sandbox provides an easily extensible framework that we used to implement checkpointing and our firewall. The checkpointing implementation consisted of approximately 450 SLOC with 50 of those being enhancements to the sandbox loader. Our firewall was around 200 SLOC with half of that being the firewall rules parser.

	create	ctx switch	http request
fork	81	0.49	454
Dune sthread	2	0.15	362

Table 5: Wedge benchmarks (times in microseconds).

6.3.2 Wedge

Wedge has two main benchmarks: sthread creation and context switch time. These are compared to *fork*, the system call used today to implement privilege separation. As shown in Table 5, sthread creation is faster than *fork* because instead of creating a new process each time, an sthread is reused from a pool and "recycled" by restoring dirty memory and state. Context switch time in sthreads is low because TLB flushes are avoided by using the tagged TLB. In Dune sthreads are created 40× faster than processes and the context switch time is 3× faster. In previous Wedge implementations sthread creation was 12× faster than *fork* with no improvement in context switch time [9]. Dune is faster because it can leverage the tagged TLB and avoid kernel calls to create sthreads. The last column of Table 5 shows an application benchmark of a web server serving a static file on a LAN where each request runs in a newly *fork*ed process or sthread for isolation. Dune sthreads show a 20% improvement here.

The original Wedge implementation consisted of a 700-line kernel module and a 1,300-line library. A userspace-only implementation of Wedge exists, though the authors lamented that POSIX did not offer adequate APIs for memory and system call protection, hence the result was a very complicated 5,000-line implementation [9]. Dune instead exposes the hardware protection features needed for a simple implementation, consisting of only 750 lines of user code.

	GCBench	LinkedList	HashMap	XML
Collections	542	33,971	161	10
Memory use (MB)				
Allocation	938	15,257	10,352	1,753
Heap	28	1,387	27	1,737
Execution time (ms)				
Normal	1,224	15,983	14,160	6,663
Dune	1,176	16,884	13,715	7,930
Dune TLB	933	14,234	11,124	7,474
Dune dirty	888	11,760	8,391	6,675

Table 6: Performance numbers of the GC benchmarks.

6.3.3 Garbage Collector

We implemented three different sets of modifications to the Boehm GC. The first is the simplest port possible with no attempt to utilize any advanced features of Dune. This benefits from Dune's fast memory protection and fault handling but suffers from the extra TLB costs. The second version improves the direct port by carefully controlling when the TLB is invalidated. The third version avoids using memory protection altogether, instead it reads the dirty bits directly. The direct port required changing 52 lines, the TLB optimized version 91 lines, and the dirty bit version 82 lines.

To test the performance improvements of these changes we used the following benchmarks:

- GCBench [11]. A microbenchmark written by Hans Boehm and widely used to test garbage collector performance. In essence, it builds a large binary tree.
- Linked List. A microbenchmark that builds increasingly large linked lists of integers, summing each one after it is built.
- Hash Map. A microbenchmark that utilizes the Google sparse hash map library [23] (C version).
- XML Parser. A full application that uses the Mini-XML library [34] to parse a 150MB XML file containing medical publications. It then counts the number of publications each author has using a hash map.

The results for these benchmarks are presented in Table 6. The direct port displays mixed results due to the improvement to memory protection and the fault handler but slowdown of EPT overhead. As soon as we start using more hardware features, we see a clear improvement over the baseline. Other than the XML Parser, the TLB version improves performance between 10.9% and 23.8%, and the dirty bit version between 26.4% and 40.7%.

The XML benchmark is interesting as it shows a slowdown under Dune for all three versions: 19.0%, 12.2% and 0.2% slower for the direct, TLB and dirty version respectively. This appears to be caused by EPT overhead, as the benchmark does not create enough garbage to benefit from the modifications we made to the Boehm GC. This is indicated in Table 6; the total amount of allocation is nearly equal to the maximum heap size. We verified this by modifying the benchmark to instead take a list of XML files, processing each sequentially so that memory would be recycled. We then saw a linear improvement in the Dune versions over the baseline as the number of files was increased. With ten 150MB XML files as input, the dirty bit version of the Boehm GC showed a 12.8% improvement in execution time over the baseline.

7 Reflections on Hardware

While developing Dune, we found VT-x to be a surprisingly flexible hardware mechanism. In particular, the fine-grained control provided by the VMCS allowed us to precisely direct how hardware was exposed. However, some hardware changes to VT-x could benefit Dune. One noteworthy area is the EPT, as we encountered both performance overhead and implementation challenges. Hardware modifications have been proposed to mitigate EPT overhead [2, 8]. In addition, modifying the EPT to support the same address width as the regular page table would reduce the complexity of our implementation and improve coverage of the process address space. Further reductions to VM exit and VM entry latency could also benefit Dune. However, we were able to aggressively optimize hypercalls, and VMX transition costs had only a small effect on the performance of the applications we evaluated.

There are a few hardware features that we have not yet exposed, despite the fact that they are available in VT-x and possible to support in Dune. Most seem useful only in special situations. For example, a user program might want to have control over caching in order to prevent information leakage. However, this would only be effective if CPU affinity could be controlled. As another example, access to efficient polling instructions (*i.e.,* `MONITOR` and `MWAIT`) could be useful in reducing power consumption for userspace messaging implementations that perform cache line polling. Finally, exposing access to debug registers could allow user programs to more efficiently set up memory watchpoints.

It may also be useful to provide Dune applications with direct access to IO devices. Many VT-x systems include support for an IOMMU, a device that can be used to make DMA access safe even when it is available to untrusted software. Thus, Dune could be modified to safely expose certain hardware devices. A potential benefit could be reduced IO latency. The availability of SR-IOV makes this

possibility more practical because it allows a single physical device to be partitioned across multiple guests.

Recently, a variety of non-x86 hardware platforms have gained support for hardware-assisted virtualization, including ARM [38], Intel Itanium [25], and IBM Power [24]. ARM is of particular interest because of its prevalence in mobile devices, making the ARM Virtualization Extensions an obvious future target for Dune. ARM's support for virtualization is similar to VT-x in some areas. For example, ARM is capable of exposing direct access to privileged hardware features, including exceptions, virtual memory, and privilege modes. Moreover, ARM provides a System MMU, which is comparable to the EPT. ARM's most significant difference is that it introduces a new deeper privilege mode call *Hyp* that runs underneath the guest kernel. In contrast, VT-x provides separate operating modes for the guest and VMM. Another difference from VT-x is that ARM does not automatically save and restore architectural state when switching between a VMM and a guest. Instead, the VMM is expected to manage state in software, perhaps creating an opportunity for optimization.

8 Related Work

There have been several efforts to give applications greater access to hardware. For example, The Exokernel [18] exposes hardware features through a low-level kernel interface that allows applications to manage hardware resources directly. Another approach, adopted by the SPIN project [7], is to permit applications to safely load extensions directly into the kernel. Dune shares many similarities with these approaches because it also tries to give applications greater access to hardware. However, Dune differs because its goal is not extensibility. Rather, Dune provides access to privileged hardware features so that they can be used in concert with the OS instead of a means of modifying or overriding it.

The Fluke project [20] supports a nested process model in software, allowing OSes to be constructed "vertically." Dune complements this approach because it could be used to efficiently support an extra OS layer between the application and the kernel through use of privilege mode hardware. However, the hardware that Dune exposes can only support a single level instead of the multiple levels available in Fluke.

A wide range of strategies have been employed to support sandboxing, such as ptrace [16], dedicated kernel modifications [16, 21, 33], binary translation [19], and binary verification [39]. To our knowledge, Dune is the first system to support sandboxing entirely through user-level access to hardware protection, improving performance and reducing code complexity. For example, Native Client [39] reports an average SPEC2000 overhead of 5% with a worst case performance of 12%—anecdotally, we observed higher overheads on modern microarchitectures. By contrast, we were able to achieve nearly zero average overhead (1.4% worst case) for the same benchmarks in Dune. Our sandbox is similar to Native Client in that it creates a secure subdomain within a process. However, Native Client is more portable than Dune because it does not require virtualization hardware or kernel changes.

Like Dune, some previous work has used hardware virtualization for non-traditional purposes. For example, VT-x has been suggested as a tool for creating rootkits [29] that are challenging to detect. Moreover, IOMMU hardware has been used to safely isolate malicious device drivers by running them in Linux processes [13].

9 Conclusion

Dune provides ordinary applications with efficient and safe access to privileged hardware features that are traditionally available only to kernels. It does so by leveraging modern virtualization hardware, which enables direct execution of privileged instructions in unprivileged contexts. Our implementation of Dune for Linux uses Intel's VT-x virtualization architecture and provides application-level access to exceptions, virtual memory, and privilege modes. Our evaluation shows both performance and security benefits to Dune. For instance, we built a sandbox that approaches zero overhead, modified a garbage collector to improve performance by up to 40.7%, and created a privilege separation system with 3× less context switch overhead than without Dune.

In an effort to spur adoption, we have structured Dune as a module that works with unmodified Linux kernels. We hope the applications described in this paper are just the first of many uses people will find for the system. The hardware mechanisms exposed by Dune are at the core of many operating systems innovations; their new accessibility from user-level creates opportunities to deploy novel systems without kernel modifications. Dune is freely available at `http://dune.scs.stanford.edu/`.

Acknowledgments

We wish to thank our shepherd, Timothy Roscoe, for his guidance and valuable suggestions. We would also like to thank Edouard Bugnion for feedback on several itera-

tions of this paper and for his valuable discussions during the early phases of Dune. Finally, we thank Richard Uhlig, Jacob Leverich, Ben Serebrin, and our anonymous reviewers for suggestions that greatly shaped our paper. This work was funded DARPA CRASH under contract #N66001-10-2-4088 and by a gift from Google. Adam Belay is supported by a Stanford Graduate Fellowship.

References

[1] Libhugetlbfs. `http://libhugetlbfs.sourceforge.net`, Apr. 2012.

[2] J. Ahn, S. Jin, and J. Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 476–487, 2012.

[3] AMD. *Secure Virtual Machine Architecture Reference Manual*.

[4] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 13–15, 2007.

[5] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Apr. 1991.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, 1995.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

[9] A. Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, 2009.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, 2008.

[11] H. Boehm. GC Bench. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`, Apr. 2012.

[12] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, 1991.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, pages 9–9, 2010.

[14] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, 1997.

[15] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, 2005.

[16] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation*, OSDI'08, pages 339–354, 2008.

[17] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, Orcas Island, Washington, May 1995.

[18] 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, 1995.

[19] B. Ford and R. Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 293–306, 2008.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, 1996.

[21] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[22] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[23] Google. sparsehash. `http://code.google.com/p/sparsehash/`, Apr. 2012.

[24] IBM. *Power ISA, Version 2.06 Revision B*.

[25] Intel. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*.

[26] Intel Corporation. Invalid Instruction Erratum Overview. `http://www.intel.com/support/processors/pentium/sb/cs-013151.htm`, Apr. 2012.

[27] K. Kaspersky and A. Chang. Remote Code Execution thorugh Intel CPU Bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[28] H. Kermany and E. Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, 2006.

[29] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, 2006.

[30] A. Kivity. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[31] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA. *IBM Syst. J.*, 30(1):34–51, Feb. 1991.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[33] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, SSYM'03, 2003.

[34] M. Sweet. Mini-XML: Lightweight XML Library. `http://www.minixml.org/`, Apr. 2012.

[35] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[36] G. Tene, B. Iyengar, and M. Wolf. C4: the Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, 2011.

[37] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48 – 56, May 2005.

[38] P. Varanasi and G. Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, 2011.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, 2009.

[40] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 13–24, 1987.