Device Driver Safety Through a Reference Validation Mechanism *

Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider
{djwill,reynolds,kwalsh,egs,fbs}@cs.cornell.edu

May 9, 2008

Abstract

Device drivers typically execute in supervisor mode and thus must be fully trusted. This paper describes how to move them out of the trusted computing base, by running them without supervisor privileges and constraining their interactions with hardware devices. An implementation of this approach in the Nexus operating system executes drivers in user space, leveraging hardware isolation and subjecting them to reference validation. These Nexus drivers exhibit performance nearly as fast as earlier in-kernel, trusted drivers. For example, the monitored driver for an Intel e1000 Ethernet card has throughput comparable to a trusted driver for the same hardware under Linux. And a monitored driver for the Intel i810 sound card provides continuous playback. Drivers for a disk and a USB mouse have also been moved successfully to operate in Nexus user space with reference validation.

1 Introduction

A Microsoft study reports that 85% of crashes in Windows XP result from device driver failures (see [27]), and a similar study based on automated bug-finding tools claims that Linux driver code has an error rate up to seven times higher than other kernel code [8]. Yet device drivers are part of the trusted computing base (TCB) of every application, because the monolithic architecture of mainstream operating systems forces device drivers to be executed inside the kernel, with high privilege. The situation is not substantially different for exokernels [10], which also run device drivers inside the kernel for performance reasons. Some microkernels and other research operating systems [2, 7, 19, 22] run device drivers in user space to isolate the operating system from accidental driver faults, but these drivers retain sufficient I/O privileges that they must still be trusted.

The advantages of isolating device drivers from each other and from other system components are well known [6, 24, 28]. However, such isolation is rare because device drivers require a rich interface to the rest of the system and because their performance requirements constrain how they may be partitioned.

This paper introduces a practical mechanism for executing device drivers in user space and without privilege. Specifically, device drivers are isolated using hardware protection boundaries. And each device driver is given access only to the minimum resources and operations necessary to support the devices it controls (least privilege), thereby shrinking the TCB.[1] A system in which device drivers have minimal privileges is easier to audit and less susceptible to Trojans in third-party device drivers.

Device drivers that run in user space still need to initiate hardware I/O operations and handle interrupts. These operations can cause device behavior that compromises the integrity or availability of a kernel or other programs. Therefore, in our driver architecture, a global, trusted *reference validation mechanism* (RVM) [3] mediates all interaction between device drivers and devices. The RVM invokes a device-specific *reference monitor* to validate every interaction between a device driver and its associated device, thereby ensuring the driver conforms to a *device safety specification* (DSS), which defines allowed and, by extension, prohibited behaviors.

The DSS is expressed in a domain-specific language and defines a state machine that accepts permissible transitions by a monitored device driver. We provide a compiler to translate a DSS into an executable reference monitor that implements the state machine. Every operation by the device driver is vetted by the reference monitor, and operations that would cause an illegal transition are blocked. The entire architecture is depicted in Figure 1.

*Supported by NICECAP cooperative agreement FA8750-07-2-0037 administered by AFRL, AFOSR grant F49620-03-1-0156, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), ONR Grant N00014-01-1-0968, and Microsoft Corporation.

[1] Some drivers, such as the clock, provide functionality needed for defining or enforcing security policies. These device drivers remain part of the TCB no matter where they execute.

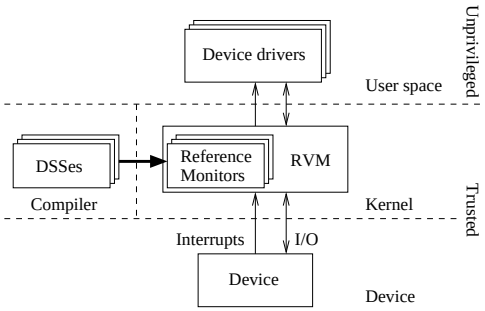

Figure 1: Safe user-space device driver architecture.

The RVM protects the integrity, confidentiality, and availability of the system, by preventing:

- **Illegal reads and writes:** Drivers cannot read or modify memory they do not own.
- **Priority escalation:** Drivers cannot escalate their scheduling priority.
- **Processor starvation:** Drivers cannot hold the CPU for more than a pre-specified number of time slices.
- **Device-specific attacks:** Drivers cannot exhaust device resources or cause physical damage to devices.

In addition, given a suitable DSS, an RVM can enforce site-specific policies to govern how devices are used. For example, administrators at confidentiality-sensitive organizations might wish to disallow the use of attached microphones or cameras; or administrators of trusted networks might wish to disallow promiscuous (sniffing) mode on network cards.

One alternative to our approach for monitoring and constraining device driver behavior is to use hardware capable of blocking illegal operations. An IOMMU [1, 4, 12, 21], for example, limits the ability of devices to perform DMA transfers to or from physical addresses the associated drivers cannot read or write directly. This mechanism, however, does not mediate other aspects of driver behavior, so it is strictly less powerful than an RVM. For example, an IOMMU cannot prevent interrupt livelock (as exemplified in Section 5.2), limit excessively long interrupt processing, or protect devices from physical harm by drivers. IOMMUs also cannot enforce limitations on the use of cameras, microphones, or network sniffing.

In sum, this paper shows how to use standard memory protection and device-specific reference monitors to execute device drivers with limited privilege and in user space. The requisite infrastructure is small, easy to audit, and shared across all devices. Our prototype implementation demonstrates that this approach can defend against malicious drivers and that the performance costs of this enhanced security are not prohibitive.

The rest of this paper is structured as follows. Section 2 describes the device I/O model, which dictates assumptions underlying our design. Section 3 describes RVM functionality and the DSS language. Section 4 describes our instantiation of these ideas in the Nexus operating system. Section 5 reports on the performance, robustness, and size of our unprivileged, isolated drivers for sound cards, mice, network interface cards, and disks. Section 6 surveys relevant work on driver isolation and hardware specification, while Section 7 concludes.

2 Device I/O Model

Device drivers send commands to devices, check device status using registers, receive notification of status changes through interrupts, and initiate bulk data transfers using direct memory access (DMA). How they do so constitutes a platform's *I/O model*. Our work is targeted to the x86 architecture and PCI buses; what follows is a brief overview of the I/O model on that platform. Similar features are found on other processors and buses.

Modern buses implement device enumeration and endpoint identification. Each device on a PCI bus is identified by a 16-bit vendor identifier and a 16-bit model number; the resulting 32-bit *device identifier* identifies the device. *Device enumeration* is a process for identifying all devices attached to a bus; *endpoint identification* is the process of querying a device for its type, capabilities, and resource requirements.

Device enumeration and endpoint identification typically occur at boot time. Interrupt lines and I/O registers are assigned, in accordance with device requests, to all devices discovered. Device identifiers govern which device drivers to load.

Devices have *registers*, which are read and written by drivers to get status, send commands, and transfer data. The registers comprise I/O ports (accessed using instructions like `inb` and `outb`), memory-mapped I/O, and PCI-configuration registers. Each register is identified by a *type* and an *address*. Contiguous sets of registers constitute a *range*, identified by type, base address, and limit (the number of addresses in the range). For all register types, accesses are parameterized by an address, a size, and, for writes, a value of the given size. Write operations elicit no response; read operations produce a value of the given size as a response. Both operations can cause side effects on a device.

Devices that transfer large amounts of data typically employ DMA rather than requiring a device driver to transfer each word of data individually through device

registers. To initiate a DMA transfer, the device driver typically writes a pointer into a device's control register. Some devices can perform DMA to or from multiple memory regions if the driver writes a pointer for a list of regions. Device drivers using DMA transfers must first obtain from the kernel a memory region with a known, fixed, physical address.

Devices can be *synchronous* or *asynchronous*. Drivers must poll synchronous devices for completed operations or changes in status. In contrast, when a driver submits an operation to an asynchronous device, the driver can yield the CPU until the device later signals its response (or any other status change) by interrupting the processor. When that interrupt occurs, the operating system invokes code specified by the driver. In most cases, an interrupt must be acknowledged by a driver, or the device will continue to send the same interrupt. Interrupts can be prioritized relative to each other, but they generally occur with a high priority, preempting most other tasks.

Each device signals interrupts using a pre-assigned *interrupt line*. On some architectures, including the x86, interrupt lines can be shared by multiple devices. Drivers must read status registers for each of these devices to determine which specific device caused the interrupt.

Devices are assumed to be in an unknown state when an operating system boots or when a driver is loaded or reloaded. When a driver is unloaded, it unregisters its interrupt handler and releases its DMA memory. At that point, the device must be placed in a state that does not generate interrupts or use DMA.

Devices are typically forgiving about device driver timing, and device drivers are similarly forgiving about device timing. This flexibility is a necessity, because a modern multitasking operating system might be heavily loaded, implement arbitrary scheduling policies, or at times execute with interrupts disabled. In addition, devices and their drivers are typically designed to work with several processor generations, which differ in execution speed. Device registers and interrupts, rather than precise timing, are used to implement synchronization between the device and its driver so that devices and drivers behave safely and predictably despite uncertain delays.

Some drivers are divided into components or hierarchies. For example, SCSI, ATA, and USB each have a controller driver plus additional drivers for peripherals, like disks, mice, keyboards, etc. In such *driver hierarchies*, only the device driver for the controller performs actual I/O operations, handles interrupts, or initiates DMA transfers. Drivers for peripherals communicate with their devices through the controller driver, hence the peripheral driver need not be monitored. So a single DSS for the controller suffices to handle the entire hierarchy.[2]

Some devices, particularly high-performance network cards, support loadable firmware, which executes on the device and thus can change the way the device behaves. This firmware must be trustworthy [16]. Firmware is loaded through I/O operations or DMA, a sequence of events that can be monitored. In principle, then, an RVM could authenticate firmware using signatures or perform analysis to show the firmware is trustworthy. Our current DSSes do not implement these checks. Doing so would be straightforward, though designing an analysis algorithm might not be.

3 Unprivileged Driver Architecture

In our user-space driver architecture, drivers, like any other user process, are loaded from a filesystem; once loaded, they execute and can be unloaded and restarted at any time. When a driver is first loaded, it executes a system call to find a compatible device. As part of this system call execution, the RVM identifies an appropriate device and reference monitor and returns to the driver a structure describing the device ID and I/O-resource assignments. Henceforth, the driver uses a driver system call interface (described in Section 4.3) to perform I/O operations and receive interrupts. Subsequent uses of that interface cause the RVM to invoke the reference monitor.

Reference monitors are instantiated immediately after endpoint enumeration, based on device IDs. Reference monitors persist, even if corresponding drivers are unloaded and restarted.

3.1 Security properties

Drivers are not trusted, but the RVM, reference monitors, and devices are. Moreover, reference monitors are compiled from DSSes, so DSSes and the DSS compiler must be trusted.

Some DSSes will be written by hardware manufacturers; others will be written by independent experts, including security firms or OS distributors. But independent of the source, a DSS ought to be small and declarative, hence conducive to auditing.

We assume devices behave safely if given sufficiently restricted inputs. Such an assumption is inescapable, because devices have the ability to read and write any mem-

[2] We have nevertheless developed an approach to composite reference monitors: the composite reference monitor is derived from the controller reference monitor and an auxiliary reference monitor for each attached device. In practice, the only property that changes when peripherals are attached seems to be the interrupt rate limit.

ory, generate arbitrary interrupts, or even starve hardware buses directly.

The two sources of driver misbehavior we consider are drivers designed by malicious authors (Trojans), and drivers with bugs that can be subverted by users or remote attackers. Both are dealt with by our RVM.

The RVM prevents drivers from performing invalid reads and writes using hardware isolation and by checking driver accesses to DMA control registers.

- Hardware isolation works as with other user processes, giving each driver process direct access only to its own memory space.
- By checking that every DMA address sent to the device is allocated to the driver, the RVM prevents a device driver from using DMA for illegal reads and writes.

The RVM must also defend against a device driver that attempts to escalate its execution priority or that starves other processes and the kernel by causing large numbers of interrupts or by spending too much time in high-priority interrupt handlers. A timer driver might set too high a timer frequency, or a sound card driver might set too small a DMA buffer for playback, causing frequent notifications to be generated when the buffer becomes empty. Some of these unacceptable behaviors can be prevented when the driver is setting up the device—for example, by a reference monitor imposing a lower bound on the sound card DMA buffer size—but RVMs provide three additional protection measures. First, the RVM limits the frequency at which a driver can receive interrupts, with different limits for different types of devices. Second, the RVM limits the length of time that an interrupt handler runs. Finally, the RVM ensures that each interrupt handler acknowledges every interrupt, to prevent devices from issuing additional interrupts for the same event. (The details of monitoring interrupt handlers in our Nexus implementation are described in Section 4.1.)

Finally, an RVM must identify and prevent invocations of operations known or suspected to harm devices. Examples include: overclocking processors, sending a monitor an out-of-range refresh rate, instructing a disk to seek to an invalid location, or writing invalid data to non-volatile configuration registers. Other attacks against devices involve exhausting finite resources, such as wearing out flash memory by writing repeatedly to one block or wasting battery power on mobile devices. The RVM prevents these attacks simply by preventing operations that would cause unsafe device states and by rate-limiting operations that would exhaust device resources.

Notice that no effort is made to protect data contents, including message sources and destinations. The RVM does not, for example, prevent a malicious driver from mirroring packets to an attacker and does not prevent a disk driver from writing data to the wrong block. Such protections concern end-to-end properties, hence they are best implemented above the driver level.

3.2 Device safety specifications (DSS)

Each DSS describes the *states* and *transitions* for a *state machine* and is compiled to create a reference monitor. Inputs to the reference monitor—operations executed by a driver and events from the corresponding device—are delivered serially to the reference monitor by the RVM. When an input does not correspond to an allowable transition, then the reference monitor deems it illegal, the RVM terminates the driver for the corresponding device, and the device is reset.

The state of a DSS state machine records interesting aspects of the history of operations and events. This state is defined in terms of *state variables*, and it often correlates with the state of the I/O device itself. Some of these state variables are explicitly defined by the program; others are implicitly defined by the RVM.

Implicitly defined state variables are given values by the RVM as a result of registration events (see Section 4.1). The implicit variables `$PORTIO[]`, `$MMIO[]`, `$PCIREG[]`, and `$INTR[]` identify I/O registers and interrupt lines set during endpoint identification. And `$MONITORED[]` and `$UNMONITORED[]` describe two types of memory regions allocated by the driver for DMA transfers. Access to a monitored memory location generates an input to the reference monitor; this form of memory is used to store commands or pointers to DMA regions, similar to device registers. Access to an unmonitored memory location is not visible to the RVM, making unmonitored memory suitable only for holding data not relevant to the DSS, such as audio samples from a sound card. Unmonitored reads and writes are considerably faster than monitored reads and writes.

Each state machine transition is specified with a predicate P_i and an action A_i. P_i is a boolean expression over events and state variables. A_i is a program fragment that modifies state variables to produce the new state. A transition that pairs a predicate P_i and an action A_i is written using the syntax $P_i \ \{ \ A_i \ \}$.[3]

[3]Some predicates and actions are too complex to write in terms of the simple syntax currently supported by our DSS language, where user-defined state variables must be scalars, and predicates cannot be recursive. The DSS compiler therefore supports embedded blocks of C, coded as `C:{...}`, appearing in predicates and in actions. Within an embedded C block, it is possible to nest an embedded block of DSS code, e.g.,

Any operation or event—though this is most useful for interrupts—can be assigned a rate limit as part of a DSS. Rate limits can be manually incorporated into transitions using counters and timers. As a convenience, the notation P_i <$rate$, max, $start$> { A_i } compiles to a transition with a leaky bucket expressing a rate limit. So, the associated transition can occur at most $rate$ times per second; bursts are allowed beyond this rate, up to max occurrences at once; when the driver starts, it has $start$ initial capacity.

As an example, an abridged version of our DSS for the Intel i810 audio device appears in the Appendix.

4 Implementation

We instantiated our user-level device driver architecture in the Nexus trusted operating system [25], which has many similarities to traditional microkernels, including hardware-implemented process isolation. Other operating systems that support process isolation (e.g., Linux or Windows) could also host an RVM.

Our implementation of user-space, unprivileged device drivers in Nexus includes the RVM, an event interface between the RVM and the reference monitor, a system call interface by which drivers can request services from the RVM, and a mechanism for limiting driver execution time and the frequency of events. We discuss each of these below and report on our experience porting Linux kernel device drivers to Nexus user space.

4.1 Reference monitor interface in Nexus

Reference monitors define functions that the RVM calls to initialize implicit state variables and to deliver inputs to be checked. These inputs are sent in response to driver system calls and device events. Each I/O operation and event described in Section 2 causes a distinct input.

State-variable setup. After device enumeration and endpoint identification occur, Nexus initializes one reference monitor for each device. The implicit state variables are arrays. The RVM populates them based on the results of endpoint enumeration by calling the function `register_region` to set up I/O ports, memory-mapped I/O, and PCI configuration registers and the function `register_intr` to set up an interrupt line.

to use an identifier or an operator not available in C. Our syntax was inspired by Java and C nesting in Jeannie [18].

Driver and device events. Device drivers affect the state of the system and the reference monitor in three ways: by performing I/O, by allocating memory, or by exiting. When the driver reads or writes a register or a monitored memory location, the RVM sends `read` or `write` events to the reference monitor. After a `read` operation, the device responds with a value, generating a `read_response` event. The `read` operation can be blocked if it would cause a disallowed side effect. The `read_response` event is never blocked, and the value it conveys can be used to change state variables.

A driver can allocate memory to use for DMA, which causes the RVM to send `register_region` events with a region type of `MONITORED` or `UNMONITORED`. Finally, if the driver exits or executes an operation not permitted by the DSS, the RVM sends a `reset` event.

Devices affect reference monitor state when sending interrupts, which generate `intr` events. When the reference monitor gets an `intr` event, it sets an interrupt `status` flag (the reference monitor state implicitly includes one flag per interrupt line) to `pending`, and the RVM schedules the driver with high execution priority. The driver then has a configurable amount of time to respond to the interrupt, by checking if the interrupt was from its device, and if so, acknowledging it so that the device does not generate further interrupts for the same device event or completed operation. This acknowledgment is implemented with I/O device `read` and `write` operations; reference monitors recognize the acknowledgment as a transition and reset the interrupt flag to `idle`. The RVM calls the `get_intr_status` function on each operation after an `intr` event (i.e., while the driver's interrupt handler is executing). As soon as the interrupt `status` flag is reset to `idle`, the RVM lowers the driver's execution priority to its default level. If the driver does not reset the `status` before the allowed time has elapsed, then it is treated as a priority escalation attack; the RVM terminates the driver and resets the device.

When an interrupt occurs on a shared line, the RVM notifies all drivers sharing that line. Each driver must then query its device to see if it was the source of the interrupt. This approach correctly handles merged interrupts, where two or more devices generate an interrupt at the same time, as well as spurious interrupts.

4.2 Rate limiting in Nexus

A device managed by a well-behaved driver should not exceed rate limits enforced by the reference monitor. Drivers can call `driver_get_rate_limits` to learn such rate limits and can manage interrupts using a throt-

tling mechanism provided by the device or by disabling interrupt-generating acts by the device when an interrupt would be disallowed.

The RVM could impose rate limits on uncooperative drivers directly or by terminating a driver when its associated device exceeds the limit. We implement the latter in Nexus. If an RVM can mask interrupts from each device independently (e.g., as with non-shared interrupts or edge- or message-signaled interrupts), then the RVM could limit the interrupt rate by masking interrupts that would exceed a rate limit. However, for shared, level-triggered interrupt lines, this approach delays interrupts for all drivers sharing the line. Since limits cannot be enforced by masking these interrupts, the driver associated with a device that violates rate limits must be terminated.

To ensure that rate limits are applied fairly to interrupts on shared lines, only acknowledged interrupts are counted. The RVM queries each reference monitor's state using the `get_intr_status` function to learn how the monitored driver handled the interrupt—by deciding it was for a different driver, or by acknowledging it.

4.3 System calls in Nexus

Nexus implements system calls for drivers to find a device, allocate memory, and perform I/O operations:

- `driver_init_pci(pci_ids[], &device)` is the main initialization routine. A device driver calls it to find devices and to find their I/O registers and interrupt lines. The first parameter is a list of PCI IDs the driver can manage. The `device` parameter returns a structure describing the I/O registers and interrupt lines for the driver to communicate with the device.
- `driver_allocate_memory(size, is_monitored, &v_addr, &p_addr)` allocates kernel memory for DMA buffers and returns the virtual and physical addresses to the device driver. The `is_monitored` parameter indicates if reads and writes should be checked by the reference monitor. If the allocated region is unmonitored, then the reference monitor will not allow pointers to that region to be written to registers that require monitored memory, such as DMA indices and command buffers.
- `driver_wait_for_intr(intr)` blocks the calling thread in the device driver until an interrupt arrives on the specified interrupt line. Normally, one thread in a driver runs a loop that executes this system call and runs an interrupt handler when the call returns.
- `driver_get_rate_limits()` returns rate limits for all transitions as an array of leaky bucket definitions. A driver can use this information to delay operations and interrupts so that no behavior exceeds rate limits.
- `driver_read(region, addr, len)` and `driver_write(region, addr, len, val)` read and write port I/O, memory-mapped I/O, PCI configuration registers, and monitored DMA memory.

4.4 Driver source compatibility

Rather than write new drivers for Nexus, we used drivers from Linux 2.4.22.[4] Our original goal was source compatibility between these Linux drivers and Nexus user space drivers. However, the Linux drivers did not provide some of the information necessary to enforce a DSS efficiently. Moreover, small changes to driver source code promised to reduce our overall effort in porting Linux drivers to Nexus and to make the resulting Nexus drivers more efficient. So we used a hybrid approach, implementing general-purpose compatibility functions for Linux drivers and also changing Linux driver code to work better with an RVM. The compatibility functions provide user-space equivalents of global variables and functions in the Linux kernel that Linux drivers would normally access directly.

Linux I/O operations. Linux drivers use functions and macros for most I/O operations. Port I/O and MMIO are implemented by macros for reading and writing each valid word size. PCI register I/O is implemented using functions. For our Nexus port, we redefined these macros and functions to call `driver_read` and `driver_write`.

Linux drivers read and write DMA memory by dereferencing pointers or by calling functions like `memcpy`. We map monitored DMA memory to invalid pages so that accessing it causes page faults. The RVM includes a trap handler that redirects these page faults to `driver_read` and `driver_write` system calls. System calls are faster than page faults (see Section 5.1), so programmers may change monitored DMA memory operations to explicit system calls wherever performance is critical.

Linux memory allocation. The Linux kernel provides a variety of memory allocation functions to allocate different block sizes, in interrupt or non-interrupt contexts, in low or high (beyond 1GB) memory, and with

[4] Linux 2.4.22, though not current, is the version on which parts of Nexus are based. We chose to copy drivers from this version of Linux to simplify implementation.

or without contiguity requirements. DMA buffers must be in low memory and mapped contiguously; network packet buffers must be allocated in the context of network interrupts. We redefine those functions to call `driver_allocate_memory`, which implements the subset of memory allocation functionality needed by our drivers. The `driver_allocate_memory` call provides contiguous memory with known addresses appropriate for DMA but does not differentiate by block size and does not allocate high memory. Memory without DMA or concurrency requirements is allocated in user space from the heap. To provide allocation in an interrupt context without deadlocking, we implemented preallocated memory pools.

Memory used for DMA operations must be *pinned*: it must have a fixed physical address and never be paged to the disk. Pinned memory is more expensive to maintain and has a stricter quota than normal heap memory. All memory, including DMA memory, is automatically freed when a driver exits. Drivers (and other processes) can free heap memory at any time, but DMA memory can only be freed when a driver exits. Freeing DMA memory in an active driver would require expensive checks by the reference monitor to ensure the device cannot use the memory in the future. Freeing DMA memory also leads to fragmentation, which makes all subsequent checks of pointers to DMA memory more expensive. Fortunately, in practice, all the Linux drivers we ported except the USB controller driver allocate DMA memory once and free it only when they exit; we easily modified the USB driver to behave this way.

Mutual exclusion. Device drivers can be invoked concurrently from client applications and from interrupts. The former is easily handled using standard thread synchronization mechanisms. The latter is not, because interrupts do not respect these mechanisms, and requesting a lock from an interrupt handler is infeasible.

Linux drivers synchronize concurrent invocations from clients using locks, which Nexus also provides. However, Linux drivers typically implement synchronization with devices by disabling interrupts. While interrupts are disabled, the driver cannot be interrupted by other drivers or by the kernel. But making this same functionality available for untrusted user-space drivers allows starvation attacks.

Fortunately, typical devices need only non-reentrant code sections, which we implement using *driver mutexes*. After a driver thread acquires a driver mutex, the Nexus scheduler marks all other threads associated with the same device as not runnable; the kernel and other user-space threads are unaffected. Interrupts for this driver are delayed until it releases the mutex.[5] Thus, driver mutexes implement non-reentrant sections of code. The driver can be interrupted by other programs, including other drivers, but not by concurrent invocations or by interrupts.

Driver	Linux LoC	Lines changed	Lines added	DSS LoC
i810	5,500	26	56	149
e1000	11,849	50	3	303
USB UHCI	13,328	169	525	508
USB mouse	650	6	16	-
USB disk	19,767	29	121	-
Compat library	-	-	5,523	-

Figure 2: Lines of code in each ported Linux driver and DSS. USB mouse and disk drivers are monitored by the UHCI DSS.

Using driver mutexes instead of disabling interrupts poses problems for drivers that must synchronize with other drivers. We could implement such synchronization by adding ordinary mutual exclusion to the driver. Driver mutexes also may pose problems for drivers that require atomicity or precise timing. For example, the Linux i810 sound card driver calibrates playback speed by measuring playback progress over a fixed-length period during initiation. Precise scheduling is difficult to implement for user space and can be viewed as a privilege that drivers do not need. So we rewrote that sound card driver to measure the interval over which its calibration routine ran rather than using a fixed-length period, because measuring time in user space requires no special privileges.

5 Results

We implemented user-space device drivers for the i810 sound card, e1000 network card, USB UHCI controllers, USB mice, and USB disks in the Nexus operating system. We here quantify the performance, robustness, and complexity of these drivers, their DSSes, and the Nexus RVM.

We quantify the ease of driver porting and the auditability of DSSes by counting the number of lines of code in each DSS and the number of lines changed to port each Linux driver to Nexus. These counts are given in Figure 2. We distinguish between lines we modified in the Linux driver files and lines we added in new files. The number of changed and added lines was small, and as expected, each DSS is dramatically smaller than the corresponding driver.

[5] This technique would be both correct and efficient on multiprocessor systems, although Nexus does not yet run on multiprocessors.

We wrote each DSS by referring to the manufacturer's documentation about device behavior and to existing drivers. The DSS for USB UHCI was derived entirely from the documentation. The i810 and e1000 DSSes are based on documentation that describes features our drivers actually use; other features are disallowed by the DSS. Writing a DSS based on an existing driver is tempting, but risks disqualifying other drivers that attempt unknown (but safe) behavior. Writing a DSS based on all features described in published documentation is more time-consuming, but in theory, it admits any legal driver. We estimate the time to develop a DSS, given a working driver, manufacturer's documentation, and familiarity with the DSS language but not with the device, as one to five days.

5.1 Driver performance

To gain insight into the performance of our user-space device drivers, we tested each at idle and under load. Our test system was a 3.0 GHz Pentium 4 system dual-booting Nexus and Linux 2.4.22. For network tests, the remote host was a 2.4 GHz Athlon 64 X2 system running Linux 2.6.22, connected over a switched, lightly loaded 1 Gbps network.

To obtain a detailed breakdown of the sources of overhead, we instrumented several versions of the e1000 network driver and the i810 sound driver:

- **Linux**: An in-kernel Linux driver.
- **Kernel**: An in-kernel Nexus driver.
- **Unsafe**: A Nexus user-space driver, but with no reference monitor. This driver has direct access to I/O and DMA.
- **Nullspec**: A monitored Nexus user-space driver but with the trivial reference monitor, which is satisfied by any sequence of events.
- **Safe**: A driver with a full reference monitor.

These driver versions specifically quantify the costs of running under Nexus (Kernel), running in user space (Unsafe), monitoring I/O and DMA operations (Nullspec), and checking operations against a specification (Safe). Overall, these drivers permit us to apportion the costs of safe user-space drivers to the various mechanisms needed to support them.

To test bulk data throughput of the e1000 driver, we sent UDP packets at a constant rate of 1 Gbps to and from a Linux host running Iperf [29]. We varied the size of each packet from 100 bytes to 1470, in order to find the limits of packet-processing rate and data rate. Figures 3 and 4 show the performance, in Mbps and in thousands of packets per second, for all versions of the e1000 driver. All five versions of the e1000 driver performed identically when receiving packets. The three user-space drivers—Unsafe, Nullspec, and Safe—show somewhat degraded performance when sending packets smaller than 800 bytes. The user-space drivers take longer to handle interrupts, and sending generates more interrupts than receiving because the e1000 driver receives (but does not send) many packets per interrupt under heavy load.

To measure interrupt handling times, we instrumented the interrupt handler for the i810 driver. This test uses the CPU cycle counter for nanosecond timing, with instrumentation added to the kernel's trap function (where an interrupt is first visible to software) and to the exit point of the interrupt handler. Average interrupt processing time, over 120 samples, was $5.3 \pm 0.2\mu$s for Linux, $8.5 \pm 0.2\mu$s for Kernel, $22.1 \pm 1.5\mu$s for Unsafe, $37.9 \pm 2.4\mu$s for Nullspec, and $46.9 \pm 3.8\mu$s for Safe. So, the user-space interrupt handlers took three to five times as long as the in-kernel Linux and Nexus drivers. This slowdown is not unexpected, because user-space handlers require a scheduler invocation and two or more context switches.

A macrobenchmark for network round-trip time, which includes driver response time, is the `ping` command, which sends an ICMP echo request packet and receives an ICMP echo reply packet in return. The replies are normally generated by the remote kernel, resulting in low latencies. The elapsed time between sending the request and receiving the reply is the network round-trip time plus the time required for the remote host to process the request. We measured ping times from a Linux box to a Nexus box running each of the four test e1000 drivers. The average round-trip time, over 100 packets, was $103 \pm 35\mu$s for Kernel, $139 \pm 41\mu$s for Unsafe, $158 \pm 55\mu$s for Nullspec, and $156 \pm 54\mu$s for Safe.

These measurements reflect some simple optimizations in the network driver:

- We changed monitored DMA memory accesses from dereferences (i.e., page faults) to explicit system calls.
- We combined sequences of unconditional reads or writes into a single system call. The driver writes between 8 and 2,048 bytes in a logical operation. Normally, these are written 4 bytes at a time; we added a system call to handle a sequence as one operation.
- We stored in the driver the result of reads from a status register. The driver reads the register repeatedly to check several bits. It does not need (and is not expecting) fresh values each time. Thus, we combined several nearby reads into a single system call.

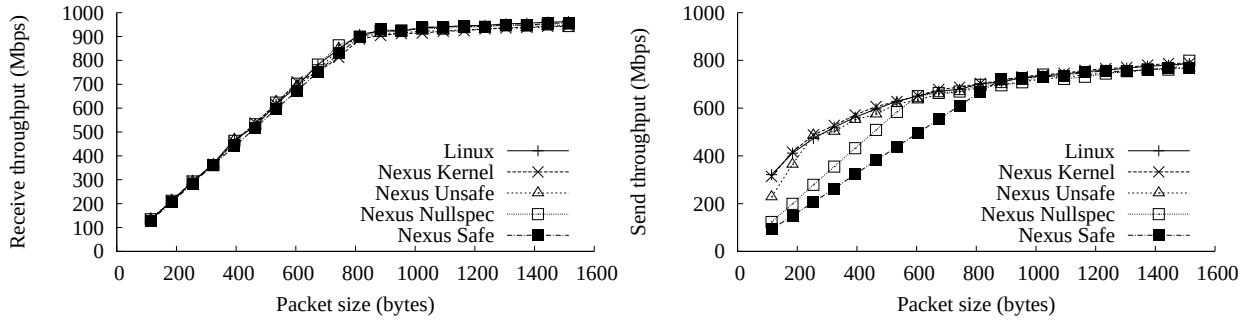

Figure 3: Throughput (Mbps) sent and received by all versions of the e1000 driver using Iperf.

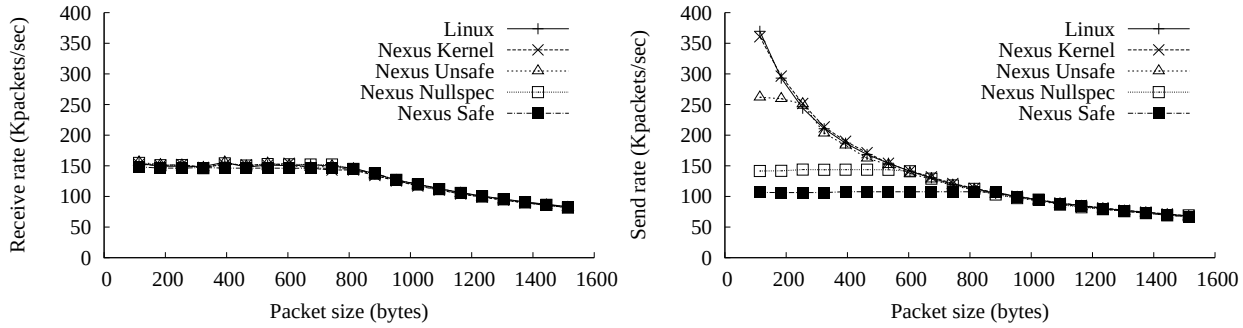

Figure 4: Throughput (thousands of packets/second) sent and received by all versions of the e1000 driver using Iperf.

Optimizations	Packets/sec	Throughput
Page faults	43,203	511.6 Mbps
Syscalls	65,074	753.5 Mbps
Syscalls+batching+caching	123,328	947.7 Mbps

Figure 5: Performance effects of replacing page faults with system calls, then batching and caching groups of operations.

We determined where to apply these techniques by identifying code in the driver that most often called `read` and `write` system calls and caused page faults. We changed 39 lines of driver code (in less than half a day), with dramatic results: we nearly doubled the receive bandwidth and nearly tripled the packet processing rate. Figure 5 shows these results.

Another important driver performance metric is the CPU time spent in drivers while performing a high-level task. To quantify CPU time for our drivers, we streamed video (with audio) over HTTP and played it using `mplayer`. The video averaged 1071 Kbps and lasted for 30 seconds. The resulting CPU time spent in the network driver, the audio driver, and the kernel is shown in Figure 6. CPU utilization is higher in Linux than in Kernel because Linux implements TCP/IP in the kernel, and Nexus implements it as a user-space library; application and library CPU time are not shown.

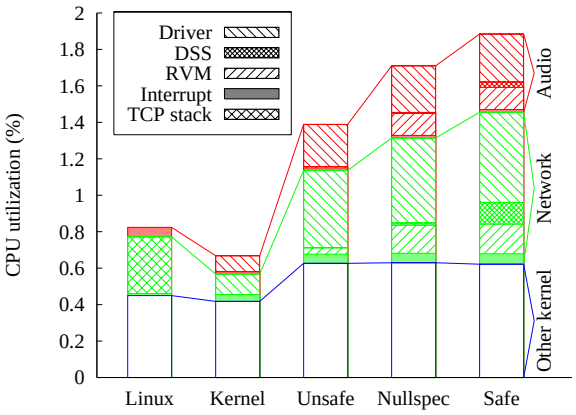

Figure 6: CPU time apportionment when streaming video over the network.

We measured how often each driver executes basic operations and what each basic operation costs. The frequencies of memory, port I/O, MMIO, and interrupts are

	Audio (playback)	Network (idle)	Network (load)	USB (idle)	USB (mouse)	USB (disk)
Unmonitored mem	8018	0	4578113*	8535	19159	223346
Monitored mem	78.3	5.6	42459	0	1930	103374
Port I/O	279	0	0	267	764	956
Interrupts	15.7	1.1	2079	0	124	138
MMIO	0	139	10586	0	0	0

Figure 7: Average rate (per second) of read and write operations during steady-state operation. (* estimated result)

shown for each driver in Figure 7. All figures are the average rate per second when the driver is idle or under load, as indicated. For this test, the network load was a flood `ping`. Counting unmonitored memory operations (by making them monitored) makes the e1000 too slow for our tests. Hence, we estimated the rate of unmonitored memory operations for the e1000 by measuring a heavily instrumented driver under partial load, scaling its results up to what they would have been given full load.

Unmonitored memory operations are anywhere from two to 100 times more frequent than monitored memory operations, depending on the driver. We measured the average cost, over 100,000 tests, of an unmonitored memory operation as 0.59ns, a monitored memory operation executed as a system call as 0.84μs, and a monitored memory operation that causes a page fault as 1.53μs. Page faults are more expensive because they must save more state and because the page fault handler must disassemble and interpret the faulting instruction.

The cost of each basic I/O operation varies relatively little. However, the cost of checking operations against the reference monitor can vary dramatically. Figure 8 shows the cost of checking USB port I/O operations (for disk I/O) against the reference monitor. We found that 80% of the time, the cost is under 2μs. The other 20% of the time, the cost is 190μs or more. This is because an expensive safety check is required only when the value read from a certain register changes. When the value has not changed—80% of the time—the check is cheap.

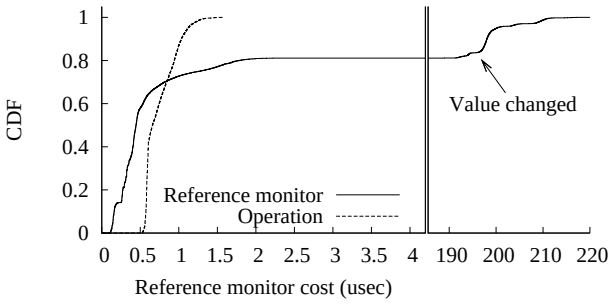

Figure 8: Cost of executing and checking USB disk port I/O operations.

5.2 Driver robustness

Accepted quantitative metrics for the security of a system do not exist. Nevertheless, to establish the security of our RVM and reference monitors, we used two approaches others have used. First, we simulated unanticipated malicious drivers by randomly perturbing the interactions between drivers and the RVM, resulting in potentially invalid operations being submitted to the reference monitor and possibly to the device. Second, we built specific drivers that perpetrate known attacks on the kernel using interrupt and DMA capabilities.

We simulated unanticipated malicious drivers by changing operations and operands in a layer interposed between a legal driver and the RVM. This layer modified each operation according to an independent probability of 1 in 16,384.[6] Each operation was a read or a write; our modifications involved replacing either the address, the length, or the value (at random) with another value in the appropriate range. So, a write to an I/O port was replaced with a write to a port in the same range, a write of a different length, or a write of another value. Reads were perturbed similarly. Note, this approach does not produce repeatable experiments, because driver behavior depends on external factors like the OS scheduler and the arrival times of packets, which are not under our control.

This *perturbation testing* is similar to fuzz testing [23, 28], except that in our approach only I/O operations were modified—not source or machine code. These other types of modifications test isolation properties, whereas we are interested only in testing properties enforced by the RVM and the reference monitor.

We applied perturbation testing to the e1000 driver. When the modifications were benign, the driver showed no apparent failures. Sometimes, the driver itself detected an error (e.g., a status register read failed a sanity check) and exited cleanly. Often, the reference monitor detected an illegal operation, and the RVM terminated the driver. Finally, our perturbations sometimes caused the driver to

[6] We also tried higher and lower probabilities, resulting in more and fewer errors than reported here.

Failure type	Driver			
	Nullspec		Safe	
No failure	7	(23%)	7	(1%)
Driver exits	7	(23%)	16	(1%)
RVM terminates driver		—	1132	(94%)
Driver out of sync	16	(52%)	45	(4%)
Hardware damaged	1	(3%)	0	(0%)
Total perturbation tests	31	(100%)	1200	(100%)

Figure 9: Results of perturbation testing: how the Nullspec and Safe drivers failed, if at all, in repeated tests. Nullspec testing was aborted when it damaged the device.

get out of sync with the device, after which no further packets were sent or received. This does not compromise the integrity or availability of the kernel or the device, so the RVM has no obligation here. Figure 9 summarizes the different cases encountered in our experiments. The Nullspec driver completed more tests with no apparent failure than the Safe driver did, because the reference monitor used for the Safe driver aggressively blocked any unknown behavior—even if it might have been benign.

We hoped the perturbed Nullspec driver would cause kernel livelock, starvation, or a crash. In practice, however, the likelihood of causing such behavior with random perturbations is far below the likelihood of driver crashes and stalls. The 31st run of the Nullspec test rendered the device unusable: neither the Linux nor the Nexus driver could thereafter initialize the card.[7] That ended our Nullspec perturbation testing. We rechecked our performance results on a replacement card, but we do not plan further perturbation testing.

In addition to perturbation testing, we wrote several malicious drivers to execute specific attacks on the kernel using the e1000's interrupt and DMA capabilities:

- **Livelock:** The driver never acknowledges interrupts, resulting in a flood of interrupt activity and starvation for all other processes.
- **DMA kernel crash:** The driver uses the device to write to kernel memory, resulting in a system crash.
- **DMA kernel read:** The driver sends a sensitive page (e.g., containing a secret key) to a remote host.
- **Direct kernel read/write:** The driver constructs a pointer and reads or writes sensitive data directly.
- **DMA kernel code injection:** The driver points a DMA buffer pointer at system call code, then pings a remote machine with attack code.[8] The response is written over the target system call implementation. The attacking driver then invokes the system call to gain control of the kernel.
- **DMA read/write to other device:** The driver uses a ping to overwrite video memory, resulting in an image appearing on the screen.

Not surprisingly, the livelock and DMA attacks succeed when run as Unsafe or Nullspec drivers, all the attacks succeed as Kernel drivers, and they are all are caught by the RVM when run in Safe mode. The livelock attack is prevented by the RVM terminating any driver that does not acknowledge the interrupt by reading the interrupt control register. The DMA attacks are prevented by the RVM terminating any driver that attempts to transmit or receive packets with any invalid addresses in the transmit or receive buffer lists. Finally, any direct attempt to read or write the memory of other drivers is blocked by hardware isolation in all modes except Kernel.

6 Related Work

Several existing operating systems implement device drivers in user space, for isolation or modularity, but without monitoring I/O and DMA operations. Hence, these systems do not defend against malicious operations by drivers. The Michigan Terminal System [7], on the IBM 360 architecture, seems to be the earliest operating system to implement device drivers as user programs. Dijkstra's THE multiprogramming system [9] is organized into *levels*. Level 3 contains device drivers; level 0 implements a scheduler and the interrupt dispatch routine; level 2 implements semaphores, which are used to convey interrupts to device drivers. The EL X8 computer that executed THE did not support memory protection. Hence, THE drivers are not isolated from each other or from the rest of the system. The SUE separation kernel [24] organizes components, including device drivers, into isolated domains akin to hosts in a distributed system. SUE uses memory protection to restrict each driver's access to I/O ports, but it provides no DMA or interrupt protection: DMA is excluded completely, and components are trusted to yield control after each interrupt or task switch.

[7] Would the reference monitor have prevented the damage if it had been enabled for that test? We cannot be sure due to the inherent nondeterminism of peripheral devices, but we believe it would have. We ran 124 reference-monitored tests with no damage to the device.

[8] The e1000 can retrieve any physical memory location by DMA and send it as a network packet, or it can overwrite any physical memory location with the contents of incoming packets. It cannot directly transfer one memory page to another. To get around this, we use `ping` packets; most other hosts will echo a packet with arbitrary contents, which enables us to copy from one local memory location to another by way of a remote host.

L3 [22] moves non-essential device drivers to user space, allowing each driver access to a limited set of I/O ports and delivering interrupts as inter-process messages. The authors acknowledge that drivers requiring DMA access are trusted, and drivers can cause system starvation by disabling interrupts or by failing to acknowledge interrupts.

Leslie et al. [20] implemented user-space device drivers, including an e1000 driver, in Linux 2.6. These drivers are most similar in design, isolation, and performance to our Unsafe user-space drivers—they have device registers and DMA buffers directly mapped into the drivers' virtual memory, so they do not incur monitoring overhead or context switches when performing I/O.

MINIX 3 [17] executes drivers in user space, with limited access to system calls and I/O ports. Drivers that crash are reincarnated transparently to provide continuing service. However, these mechanisms protect against programming errors only, not against malicious drivers.

Nooks [28] and Shadow Drivers [27] provide isolation and fail-over operation for drivers within the Linux kernel, to prevent accidental overwriting of kernel structures. Nooks protects against common bugs, like accidental writes to memory structures belonging to another kernel component. Program rewriting techniques, such as Software-based Fault Isolation (SFI) and its successors [11], implement similar isolation properties. Neither Nooks nor SFI protects restricts what I/O operations are sent to devices.

Microdrivers [14, 15] are a hybrid implementation of Linux device drivers, with up to 65% of the driver running in user space and only the most performance-sensitive code remaining in the kernel. Microdrivers handle network interrupts in the kernel, so they are not secure. Their performance is comparable to the performance of Nexus Unsafe drivers.

Some operating systems take steps to prevent malicious drivers from misusing I/O ports or DMA transfers. Mungi [21] (on Alpha and Itanium platforms) and Scomp [12] (on custom hardware) use an IOMMU for DMA protection. Singularity [19, 26] enforces type-safe interactions between components, including the interactions between drivers and devices. Originally, this type safety meant unmediated access to a restricted set of ports and memory. As IOMMU hardware becomes common, Singularity will rely on IOMMUs to validate DMA operations. Singularity does not limit interrupt rates. Bierhoff and Hawblitzel use an automated theorem prover to verify statically that Singularity device drivers comply with devices specifications written in SPEC# [6]. These specifications limit I/O and DMA misbehavior but not interrupt rates or interrupt handling times.

Virtual machine monitors (VMMs) sometimes use drivers running in a guest operating system to control devices, instead of virtualizing all devices with drivers in the VMM. These *pass-through drivers* are inherently safe for some devices, such as USB peripherals, but not for other devices, such as disks or network cards. Xen [5, 13] puts some device drivers in *driver domains*, which are protected against most crashes but not against malicious behavior; hence, driver domains are trusted.

7 Conclusion

In traditional monolithic and microkernel operating systems, every flaw in a device driver is a potential security hole given the absence of mechanisms to contain the (mis)behavior of device drivers. We have applied the principle of least privilege to Nexus device drivers by creating an infrastructure to run these drivers in user space and by filtering their I/O operations through a reference validation mechanism (RVM). The RVM is independent of drivers and devices; device-specific information is gathered into a device safety specification (DSS) that we compile into a reference monitor. The RVM consults the reference monitor before allowing each I/O operation; any disallowed operation results in the offending driver being terminated.

An obvious question is whether or not the attacks our RVM prevents are realistic. We do not know of malicious drivers "in the wild" that use a device to escalate their privileges, although we have built several of them. The reason such drivers are not yet a real threat is probably that production systems run most drivers in the kernel and in the TCB, where violating security properties can be done directly. Systems with drivers in user space are increasingly common and will inspire attacks through devices. Our RVM and DSS can prevent these attacks.

Acknowledgments. We are grateful to Mike Swift for feedback on a draft of this work.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), Aug. 2006.

[2] B. B. Alessandro Forin, David Golub. An I/O system for Mach. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, Nov. 1991.

[3] J. P. Anderson. Computer security technology planning study—Vol. II. Technical Report ESD-TR-73-51 Vol. II, Electronic Systems Division, AFSC, L. G. Hanscom Field, Bedford, MA, Sept. 1972.

[4] S. Apiki. I/O virtualization and AMD's IOMMU. `http://developer.amd.com/documentation/articles/pages/892006101.aspx`, Aug. 2006.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, Bolton Landing, NY, Oct. 2003.

[6] K. Bierhoff and C. Hawblitzel. Checking the hardware-software interface in Spec#. In *Proceedings of PLOS*, Stevenson, WA, Oct. 2007.

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

[8] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of SOSP*, Banff, Canada, Oct. 2001.

[9] E. W. Dijkstra. The structure of the 'THE'-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[10] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of SOSP*, Copper Mountain, CO, Dec. 1995.

[11] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of OSDI*, Seattle, WA, Nov. 2006.

[12] L. J. Fraim. Scomp: A solution to the multilevel security problem. *Computer*, 16(7):26–34, 1983.

[13] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of The 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.

[14] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *Proceedings of HotOS*, San Diego, CA, May 2007.

[15] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of ASPLOS*, Seattle, WA, Mar. 2008.

[16] J. Hendricks and L. van Doorn. Secure bootstrap is not enough: Shoring up the trusted computing base. In *Proc. of the Eleventh SIGOPS European Workshop, ACM SIGOPS*, Leuven, Belgium, Sept. 2004.

[17] J. N. Herder, H. Bos, and A. S. Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Vrije Universiteit, Amsterdam, The Netherlands, Jan. 2006.

[18] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *Proceedings of OOPSLA*, Montréal, Canada, Oct. 2007.

[19] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, Oct. 2005.

[20] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science & Technology*, 20(5):654–664, Sept. 2005.

[21] B. Leslie and G. Heise. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, University of New South Wales, Sydney, Australia, Mar. 2003.

[22] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based OS. *SIGOPS Oper. Syst. Rev.*, 25(2):51–62, 1991.

[23] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *Proceedings of the IEEE Symposium on Fault-Tolerant Computing (FTCS)*, Madison, WI, June 1999.

[24] J. Rushby. The design and verification of secure systems. In *Proceedings of SOSP*, Asilomar, CA, Dec. 1981.

[25] A. Shieh, D. Williams, E. G. Sirer, and F. B. Schneider. Nexus: A new operating system for trustworthy computing (extended abstract). Brighton, UK, Oct. 2005.

[26] M. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts. In *Proceedings of EuroSys*, Leuven, Belgium, Apr. 2006.

[27] M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, Nov. 2006.

[28] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, Feb. 2005.

[29] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects/Iperf`, May 2005.

Appendix: DSS Example

The following is an abridged version of our DSS for the Intel i810 audio device. It defines the device ID, followed by the state variables and a reset routine. A *NAMES* section then introduces labels for the various events associated with I/O register operations and interrupts. Finally, a *TRANSITIONS* section defines the allowed transitions for the state machine. By default, upon receipt of an input, all transitions are checked, and actions are applied (in unspecified order) for each satisfied predicate. Inside an **ordered** block, transitions are checked sequentially only until a predicate is matched; at most one action is applied inside the block. Several transitions in this DSS have empty actions—they accept an input without changing the state of the state machine.

```
hardware: "PCI:8086:24d5";
monitored region $RING_DMA;        // Define a monitored region to contain DMA descriptors.
const $RING_LEN = 8 * 32;
var $DMA_ENABLED = 0;              // Define a state variable: true when device DMA is active.
reset: C:{                         // Restore device to state with no DMA or interrupts.
   outb(0, $PORTIO[1].base + $CONTROL_OFFSET);              // Turn off playback DMA.
   while(inb($PORTIO[1].base + $CONTROL_OFFSET) != 0) ;     // Wait for acknowledgment.
   $DMA_ENABLED = 0;
}

//***************** NAMES ********************
// Each line maps write, read, and read_response operations on a register (address, size) to a logical name.
// Syntax: <offset, length> --> write_name, read_name, read_response_name;
names for $PORTIO[1], $MMIO[1]:
// Writes to base+0x10 with size=4 are known as write_playback_dma_base.
<0x10, 4> --> write_playback_dma_base($VAL), safe, safe;
<0x16, 1> --> write_status($VAL), safe, read_response_status($VAL);
<0x1b, 1> --> write_control($VAL), safe, safe;                  // Reading the control register is always allowed.
names for $RING_DMA mod 8:                                      // Define names for writes to DMA descriptors.
<0x00, 4> --> write_descriptor_base($ADDR, $VAL), safe, safe;   // offsets 0, 8, 16, ...
<0x04, 4> --> write_descriptor_len($ADDR, $VAL), safe, safe;    // offsets 4, 12, 20, ...
names for $INTR[0]:
 * --> i810_intr;                                               // The only interrupt is named i810_intr.

//*************** TRANSITIONS **************
// Syntax: P_i { A_i }
// Modifying the DMA base register is only allowed if DMA is not running and the address points to monitored memory.
write_playback_dma_base(val) && $DMA_ENABLED == 0 && exists($MONITORED[i]) suchthat
   range(val, $RING_LEN) in $MONITORED[i] { $RING_DMA = range(val, $RING_LEN); }

// Starting DMA is allowed only when the DMA base register points to 32 pointers to pinned, unmonitored memory.
write_control(val) && (val & 0x01) == 1 && $RING_DMA != null && (forall(k) = 0..31 (exists($UNMONITORED[j])
   suchthat range(fetch($RING_DMA.base + 8*k, 4), fetch($RING_DMA.base + 8*k+4, 2)) in $UNMONITORED[j]))
   { $DMA_ENABLED = 1; }
write_control(val) && (val & 0x01) == 0 { $DMA_ENABLED = 0; }

// Changing DMA descriptors is legal if DMA is inactive, or if the modified entry points to pinned, unmonitored memory.
write_descriptor_base(addr, val) && ($DMA_ENABLED == 0) {}
write_descriptor_base(addr, val) && ($DMA_ENABLED != 0) &&
   (exists($UNMONITORED[j]) suchthat range(val, fetch(addr + 4, 2)) in $UNMONITORED[j]);
write_descriptor_len(addr, val) && ($DMA_ENABLED == 0) {}
write_descriptor_len(addr, val) && ($DMA_ENABLED != 0) &&
   (exists($UNMONITORED[k]) suchthat range(fetch(addr - 4, 4), bits(val, 0..15)) in $UNMONITORED[k]);

// The i810 interrupt acknowledgment protocol: first, the driver checks if the interrupt came from i810 by reading status bits 2..4;
```

```
// then, if so, acknowledges it by writing status bits 2..4.
ordered {                              // In an "ordered" block, transitions are checked only until the first match.
   read_response_status(val) && bits(val, 2..4) == 0 { $INTR[0].status = idle; }      // i810 is not asserting an interrupt.
   read_response_status(val) {}                                                       // Otherwise interrupt is still pending.
}
write_status(val) && bits(val, 2..4) != 0 { $INTR[0].status = idle; }                 // Acknowledging interrupts is legal.

i810_intr <16, 1, 1> {}                // Interrupt is rate-limited to 16 per second, no bursts.
```