Lightweight Kernel Isolation with Virtualization and VM Functions

Vikram Narayanan
University of California, Irvine

Yongzhe Huang
Pennsylvania State University

Gang Tan
Pennsylvania State University

Trent Jaeger
Pennsylvania State University

Anton Burtsev
University of California, Irvine

Abstract

Commodity operating systems execute core kernel subsystems in a single address space along with hundreds of dynamically loaded extensions and device drivers. Lack of isolation within the kernel implies that a vulnerability in any of the kernel subsystems or device drivers opens a way to mount a successful attack on the entire kernel.

Historically, isolation within the kernel remained prohibitive due to the high cost of hardware isolation primitives. Recent CPUs, however, bring a new set of mechanisms. Extended page-table (EPT) switching with VM functions and memory protection keys (MPKs) provide memory isolation and invocations across boundaries of protection domains with overheads comparable to system calls. Unfortunately, neither MPKs nor EPT switching provide architectural support for isolation of privileged ring 0 kernel code, i.e., control of privileged instructions and well-defined entry points to securely restore state of the system on transition between isolated domains.

Our work develops a collection of techniques for lightweight isolation of privileged kernel code. To control execution of privileged instructions, we rely on a minimal hypervisor that transparently deprivileges the system into a non-root VT-x guest. We develop a new isolation boundary that leverages extended page table (EPT) switching with the VMFUNC instruction. We define a set of invariants that allows us to isolate kernel components in the face of an intricate execution model of the kernel, e.g., provide isolation of preemptable, concurrent interrupt handlers. To minimize overheads of virtualization, we develop support for exitless interrupt delivery across isolated domains. We evaluate our approach by developing isolated versions of several device drivers in the Linux kernel.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VEE '20, March 17, 2020, Lausanne, Switzerland

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7554-2/20/03…$15.00
https://doi.org/10.1145/3381052.3381328

ACM Reference Format:
Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of (VEE '20).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3381052.3381328

1 Introduction

Despite many arguments for running kernel subsystems in separate protection domains over the years, commodity operating systems remain monolithic. Today, the lack of isolation within the operating system kernel is one of the main factors undermining its security. While the core kernel is relatively stable, the number of kernel extensions and device drivers is growing with every hardware generation (a modern Linux kernel contains around 8,867 device drivers [3], with around 80-130 drivers running on a typical system). Developed by third party vendors that often have an incomplete understanding of the kernel programming and security idioms, kernel extensions and device drivers are a primary source of vulnerabilities in the kernel [7, 17]. While modern kernels deploy a number of security mechanisms to protect their execution, e.g., stack canaries [20], address space randomization (ASLR) [49], data execution prevention (DEP) [88], superuser-mode execution and access prevention [19, 28], a large fraction of vulnerabilities remains exploitable. Even advanced defense mechanisms like code pointer integrity (CPI) [2, 57] and safe stacks [16] that are starting to make their way into the mainstream kernels remain subject to data-only attacks that become practical when combined with automated attack generation tools [54, 93]. Lack of isolation within the kernel implies that a vulnerability in any of the kernel subsystems creates an opportunity for an attack on the entire kernel.

Unfortunately, introducing isolation in a modern kernel is hard. The emergence of sub-microsecond 40-100Gbps network interfaces [66], and low-latency non-volatile PCIe-attached storage pushed modern kernels to support I/O system calls and device drivers capable of operating with latencies of low thousands of cycles. Minimal cycle budgets put strict requirements on the overheads of isolation solutions. For decades the only two isolation mechanisms exposed

by commodity x86 CPUs were segmentation and paging. Segmentation was deprecated when the x86 architecture moved from 32bit to 64bit addressing mode. On modern machines with recently introduced tagged TLBs, a carefully-optimized page-based inter-process communication mechanism requires 952 cycles for a cross-domain invocation [4]. With two domain crossings on the network transmission path, traditional page-based isolation solutions [83, 84] result in prohibitive overheads on modern systems.

Today, the landscape of isolation solutions is starting to change with the emergence of new hardware isolation primitives. VM function extended page-table (EPT) switching and memory protection keys (MPKs) provide support for memory isolation and cross-domain invocations with overheads comparable to system calls [45, 51, 62, 68]. Unfortunately, neither MPKs nor EPT switching implement architectural support for isolation of privileged ring 0 kernel code, the code that runs with superuser privileges and can easily escape such isolation by accessing a wide range of privileged CPU instructions. Traditionally, to control execution of privileged instructions, isolation of kernel subsystems requires an exit into ring 3 [6, 13, 22, 24, 29, 35, 36, 44, 55, 76, 85, 89, 92]. The change of the privilege level, however, can incur 0.35-6.3x overhead for relatively lightweight VMFUNC and MPK based isolation techniques.

Our work on Lightweight Virtualized Domains (LVDs) develops new mechanisms for isolation of privileged kernel code through a combination of hardware-assisted virtualization and EPT switching. First, to control execution of privileged instructions without requiring a change of the privilege level, we execute the system under control of a minimal late-launch hypervisor. When the isolated subsystem is loaded into the kernel we transparently deprivilege the system into a non-root VT-x guest. Effectively, we trade the cost of changing the privilege level on cross-domain invocations for exits into the hypervisor on execution of privileged instructions. We demonstrate that for I/O intensive workloads that pose the most demanding requirements on the cost of the isolation mechanisms this tradeoff is justified: the exits caused by privileged instructions are much less frequent compared to the number of crossings of isolation boundaries (Section 5).

Second, to protect the state of the system, we develop a new isolation boundary that leverages extended page tables (EPTs) and EPT switching with the VMFUNC instruction. VMFUNC allows the VT-x guest to change EPT mappings, and hence change the view of all accessible memory with a single instruction that takes 109-147 cycles [45, 56, 68]. Several projects explore the use of VMFUNC for isolation of code within user programs [45, 63], implementation of microkernel processes [68], and protecting legacy systems against the Meltdown speculative execution attack [51]. In contrast to previous work, we develop techniques for enforcing isolation of kernel subsystems. Kernel subsystems, e.g., device drivers, adhere to a complex execution model of the kernel, i.e., they run in the context of user and kernel threads, interrupts, and bottom-half software IRQs. Most driver code (including interrupt handlers) runs on multiple CPUs and is fully reentrant, i.e., it runs with interrupts enabled, calls back into the kernel, and can yield execution. VMFUNC does not provide architectural support for protecting the state of the system upon crossing boundaries of isolated domains. We define a set of security invariants and develop a collection of mechanisms that allow us to retain isolation in a complex execution environment of the kernel.

Finally, to minimize overheads introduced by running the system as a non-root VT-x guest, we develop support for exitless interrupt delivery. Even though modern device drivers use interrupt coalescing and polling, interrupt exits can be a major source of overhead in virtualized environments [39, 86]. We develop a collection of new mechanisms that allow us re-establish correct state of the system while handling an interrupt in a potentially untrusted state inside an isolated domain and without exiting into the hypervisor.

To evaluate practicality of our approach, we isolate several performance-critical device drivers of the Linux kernel. In general, irrespective of isolation boundary, isolation of kernel code is challenging as it cuts through a network of data and control flow dependencies in a complex, feature-rich system [72, 84]. To isolate device drivers, we leverage an existing device driver isolation framework, LXDs [72]. We evaluate overheads of LVDs on software-only network and NVMe block drivers, and on the 10Gbps Intel Ixgbe network driver.

We argue that our work—a practical, lightweight isolation boundary that supports isolation of kernel code without breaking its execution model—makes another step towards enabling isolation as a first-class abstraction in modern operating system kernels. Our isolation mechanisms can be implemented either as a loadable late-launch hypervisor that transparently provides isolation for a native non-virtualized system, or as a set of hypervisor extensions that enable isolation of kernel code in a virtualized environment. While we utilize EPTs for memory isolation, we argue that our techniques—control over privileged instructions, secure state saving, and exitless interrupts—are general and can be applied to other isolation mechanisms, for examplee MPK.

2 Background and Motivation

Historically, two factors shape the landscape of in-kernel isolation: the availability of hardware isolation mechanisms, and the complexity of decomposing existing shared-memory kernel code.

2.1 Isolation Mechanisms and Overheads

Segmentation and paging For decades the only two isolation mechanisms exposed by commodity x86 CPUs were

segmentation and paging. Segmentation was demonstrated as a low-overhead isolation mechanism by the pioneering work on L4 microkernel [60]. Unfortunately, segmentation was deprecated when the x86 architecture moved from 32bit to 64bit addressing mode. On modern machines with recently introduced tagged TLBs, a carefully-optimized page-based isolation mechanism requires 952 cycles for a minimal cross-domain invocation [4] (ironically, the cost of the context switch is growing over the years [25]). With two domain crossings on the network transmission path, page-based isolation solutions like Nooks [84] would introduce an overhead of more than 72%. Less optimized approaches like SIDE [83] that rely on a pagefault to detect cross-domain access in addition to a page table switch would result in a 2x slowdown.

Cache-coherent cross-core invocations With the commoditization of multi-core CPUs, multiple systems suggested cross-core invocations for acceleration of system calls [82] and cross-domain invocations [8, 50, 72]. Faster than address space switches, the cross-core invocations are still expensive. A minimal call/reply invocation requires four cache-line transactions [72] each taking 109-400 cycles [21, 70, 71] depending on whether the line is transferred between the cores of the same socket or over a cross-socket link. Hence the whole call/reply call takes 448-1988 cycles [72]. More importantly, during the cross-core invocation, the caller core has to wait for the reply from the callee core. At this point, two cores are involved in the cross-domain invocation, but the caller core is wasting time constantly checking for the reply in a tight loop. Asynchronous runtimes like LXDs [72] and AC [43] provide a way to utilize the caller core by performing a lightweight context switch to another asynchronous thread. Unfortunately, exploiting asynchrony is hard: kernel queues are often short and overheads of creating and joining asynchronous threads add up quickly. Overall, cross-core isolation achieves acceptable overhead (12-18% overheads for an isolated 10Gbps network drivers) but at the cost of additional cores [72].

Memory Protection Keys (MPKs) Recent Intel CPUs introduced memory protection keys (MPKs) to provide fine-grained isolation within a single address space by tagging each page with a 4-bit protection key in the page table entry. A special register, `pkru`, holds the current protection key. The read or write access to a page is allowed only if the value of the `pkru` register matches the tag of the page. Crossing between protection domains is performed by writing a new tag value into the `pkru`, which is a fast operation taking 20-26 cycles [45, 74]. However, several challenges complicate the use of MPKs for isolating kernel code. First, the 4-bit protection keys are interpreted by the CPU only for user-accessible pages, i.e., the page table entries with the "user" bit set. It is possible to map the kernel pages of an isolated subsystem as user-accessible, but additional measures have to be taken to protect the isolated kernel code from user accesses through either a page table switch [40] (which is expensive), or MPKs themselves. The reliance on MPKs requires either binary rewriting of all user applications [87] (which in general is undecidable [45]), or dynamic validation of all `wrpkru` instructions with hardware breakpoints [45] (which can also accumulate significant overhead). Second, the isolation of the kernel code requires careful handling of privileged instructions, e.g., updates of control and segment registers, etc. In turn, this requires either exiting into privilege level 3 (which can be done with an overhead of a system call), through compile-time or load-time binary rewriting of all privileged instructions (which becomes challenging in light of possible control-flow attacks), or executing the system as a non-root VT-x guest, which requires techniques developed in this work.

Extended Page Table switching with VM functions The EPTP switching via the `vmfunc` instruction is yet another new hardware mechanism appearing in Intel CPUs that enables a virtual machine guest to change the root of the extended page table (EPT) by re-loading it with one of a set of values preconfigured by the hypervisor. VMFUNC allows the guest to change EPT mappings, and hence change the boundaries of a protection domain, with a single instruction that takes 109-147 cycles [45, 56, 68]. Compared with MPKs, EPT switching does not require exits into ring 3, binary rewriting, or validation of VMFUNC instructions for isolation of privileged kernel code—all sensitive state can be protected by the hypervisor through construction of non-overlapping address spaces [62] (we describe details of isolating privileged ring 0 code in Section 3.3). Several projects explore the use of VMFUNC for isolation of user programs [45, 63], addressing speculative execution attacks [51], and implementing fast microkernel IPC [68]. LVDs extend VMFUNC-based solutions with support for isolation of privileged kernel code and isolation invariants in the face of fast exitless interrupts.

Software fault isolation (SFI) and MPX Software fault isolation (SFI) allows enforcing isolation boundaries in software without relying on hardware protection [91]. XFI [26], BGI [14], and LXFI [64] apply SFI for isolation of kernel modules in Windows [14, 26] and Linux [64] kernels. LXFI [64] saturates a 1Gbps network adapter for TCP connections, but results in a 2.2-3.7x higher CPU utilization (UDP throughput drops by 30%). On modern network and storage interfaces increase in CPU utilization will likely result in a proportional drop in performance. Recent implementations of SFI, e.g., MemSentry [56] rely on Intel Memory Protection Extensions (MPX)—a set of architectural extensions that provide support for bounds checking in hardware—to accelerate bounds checks. Nevertheless, the overhead of MPX-based SFI remains high: on a CPU-bound workload, the NGINX experiences a 30% slowdown [87]. Moreover, additional control

flow enforcement mechanisms [65, 81, 94] are required to secure SFI in the face of control-flow attacks (such mechanisms will result in additional overhead).

2.2 Complexity of decomposition

Clean slate designs Representing one side of the isolation spectrum, microkernel projects develop kernel subsystems and device drivers from scratch [8–10, 27, 30, 37, 46–48, 52, 53, 61]. Engineered to run in isolation, microkernel drivers synchronize their state via explicit messages or cross-domain invocations. To assist isolated development, microkernels typically rely on interface definition languages (IDLs) [23, 38, 42] to generate caller and callee stubs and message dispatch loops. Unfortunately, clean slate device driver development requires a large engineering effort that also negates decades of work aimed at improving reliability and security of the kernel code.

Device driver frameworks and VMs More practical strategies for isolating parts of the kernel are device driver frameworks and virtualized environments that provide a backward compatible execution environment for the isolated code [6, 13, 22, 24, 29, 35, 36, 44, 55, 76, 85, 89, 92]. While requiring less effort compared to re-writing device drivers from scratch, development of a backward compatible driver execution environment is still a large effort. Outside of several self-contained device driver frameworks, e.g., IOKit in MacOS [59], device drivers rely on a broad collection of kernel functions that range from memory allocation to specialized subsystem-specific helpers.

Alternatively, an unmodified device driver can execute inside a complete copy of the kernel running on top of a virtual machine monitor [11, 12, 31, 33, 58, 73, 79]. Unfortunately, a virtualized kernel extends the driver execution environment with multiple software layers, e.g., interrupt handling, thread scheduling, context-switching, memory management, etc. These layers introduce overheads of tens of thousands of cycles on the critical data-path of the isolated driver, and provide a large attack surface.

Backward compatible code isolation SawMill [34] was probably the first system aiming at development of in-kernel isolation mechanisms for isolation of unmodified kernel code. SawMill relied on the Flick IDL [23] for communication with isolated subsystems, hence, isolation required re-implementation of all interfaces. Nooks developed a framework for isolating Linux kernel device drivers into separate protection domains [84]. Nooks maintained and synchronized private copies of kernel objects between the kernel and the isolated driver, however, the synchronization code had to be developed manually. Nooks' successors, Decaf [77] and Microdrivers [32] developed static analysis techniques [77] to generate synchronization glue code directly from the kernel source. Wahbe et al. [91] and later XFI [26] and BGI [14] relied on SFI to isolate kernel extensions but were not capable of handling semantically-rich boundary between the isolated subsystem and the kernel. LXFI [64] extended previous SFI approaches with support for explicit, fine-grained policies that control access to all data structures shared between the isolated subsystem and the kernel. Conceptually, LXFI's policies serve the same goal as projections in LXDs [72] (that we use in this work)—they are designed to control the access of an isolated subsystem to a specific subset of kernel objects.

3 LVDs Architecture

LVDs are designed to block an adversary who discovers an exploitable vulnerability in one of the kernel subsystems from attacking the rest of the kernel, i.e., gaining additional privileges by reading kernel data structures or code, hijacking control flow, or overwriting sensitive kernel objects.

Similar to prior work, LXFI [64], LVDs aim to enforce 1) *data structure safety*, i.e., the isolated driver can only read and write a well-defined subset of objects and their fields that are required for the driver to function (effectively we enforce *least privilege* [80]), 2) *data structure integrity* [64], i.e., the isolated driver cannot change pointers used by the kernel or the types of objects referenced by those pointers, and 3) *function call integrity* [64], i.e., the isolated code a) can only invoke a well-defined subset of kernel functions and pass legitimate pointers to objects they "own" as arguments, and b) cannot trick the kernel into invocation of an unsafe function pointer registered as part of the driver interface.

Ensuring that an isolation mechanism achieves these goals for complex kernel subsystems like device drivers is challenging. Despite many advances in the modularity of modern kernels, device drivers interact with the kernel through a web of functions and data structures. While the device drivers themselves expose a well-defined interface to the kernel—a collection of function pointers that implement the driver's interface—the driver itself relies on thousands of helper functions exported by the kernel (a typical driver imports over 110-200 functions) that often have deep call graphs.

Threat model We assume a powerful adversary that has full control over the isolated subsystem (its memory, CPU register state, and control flow). Specifically, an attacker can make up cross-domain invocations with any arguments, attempt to read and write CPU registers, try accessing hardware interface, and trigger interrupts. We trust that attacks will not originate from the kernel domain. While LVDs can detect denial of service attacks, we leave efficient handling of driver restart for future work. Also, while LVDs provide a least-privileged isolation boundary and block trivial Iago-style attacks [15], e.g., an isolated driver cannot return a rogue pointer to the kernel, we leave complete analysis of the feasibility to construct an arbitrary computation (e.g., to overwrite sensitive kernel data structures like page tables) in the kernel by modifying shared objects, passing or returning values to and from cross-domain invocations, etc., as future

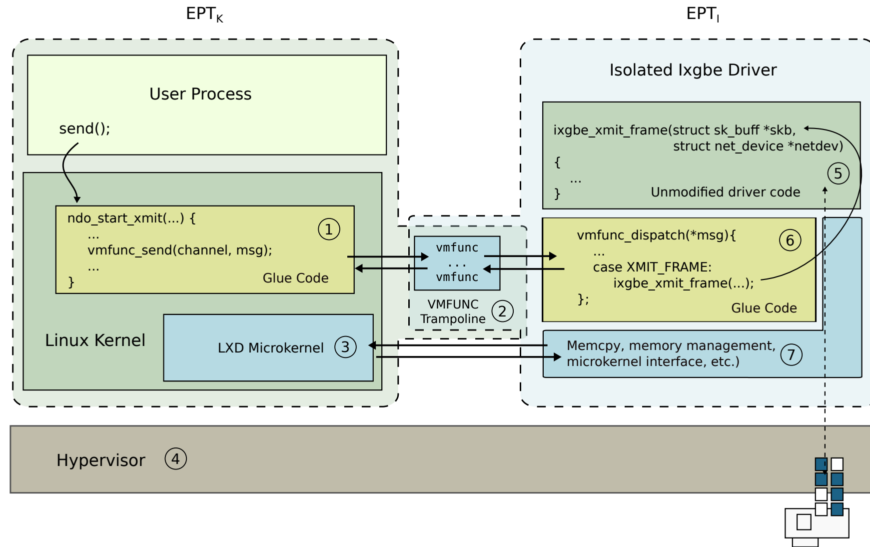

Figure 1. LVDs architecture.

work. Finally, speculative execution and side channel attacks are out of scope of this work as well.

3.1 Overview of the LVDs Architecture

LVDs utilize hardware-assisted virtualization for isolation and control of privileged instructions inside isolated domains (Section 4). We execute the system under control of a minimal late-launch hypervisor that transparently demotes the system into a non-root VT-x guest right before it loads the first isolated subsystem (Figure 1, ④). Specifically, we leverage a modified version of the Bareflank [1] hypervisor that is loaded as a kernel module that pushes the system into a VT-x non-root mode by creating a virtual-machine control structure (VMCS) and a hierarchy of per-CPU extended page tables. The hypervisor remains transparent to the monolithic kernel, i.e., all exceptions and interrupts are delivered directly to the demoted kernel through the original kernel interrupt descriptor table (IDT). The demoted kernel can access entire physical memory and I/O regions via the one-to-one mappings in EPT.

LVDs run as a collection of isolated domains managed by a small kernel module that exposes an interface of a microkernel to the isolated domains (Figure 1, ③). When a new isolated driver is created, the microkernel module creates a new EPT (EPT_I) that maps physical addresses of the driver domain. Upon cross-domain invocation the VMFUNC instruction switches between EPT_K and EPT_I (we discuss details of our implementation below in Section 3.3).

3.2 Device Driver Isolation

Isolation of kernel code requires analyzing all driver dependencies, deciding the cut between the driver and the kernel, and providing mechanisms for cross-domain calls and secure synchronization of data structures that are no longer shared between the isolated subsystems. LVDs rely on the LXDs decomposition framework [72] that includes an interface definition language (IDL) for specifying the interface between kernel modules and generating code for synchronizing the hierarchies of data structures across isolated subsystems.

In LXDs, isolated subsystems do not share any state that might break isolation guarantees, e.g., pointers, indexes into memory buffers, etc. Each isolated subsystem maintains a private copy of each kernel object. To support synchronization of object hierarchies across domains, the IDL provides the mechanism of *projections* that describe how objects are marshaled across domains. A projection explicitly defines a subset of fields of a data structure that is synchronized upon domain invocation (hence, defining how a data structure is projected into another domain).

Definitions of cross-domain invocations can take projections as arguments. Passed as an argument, a projection grants another domain a right to access a specific object, i.e., synchronize a subset of object's fields described by the projection. LXDs rely on the idea of *capabilities* that is similar to object capability languages [67, 69], where capabilities are unforgeable cross-domain object references. The IDL generates the code to reflect the capability "grant" operation by inserting an entry in a capability address space, CSpace, the data structure that links capabilities to actual data structures. The capability itself is an opaque number that has no meaning outside of a specific CSpace. Projections, therefore, define the minimal set of objects and their fields accessible to another domain. As projections may define pointers to other projections, LXDs provide a way to synchronize hierarchies of objects. Finally, the IDL provides a way to define remote

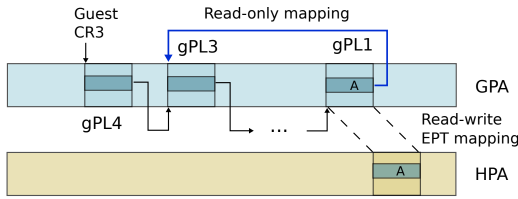

Figure 2. Enforcement of read-only access for guest pagetable pages. The page table entry in gPL1 maps one of the pages of the page table (gPL3) as read-only.

procedure calls specifying all functions accessible across the isolation boundary.

LXDs provide a backward-compatible execution environment capable of executing unmodified device drivers (Figure 1, ⑤). Inside the isolated driver, LXDs provide: 1) the glue code generated by the IDL compiler that implements marshaling and synchronization of objects (Figure 1, ⑥), and 2) a minimal library that provides common utility functions compatible with the non-isolated kernel (Figure 1, ⑦), i.e., memory management, common utilities like memcpy(), and a collection of functions to interface with the microkernel, e.g., capability management, debugging, etc. To ensure that the legacy, non-decomposed kernel can communicate with an isolated driver, a layer of synchronization glue code is used on the kernel side (Figure 1, ①).

3.3 Lightweight Isolation with VMFUNC

VMFUNC is a machine instruction available in recent Intel CPUs that allows a non-root VT-x guest to switch the root of the extended page table (EPT) to one of a set of preconfigured EPT pointers, thus changing the entire view of accessible memory. To enable EPT switching, the hypervisor configures a table of possible EPT pointers. A non-root guest can freely invoke VMFUNC at any privilege level and select the active EPT by choosing it from the EPT table. Immediately after the switch, all guest physical addresses (GPAs) are translated to host-physical addresses (HPAs) through the new EPT. As EPTs support TLB tagging (virtual processor identifiers (VPIDs)), the VMFUNC instruction is fast (Section 5).

Isolation with EPTs Lightweight EPT switching allows for a conceptually simple isolation approach. We create two EPTs that map disjoint subsets of machine pages isolating the address spaces of mistrusting domains. To switch between the address spaces, a call-gate page with the VMFUNC instruction is mapped in both EPTs. This straightforward approach however requires a range of careful design decisions to ensure security of the isolation boundary.

4 Enforcing Isolation

In contrast to traditional privilege transition mechanisms, e.g., interrupts, system call instructions, and VT-x entry/exit transitions, VMFUNC provides no support for entering an isolated domain at a predefined entry point. The next instruction after the VMFUNC executes with the memory rights of another domain. The cross-domain invocation mechanisms, however, must ensure that transition is safe, i.e., all possible VMFUNC invocations lead to a set of well defined entry points in the kernel, and the kernel can securely restore its state.

Safety of the VMFUNC instructions By subverting the control flow inside an isolated domain, an attacker can potentially find executable byte sequences that form a valid VMFUNC instruction. If the virtual address next after the VMFUNC instruction is mapped in the address space of another domain, an attacker can escape the isolation boundary.

Two possible approaches to prevent such attacks are to: 1) ensure that virtual address spaces across isolated domains are not overlapping [62], or 2) ensure that no sequences of executable bytes can form a valid VMFUNC instruction [56, 68, 90]. Inspired by ERIM [87], SkyBridge [68] relies on scanning and rewriting executable space of the program to ensure that no byte sequences form valid VMFUNC instructions. In the case of LVDs, the attack surface for preventing unsafe VMFUNC instructions expands into user applications, i.e., any user program in the system can invoke a VMFUNC instruction triggering a switch into the isolated device driver. In the face of dynamic code compilation, program rewriting [87] requires a large TCB with a large attack surface. We therefore choose the memory isolation approach similar to SeCage [62]. Specifically, we enforce the following invariant:

> *Inv 1.* Virtual address spaces of isolated domains, kernel, and user processes do not overlap.

This invariant ensures that if an isolated domain or a user process invokes a self-prepared VMFUNC instruction anywhere in its address space, the next instruction after the VMFUNC causes a page fault.

Locking the LVD's address space To maintain *Inv 1*, we have to ensure that isolated domains cannot modify the layout of their address space, or specifically:

> *Inv 2.* Isolated domains have read-only access to their page table.

This is challenging: isolated subsystems run in ring 0 and have privileges to change their page tables. It is possible to map all pages of the page table as read-only in the EPT of the isolated domain. While this ensures that domain's page table hierarchy cannot be modified, it also causes a prohibitive number of VT-x exits when the CPU tries to update the dirty and accessed bits in the page table of the isolated driver [51]. We therefore, employ a technique similar to EPTI [51] and map all the physical pages of the page table as read-only in the leaf entries of the guest page table (Figure 2). I.e., all virtual addresses that point into the pages of the page table have only the read permission. At the same time, the

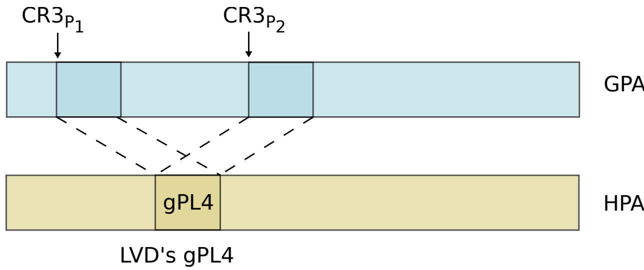

Figure 3. Cr3 remapping inside an LVD. The HPA page that contains the root of the LVD's page table (gPL4) is mapped at two GPA locations making it possible for two processes P_1 and P_2 to enter the LVD.

pages that contain the page itself are mapped with write permissions in the EPT_I. This way the CPU can access pages of the page table and update accessed and dirty bits without causing an exit.

The following design allows us to avoid modifications to the read-only page table inside the LVD. We create a large virtual address space when the LVD starts, i.e., create a page table that maps guest virtual pages to guest physical. The physical pages are not backed up by real host physical pages. We then never update the LVD's page table. Instead we allocate host physical pages and update the EPT mappings to map these pages into guest physical addresses already mapped by the read-only page table.

CR3 remapping for EPT switch While by itself the VMFUNC instruction does not change the root of the page table hierarchy, i.e., the CR3 register on x86 CPUs, the ability to switch EPTs, i.e., the GPA to HPA mappings, opens the possibility to change the guest page table too. The advantage of such design is the ability to execute non-isolated kernel and isolated drivers on independent virtual address spaces and page table hierarchies. Since individual processes and kernel threads execute on different page tables, we need to ensure that for each new process that tries to enter an LVD the physical address of the process' root of the page table, i.e., the physical address pointed by the CR3, is mapped to the HPA page that contains the root of the page table of the LVD's address space (Figure 3). We enforce the following invariant:

> *Inv 3.* Physical address spaces of isolated domains and the kernel must not overlap.

This guarantees that the physical address that is used for the root of the page table inside the kernel is not occupied inside the isolated domain, and hence can be remapped into the HPA page that contains the root of the page table inside the isolated domain.

Protecting sensitive state Isolated subsystems execute with ring 0 privileges. Hence they can read and alter sensitive hardware registers, e.g., re-load the root of the page table hierarchy by changing the cr3 register. To ensure isolation, we enforce the following invariant:

> *Inv 4.* Access to sensitive state is mediated by the hypervisor.

To implement *Inv 4*, we configure the guest VM to exit into the hypervisor on the following instructions that access the sensitive state: 1) stores to control registers (cr0, cr3, cr4), 2) stores to extended control register (xcr0), 3) reads and writes of model specific registers (MSRs), 4) reads and writes of I/O ports, 5) access to debug registers, and 6) loads and stores of GDT, LDT, IDT, and TR registers. Inside the hypervisor we validate if the exit happens from the non-isolated kernel and emulate the exit-causing instruction.

Restoring kernel state When the execution re-enters the kernel from the isolated domain, e.g., either returning form the domain invocation, or entering the kernel with a call from an isolated subsystem, the kernel cannot trust any of the general, segment, or floating-point registers.

> *Inv 5.* General, segment, and extended state (x87 FPU, SSE, AVX, etc.), registers are saved and restored on domain crossings.

As we cannot trust any general registers, upon entering the kernel we restore the kernel's stack pointer from a trusted location in memory and then use the stack to restore other registers. We rely on the fact that the isolated driver cannot modify kernel's address space (ensured by *Inv 2* and *Inv 4*). We create a special page, vmfunc_state_page, which stores the pointer to the kernel stack of the current thread right before it enters the LVD. The entry-exit trampoline code uses the stack to save and restore the state of the thread.

LVDs are multi-threaded and re-entrant. While we do not allow context switches inside LVDs, the same LVD can run simultaneously on multiple CPUs. We therefore create a private copy of vmfunc_state_page on each CPU. Linux uses the gs register to implement per-CPU data-structures (on each CPU gs specifies a different base for the segment that stores local CPU variables). As we cannot trust the value of gs on entry into the kernel from an LVD, we create a per-CPU mapping for the vmfunc_state_page in EPT_K. This ensures that on different CPUs, the vmfunc_state_page is mapped by a different machine page and hence holds local CPU state.

Stacks and multi-threading Any thread in the system can enter an isolated domain either as part of the system call that invokes a function of an isolated subsystem, or as part of an interrupt handler implemented inside an LVD. Every time the thread enters an isolated domain we allocate a new stack for execution of the thread inside the LVD. We use a lock-free allocator that relies on a per-CPU pool of pre-allocated stacks inside each LVD. From inside the LVD the thread can invoke a kernel function that in turn can re-enter the isolated domain. To prevent allocation of a new stack,

we maintain a counter to count nested invocations of the isolated subsystem.

4.1 Exitless Interrupt Handling

Historically, lack of hardware support for fine-grained assignment of interrupts across VMs and hypervisor required multiple exits into the hypervisor on the interrupt path [39, 86]. ELI [39] and DID [86] developed mechanisms for exitless delivery of interrupts for hardware-assisted VMs. We develop an exitless scheme that allows LVDs to handle interrupts even when execution is preempted inside an isolated domain.

At a high level, LVDs allow delivery of interrupts through the interrupt descriptor table (IDT) of the non-isolated kernel. The IDT is mapped inside both kernel and isolated domains. When interrupt is delivered we switch back to the kernel EPT early in the interrupt handler. To ensure that interrupt delivery is possible, we map the IDT, global descriptor table (GDT), task-state segment (TSS), and interrupt handler trampolines on both EPT_K and EPT_I. Upon interrupt transition the hardware takes the normal interrupt delivery path, i.e., saves the state of the currently executing thread on the stack, locates the interrupt handler through the IDT, and jumps to it. The interrupt handler trampoline checks if the execution is still inside the LVD, and performs a VMFUNC transition back to the kernel if it's required.

While conceptually simple, the exitless interrupt delivery scheme requires careful design in the face of possible isolation attacks.

Interrupt Stack Table (IST) Both non-isolated kernel and LVDs execute with the privileges of ring 0. As privilege level does not change during the interrupt transition, the traditional interrupt path does not require change of the stack, i.e., the hardware saves the trap frame on the stack pointed by the current stack pointer. This opens a possibility for a straightforward attack: an LVD can configure the stack to point to a writable kernel memory in the kernel domain, and perform a VMFUNC transition back into the kernel through one of the trampoline pages. VMFUNC is a long-running instruction, and often interrupts are delivered right after the VMFUNC instruction completes[1]. The interrupt will be delivered inside the kernel domain and hence will overwrite the kernel memory pointed by the stack pointer register configured by the isolated domain.

To prevent this attack, and to make sure that an interrupt is always executed on a valid stack, we rely on Interrupt Stack Table (IST) [5]. The IST allows one to configure the interrupt handler to always switch to a preconfigured new stack even if the privilege level remains unchanged. Each IDT entry has 8 bits to specify one of the seven available IST stacks. Linux already uses ISTs for NMIs, double-fault, debug, and machine-check exceptions.

[1]We empirically confirmed this with perf, a profiler tool that relies on frequent interrupts from the hardware performance counter interface.

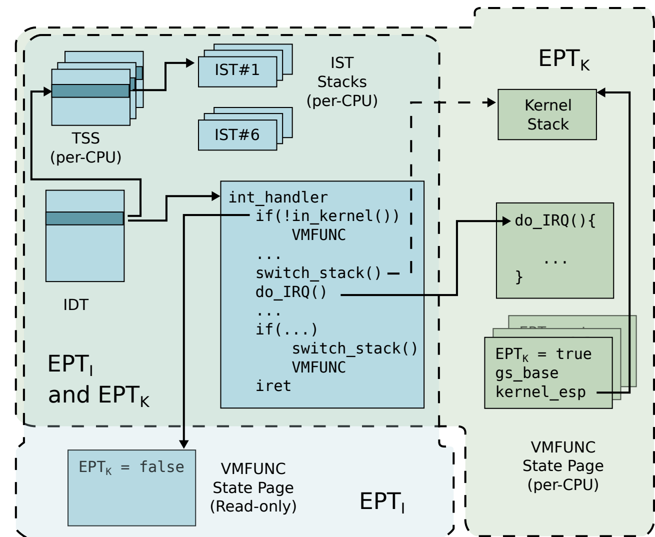

Figure 4. Data-structures involved in interrupt transition

To protect the kernel from a rogue stack interrupt attack, we configure two additional IST stacks for execution of synchronous exceptions and asynchronous interrupts (Figure 4). Upon an interrupt the hardware switches to a fresh IST stack. We use the IST stack inside a small interrupt handler trampoline that is mapped in both kernel and isolated domains. The trampoline checks whether the system is running inside the kernel or in one of the isolated domains. It switches to EPT_K if needed, securely restores the system's state by using the information from the vmfunc_state_page page (we restore the gs register used by the kernel to maintain per-CPU data structures, and the stack pointer register that points to the kernel stack). After that we copy the saved interrupt frame to the normal kernel stack and continue execution of the interrupt handler through the normal kernel path. Note that the kernel can re-enable interrupts at this point, as the IST stack is no longer used for the current interrupt. Upon exit we check whether the switch back to LVD is required. If yes the handler copies the exception frame back to the IST stack (since only the IST stack is mapped inside the LVD), switches back to EPT_I, and returns from the interrupt with the regular iret instruction.

We configure all interrupt handlers to disable subsequent interrupts upon interrupt transition—this ensures that IST stack will not be overwritten until we copy out the interrupt frame onto the normal kernel stack. Anytime during processing of the interrupt a non-maskable interrupt (NMI) can be delivered. We configure a separate IST stack for the NMI to prevent overwriting the state of the previous interrupt frame on the IST.

To reliably detect whether the interrupt handler is running inside the kernel or inside an LVD we rely on the vmfunc_state_page that is mapped by both EPT_K and EPT_I. Inside the kernel the state page has a flag set to true. This flag is false in the page mapped by EPT_I.

4.2 VMFUNC Isolation Attacks and Defenses

Due to its unusual semantics, VMFUNC opens possibility for a series of non-traditional attacks that we discuss below.

Rogue VMFUNC transitions A compromised LVD can use one of the available VMFUNC instruction instances to perform a rogue transition into the kernel, e.g., try to enter the kernel via the kernel exit trampoline. We insert a check right after each VMFUNC instruction to see that ECX register that is used as an index to choose the active EPT is set to the correct value, i.e., zero or one based on the direction. If we detect a violation we abort execution by exiting into the hypervisor that upcalls into the kernel triggering termination of the LVD.

Asynchronous write into the IST stack from another CPU An LVD can try to write into the IST stack of another core that executes an interrupt handler at the same time, hence crashing or confusing the kernel. We ensure that all IST stacks are private to a core, i.e., are allocated for each core and are mapped only on the core that uses them.

Interrupt injection attack As LVDs run in ring 0, they can invoke INT instruction injecting an interrupt into the kernel. We disable synchronous interrupts 0-31 originating from isolated domains, i.e., in the interrupt handler we check if the handler is executing on EPT_I and if so terminate the LVD. Note, legit asynchronous interrupts can preempt the LVD and hence it is impossible to say whether interrupt injection is happening without inspecting the instruction that was executed right when interrupt happened. While we do not implement this defense at the moment, we suggest a periodic inspection of the LVD's instruction if the frequency of a specific interrupt exceeds an expected threshold.

Interrupt IPI from another core running LVD An LVD running on another core can try to inject an inter-processor interrupt (IPI) implementing a flavor of interrupt injection attack. We protect against this attack by making sure that the APIC I/O pages used to send inter-processor interrupts are not mapped inside LVDs.

VMFUNC to a non-existent EPT entry An LVD can try to VMFUNC into a non-existent EPT list entry. We configure the EPTP list page to have an invalid pointer to make sure that such transition causes an exit into the hypervisor. The hypervisor then delivers an upcall exception to the kernel which terminates the LVD.

VMFUNC from one LVD into another One LVD can try to VMFUNC into another LVD. We configure the EPTP list page to have only two entries at every moment of time: an EPT of the kernel (entry zero), and EPT of the LVD that is about to be invoked. Since, EPTP list is mapped inside the kernel, we re-load the first entry when the kernel is about to invoke a specific LVD.

Kernel stack exhaustion attack An LVD might try to arrange a *valid* control flow to force the kernel to call the LVD over and over again until the kernel stack is exhausted. To prevent this loop we check that the kernel stack is above the threshold every time we enter the LVD.

LVD never exits LVDs can disable interrupts by clearing the interrupt flag in the EFLAGS register. An LVD then never returns control to the kernel. We configure a non-conditional VM preemption timer that passes control to the hypervisor periodically. The hypervisor checks if the kernel is making progress by checking an entry in the vmfunc_state_page entry.

LVD re-enables interrupts An LVDs can re-enable interrupts by setting the interrupt flag in the EFLAGS register. The kernel might then receive an interrupt in an interrupt-disabled state and as a result crash or corrupt sensitive kernel state. We make a practical assumption that under normal conditions the isolated subsystem should not re-enable interrupts. We save the interrupt flag when we enter into an LVD and check the state of the flag in the interrupt handler. If the interrupt originates while inside the LVD and the interrupts were disabled before entering the isolated domain we signal the attack. We also check the state of the saved interrupt flag every time we exit from LVD, and signal the attack if it does not match the saved value.

5 Evaluation

We conduct all experiments in the openly-available CloudLab cloud infrastructure testbed [18] (we make all experiments available via an open CloudLab [78] profile[2] that automatically instantiates software environment used in this section).[3] Our experiments utilize two CloudLab c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.60 GHz, 160 GB RAM, and a dual-port Intel X520 10Gb NIC. All machines run 64-bit Ubuntu 18.04 Linux with kernel version 4.8.4. In all experiments we disable hyper-threading, turbo boost, and frequency scaling to reduce the variance in benchmarking.

5.1 VMFUNC Domain-Crossings

To understand the overheads of the VMFUNC-based isolation we conduct a series of experiments aimed at measuring overheads of VMFUNC instructions, and VMFUNC-based cross-domain invocations (Table 1). In all tests we run 10 million iterations and measure the latency in cycles with the RDTSC and RDTSCP instructions. Further, to avoid flushing the cached EPT translations, we enable support for virtual processor identifiers (VPIDs). On our hardware, a single invocation of the VMFUNC instruction takes 169 cycles. To put this number in perspective, we measure the overhead of a null system call to be 140 cycles.

[2]CloudLab profile is available at https://www.cloudlab.us/p/lvds/lvd-linux
[3]LVDs source code is available at https://mars-research.github.io/lvds

Operation	Cycles
VMFUNC instruction	169
System call	140
seL4's call/reply invocation	834
VMFUNC-based call/reply invocation	396

Table 1. Cost of VMFUNC-based cross-domain invocations.

Operation	Cycles	
	Native	Virtualized
write MSR	127	1367
out instruction	4213	5384
write cr3	130	143

Table 2. Cost of hypervisor exits.

To understand the benefits of VMFUNC-based cross-domain invocations over traditional page-based address-space switches, we compare LVDs' cross-domain calls with the synchronous IPC mechanism implemented by the seL4 microkernel [25]. We choose seL4 as it implements the fastest synchronous IPC across several modern microkernels [68]. To defend against Meltdown attacks, seL4 provides support for a page-table-based kernel isolation mechanism similar to KPTI [41]. However, this mechanism negatively affects IPC performance due to an additional reload of the page table root pointer. Since recent Intel CPUs address Meltdown attacks in hardware, we configure seL4 without these mitigations. With tagged TLBs seL4's call/reply IPC takes 834 cycles (Table 1). LVDs's VMFUNC-based call/reply invocation requires 396 cycles.

5.1.1 Overheads of Running Under a Hypervisor

LVDs execute the system under control of a hypervisor resulting in two kinds of overheads: 1) overheads due to virtualization, i.e., EPT translation layer, and 2) exits to the hypervisor caused by the need to protect sensitive instructions that can potentially break isolation if executed by an LVD.

Sensitive instructions We first conduct a collection of experiments aimed at measuring the cost of individual VM-exits that are required to mediate execution of sensitive instructions (Table 2). On average an exit into the hypervisor takes 1171-1240 cycles. To reduce the number of exits due to updates of the cr3 register, we implement an LRU cache and maintain the list of three target cr3 values.

Whole-system benchmarks We evaluate the impact of virtualization and LVD-specific kernel modifications by running a collection of Phoronix benchmarks [75]. The Phoronix suite provides a large library of whole-system benchmarks; we use a set of benchmarks that characterize both whole-system performance and stress specific subsystems. The whole-system benchmarks include apache (measures sustained requests/second; 100 concurrent requests); nginx (measures sustained requests/second; 500 concurrent requests); pybench (tests basic, low-level functions of Python); phpbench (large numbers of simple tests against the PHP interpreter). Subsystem-specific benchmarks include dbench (file system calls to test disk performance, varying the number of clients); postmark (transactions on 500 small files (5–512 KB) simultaneously); povray (3D ray tracing); sysbench (performs CPU and memory tests); gnupg (encryption time with GnuPG).

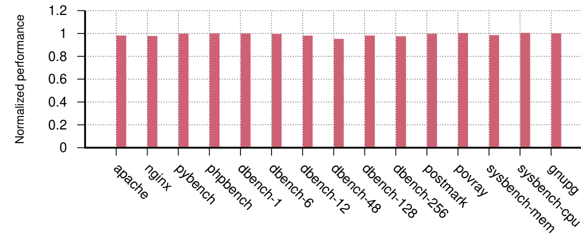

Figure 5. Phoronix benchmarks.

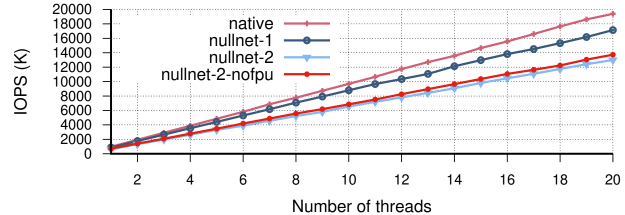

Figure 6. Null net Tx IOPS (K)

Figure 5 shows the performance of the virtualized LVD kernel relative to the performance of an unmodified Linux running on bare metal. Apache, nginx, dbench (with more than 12 clients), and sysbench incur 2–5% slowdown relative to the unmodified Linux system. All other benchmarks stay within 1% of the performance of the bare-metal system.

Breakdown of VM exits To better understand the reasons of possible performance degradation due to virtualization, we collect the number and the nature of VM-exits for Phoronix benchmarks. In our tests Phoronix benchmarks experience from 465 (phpbench) to 155862 (nginx) VM-exits per second. Two most frequent exit reasons are 1) access to MSRs required for programming the APIC timer (IA32_TSC_DEADLINE) and updating the base address of the FS segment during a context switch (IA32_FS_BASE), and 2) access to control registers.

5.2 Overheads of isolation

To evaluate the overheads of isolation we developed several isolated device drivers in the Linux kernel. Specifically, we developed isolated versions of 1) a software-only "null" network driver (nullnet), 2) an Intel 82599 10Gbps Ethernet driver (ixgbe), and 3) a software-only "null" block NVMe driver (nullblock). Neither null net nor null block are connected to a real hardware device. Instead they emulate infinitely fast devices in software. The software-only drivers allow us to stress overheads of isolation without any artificial hardware limits. Both network and storage layers are kernel subsystems with the tightest performance budgets; hence we choose them for evaluating overheads of our isolation mechanisms.

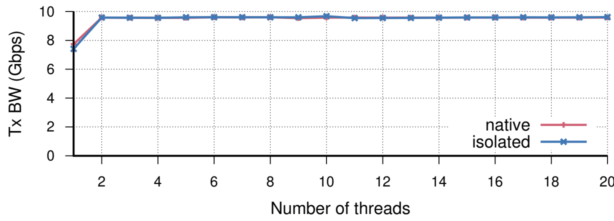

Figure 7. Ixgbe Tx Bandwidth (Gbps)

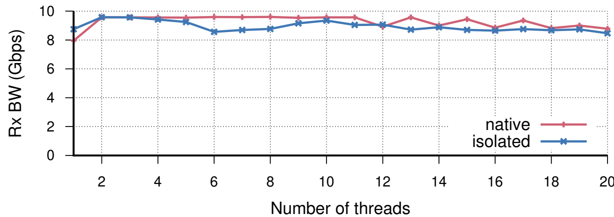

Figure 8. Ixgbe Rx Bandwidth (Gbps)

5.2.1 Nullnet Device Driver

We utilize the nullnet driver as a use-case for several benchmarks that highlight overheads of isolation in the context of a "fast" device driver (nullnet is a good, representative example of such device driver as it serves an infinitely fast device and is accessed through a well-optimized I/O submission path of the kernel network stack). To evaluate the isolated nullnet and ixgbe drivers, we use the iperf2 benchmark that measures the transmit bandwidth for the MTU sized packets and by varying the number of threads from 1 to 20 (Figure 6). We report total packet transmission I/O requests per-second (IOPS) across all CPUs (Figure 6).

In our first experiment we change nullnet to perform only *one* crossing between the kernel and the driver for sending each packet (nullnet−1, Figure 6). This synthetic configuration allows us to analyze overheads of isolation in the ideal scenario of a device driver that requires only one crossing on the device I/O path. With one application thread the non-isolated driver achieves 968K IOPS (i.e., on average, a well-optimized network send path takes only 2680 cycles to submit an MTU-sized packet from the user process to the network interface). The isolated driver (nullnet−1) achieves 876K IOPS (91% of the non-isolated performance), and on average requires 2960 cycles to submit one packet. Since, iperf application uses floating point operations, we incur additional overhead due to saving and restoring of FPU regs when we jump between the kernel and isolated domain. On 20 threads the isolated driver achieves 91% of the performance of the non-isolated driver.

In our second experiment, we run the isolated nullnet driver in its default configuration (i.e., perform two domain crossings per packet transmission). In a single threaded test, the isolated driver achieves 65% performance of the non-isolated driver.

Finally, we measure the overhead of saving and restoring the processor extended state, i.e., floating-point, SSE, and AVX registers. Not all programs use extended state registers and thus can benefit from faster domain crossings. The Linux kernel dynamically tracks if extended state was used, hence we save and restore it only when needed. We disable saving and restoring extended state for the iperf nullnet benchmarks ((nullnet−2−nofpu, Figure 6). Without extended state on a single core the isolated driver achieves 72% performance of the non-isolated driver.

5.2.2 Ixgbe Device Driver

To measure performance of the isolated ixgbe driver, we configure an iperf2 test with a varying number of iperf threads ranging from 1 to 20 (Figure 7). On our system, even two application threads saturate a 10Gbps network adapter. Configured with one iperf thread, on the transmission path using an MTU sized packet, the isolated ixgbe achieves 95.7% of the performance of the non-isolated driver. This difference disappears as we add more iperf clients. For two or more threads, both drivers saturate the network interface and show a nearly identicall throughput. On the receive path, the isolated driver is 10% faster for one application thread. With a higher number of application threads, the isolated driver is within 1% to 11% of the performance of the native driver (Figure 8).

To measure the end-to-end latency, we rely on the UDP request-response test implemented by the netperf benchmarking tool. The UDP_RR measures the number of round-trip request-response transactions per second, i.e., the client sends a 64 byte UDP packet and waits for the response from the server. The native driver achieves 29633 transactions per second (which equals the round-trip latency of 33.7μs), the isolated driver is 8% (1.2μs) slower with 27333 transactions per second (round-trip latency of 36.5μs).

5.2.3 Multi-queue block device driver

In our block device experiments, we use the fio benchmark to generate I/O requests. To set an optimal baseline for our evaluation, we chose the configuration parameters that can give us the lowest latency path to the driver, so that overheads of isolation are more profound. We use fio's libaio engine to overlap I/O submissions, and bypass the page cache by setting direct I/O flag to ensure raw device performance. We vary the number of fio threads involved in the test from 1 to 20 and use the block size of 512B, and I/O submission queue length of 1 and 16 (Figure 9). We submit a set of requests at once, i.e., either 1 or 16 and also poll for the same number of completions. Since the nullblock driver does not interact with an actual storage medium, reads and writes perform the same, so we use read I/O requests in all experiments.

For 512 byte requests on a single thread and submission queue length of one the isolated driver achieves 337K IOPS compared to the 559K IOPS for the native (Figure 9). The isolated nullblock driver goes through three cross-domain calls on the I/O path and hence it incurs higher overhead compared

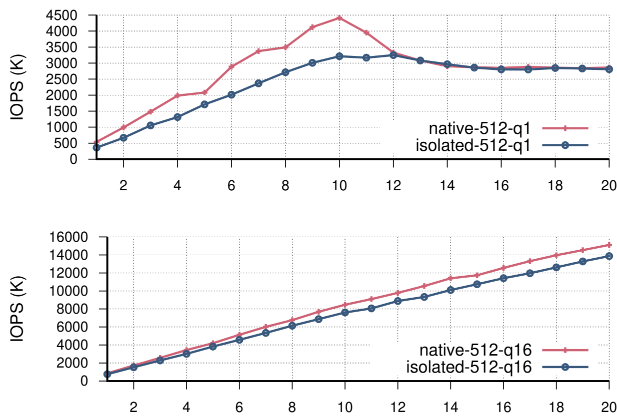

Figure 9. Performance of the nullblock driver

Kernel/hypervisor setup	Cycles
Vanilla kernel	607
Vanilla kernel (virtualized)	620
LVD kernel	666
LVD kernel (virtualized) inside kernel	680
LVD kernel (virtualized, inside isolated domain)	1060

Table 3. Overhead of interrupt delivery.

to nullnet. As the number of threads grows both isolated and native drivers bottleneck on the access to a shared variable involved in collection of I/O statistics. This bottleneck allows isolated driver to catch up with the performance of the native driver when the number of threads is twelve or more (at eleven threads the access to the shared variable crosses the socket boundary and hence becomes more expensive).

In contrast, the gap between the native and isolated driver shortens as the length of the submission queue increases to 16. For 512 byte requests on a single thread with submission queue length of 16, the isolated driver achieves 760K IOPS compared to 837K IOPS for the non-isolated driver. With 20 threads, the isolated driver achieves 9,030K IOPS compared to 15,138K IOPS for the native driver.

5.2.4 Exitless Interrupt Delivery

To understand the overheads of exitless interrupt delivery, we measure latencies introduced on the interrupt path by virtualization, and LVDs' isolation mechanisms; specifically execution on IST stacks, and VMFUNC domain crossings when interrupt is delivered while inside an isolated domain. To eliminate the overheads introduced by general layers of interrupt processing in the Linux kernel, we register a minimal interrupt handler that acknowledges the interrupt right above the machine-specific interrupt processing layer. Our tests invoke the int instruction in the following system configurations which we run on bare metal and on top of the hypervisor (Table 3): 1) in a kernel module loaded inside an unmodified vanilla Linux kernel, 2) inside a kernel module in the LVD kernel, and 3) inside an LVD. In both

Transitions/exits	Experiments		
	nullnet	ixgbe	nullb
VMFUNC	41 x 10^6	27 x 10^6	33 x 10^6
VM-Exit	14074	13235	25789

Table 4. VMFUNC crossings vs number of exits

vanilla and LVD kernels virtualization itself introduces only a minimal overhead to the interrupt processing path (13 and 14 cycles respectively). The LVD kernel executes all interrupts and exceptions on IST stacks, and thus pays additional price of switching to an IST stack while entering the interrupt handler (59-60 cycles). An interrupt inside an isolated domain introduces the overhead of 380 cycles due to two VMFUNC transitions required to exit and re-enter the LVD. In all three tests (nullnet, nullblock, and ixgbe) 11% of interrupts are delivered while inside an LVD.

LVDs trade the cost of changing the privilege level on cross-domain invocations for exits into the hypervisor on execution of privileged instructions. To justify this choice, we measure the number of VM exits and compare it with the number of VMFUNC transitions for I/O intensive workloads (nullnet, nullblock, and ixgbe tests) (Table 4). For the iperf test that stresses performance of the isolated ixgbe driver, we recorded a total of 13235 VM exits, a number that is three orders of magnitude smaller compared to the number of cross-domain transitions (27 x 10^6).

6 Conclusions

Over the last four decades operating systems gravitated towards a monolithic kernel architecture. However, the availability of new low-overhead hardware isolation mechanisms in recent CPUs brings a promise to enable kernels that employ fine-grained isolation of kernel subsystems and device drivers. Our work on LVDs develops new mechanisms for isolation of kernel code. We demonstrate how hardware-assisted virtualization can be used for controlling execution of privileged instructions and define a set of invariants that allows us to isolate kernel subsystems in the face of an intricate execution model of the kernel, e.g., provide isolation of preemptable, concurrent interrupt handlers. While our work utilizes EPTs for memory isolation, we argue that our techniques can be combined with other architectural mechanisms, e.g., MPK, a direction we plan to explore in the future.

Acknowledgments

We would like to thank OSDI 2019, ASPLOS 2019, and VEE 2020 reviewers for numerous insights helping us to improve this work. We are further grateful to the Utah CloudLab team for the patience with accommodating our countless requests and outstanding technical support. This research is supported in part by the National Science Foundation under Grant Number 1840197.

References

[1] Bareflank Hypervisor SDK. http://bareflank.github.io/hypervisor/.

[2] Code-Pointer Integrity in Clang/LLVM. https://github.com/cpi-llvm/compiler-rt.

[3] LKDDb: Linux Kernel Driver DataBase. https://cateee.net/lkddb/. Accessed on 04.23.2019.

[4] seL4 performance. https://sel4.systems/About/Performance/.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2017. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[6] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42. Technical report, Technical report, IBM Watson Research, 2002.

[7] Scott Bauer. Please stop naming vulnerabilities: Exploring 6 previously unknown remote kernel bugs affecting android phones. https://pleasestopnamingvulnerabilities.com, 2017.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 29–44, New York, NY, USA, 2009. ACM.

[9] D. W. Boettner and M. T. Alexander. The Michigan Terminal System. *Proceedings of the IEEE*, 63(6):912–918, June 1975.

[10] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Berkeley, CA, USA, 1992.

[11] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *USENIX ATC*, pages 9–9, 2010.

[12] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[13] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[14] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 45–58, New York, NY, USA, 2009. ACM.

[15] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS XVIII*, pages 253–264. ACM, April 2013.

[16] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10(6):368–379, 2013.

[17] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *APSys*, pages 5:1–5:5, 2011.

[18] CloudLab testbed. http://cloudlab.us/.

[19] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/, 2012.

[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, and Jonathan Walpole. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[21] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[22] DDEKit and DDE for linux. http://os.inf.tu-dresden.de/ddekit/.

[23] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *ACM SIGPLAN Notices*, volume 32, pages 44–56. ACM, 1997.

[24] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 256–269. Springer, 2004.

[25] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[26] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[27] Feske, N. and Helmuth, C. *Design of the Bastei OS architecture.* Techn. Univ., Fakultät Informatik, 2007.

[28] Stephen Fischer. Supervisor mode execution protection. NSA Trusted Computing Conference, 2011.

[29] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, 1997.

[30] Alessandro Forin, David Golub, and Brian N Bershad. An I/O system for Mach 3.0. Carnegie-Mellon University. Department of Computer Science, 1991.

[31] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[32] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.

[33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP*, pages 193–206, 2003.

[34] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. ACM, 2000.

[35] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the USENIX Annual Technical Conference*, pages 65–74, 1996.

[36] David B Golub, Guy G Sotomayor, and Freeman L Rawson III. An architecture for device drivers executing as user-level tasks. In *USENIX MACH III Symposium*, pages 153–172, 1993.

[37] Google. Fuchsia project. https://fuchsia.dev/fuchsia-src/getting_started.md.

[38] Google. Protocol buffers. https://developers.google.com/protocol-buffers/.

[39] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[40] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. Iskios: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654*, 2019.

[41] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[42] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In

Proceedings of the 1st Workshop on Industrial Experiences with Systems Software, San Diego, CA, October 22 2000.

[43] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. AC: composable asynchronous I/O for native languages. In *ACM SIGPLAN Notices*, volume 46, pages 903–920. ACM, 2011.

[44] Hermann Härtig, , Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical report, TU Dresden, Dresden, Germany, 2003.

[45] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[46] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[47] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[48] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.

[49] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 298–307, 2004.

[50] Tomas Hruby, Herbert Bos, and Andrew S Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 255–266, 2013.

[51] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: Efficient defence against meltdown attack for unpatched vms. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 255–266, 2018.

[52] Galen Hunt and Jim Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41/2:37–49, April 2007.

[53] INTEGRITY Real-Time Operating System. http://www.ghs.com/products/rtos/integrity.html.

[54] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1868–1882, New York, NY, USA, 2018. ACM.

[55] Antti Kantee. *Flexible operating system internals: the design and implementation of the Anykernel and Rump kernels.* PhD thesis, 2012.

[56] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys '17, pages 437–452, New York, NY, USA, 2017. ACM.

[57] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[58] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[59] Jonathan Levin. *Mac OS X and IOS Internals: To the Apple's Core.* John Wiley & Sons, 2012.

[60] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical report, GMD SET-RS, Schlo Birlinghoven, 53754 Sankt Augustin, Germany, 1995.

[61] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based os. *SIGOPS Oper. Syst. Rev.*, 25(2):51–62, April 1991.

[62] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619. ACM, 2015.

[63] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *22nd ACM Conference on Computer and Communications Security (CCS)*, pages 1607–1619, 2015.

[64] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 115–128. ACM, 2011.

[65] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. 2005.

[66] Mellanox. Connectx-6 single/dual-port adapter supporting 200gb/s with vpi. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card, 2019.

[67] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A security-oriented subset of Java. In *Proc. NDSS*, February–March 2010.

[68] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 9. ACM, 2019.

[69] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, May 2006.

[70] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Workshop on Memory Systems Performance and Correctness*, pages 4:1–4:10, 2014.

[71] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT*, pages 261–270. IEEE, 2009.

[72] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs : Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[73] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132, New York, NY, USA, 2013. ACM.

[74] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[75] Phoronix Test Suite: An automated, open-source testing framework. http://www.phoronix-test-suite.com/.

[76] Octavian Purdila. Linux kernel library. https://lwn.net/Articles/662953/.

[77] Matthew J Renzelmann and Michael M Swift. Decaf: Moving device drivers to a modern language. In *USENIX Annual Technical Conference*, 2009.

[78] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[79] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Technical Report*, 2010.

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

[81] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. 2010.

[82] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, pages 1–8, 2010.

[83] Yifeng Sun and Tzi-cker Chiueh. SIDE: Isolated and efficient execution of unmodified device drivers. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[84] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107. ACM, 2002.

[85] Hajime Tazaki. An introduction of library operating system for Linux (LibOS). https://lwn.net/Articles/637658/.

[86] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *Acm Sigplan Notices*, volume 50, pages 1–15. ACM, 2015.

[87] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient inprocess isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[88] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.8, update 3. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[89] Kevin Thomas Van Maren. The Fluke device driver framework. Master's thesis, The University of Utah, 1999.

[90] David A. Wagner. Janus: An approach for confinement of untrusted applications. Technical report, Berkeley, CA, USA, 1999.

[91] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[92] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 241–254, Berkeley, CA, USA, 2008. USENIX Association.

[93] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th Usenix Security Symposium*, 2018.

[94] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.