# iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support

Lin Tan, Ellick M. Chan, Reza Farivar, Nevedita Mallick
Jeffrey C. Carlyle, Francis M. David, Roy H. Campbell

University of Illinois at Urbana-Champaign
201 North Goodwin Ave, Urbana, IL 61801
{lintan2, emchan, farivar2, nmallic2, jcarlyle, fdavid, rhc}@uiuc.edu

## Abstract

*The users of today's operating systems demand high reliability and security. However, faults introduced outside of the core operating system by buggy and malicious device drivers can significantly impact these dependability attributes. To help improve driver isolation, we propose an approach that utilizes the latest hardware virtualization support to efficiently sandbox each device driver in its own minimal Virtual Machine (VM) so that the kernel is protected from faults in these drivers. We present our implementation of a low-overhead virtual-machine based framework which allows reuse of existing drivers.*

*We have constructed a prototype to demonstrate that it is feasible to utilize existing hardware virtualization techniques to allow device drivers in a VM to communicate with devices directly without frequent hardware traps into the Virtual Machine Monitor (VMM). We have implemented a prototype parallel port driver which interacts through iKernel to communicate with a physical LED device.*

## 1 Introduction

The users of today's operating systems demand high reliability and security, however, software bugs and malicious attacks can greatly impact these attributes. Despite extensive testing and validation techniques, many bugs still escape and remain in released software.

The primary cause of most operating system failures is due to errors in the execution of extension code, such as device drivers inside the kernel. Previous work [4, 16] has shown that device drivers are responsible for a disproportionately large number of bugs. These faults are mainly due to device driver code written by third party vendors. Since typically 70% of an operating system consists of device drivers, the opportunity for faults in the driver subsystem is significantly higher. A malicious device driver, operating in kernel mode, can crash the whole operating system or compromise its integrity because of unrestricted access to resources. Since drivers are the main source of operating system failures, this paper focuses on how to make operating systems more tolerant to faults in drivers.

Various techniques to prevent operating system failures caused by device drivers have been proposed [16, 12, 17, 7, 10, 9]. Many of these approaches [16, 17, 7, 14] do not address driver code which is intentionally *malicious*. One of these approaches is based on sandboxing device drivers, which acts as a protection layer [16]. Some are based on implementation using a type-safe language which will prevent errors prior to code execution [17]. Another approach is based on controlling the execution flow of driver code [7].

Microkernel and virtual machine-based approaches, such as [12, 9] can provide security against malicious code, but the communication between the driver and the applications in these approaches is done using a heavyweight message passing approach which can have a large overhead in terms of latency of request and response time.

Furthermore, previous approaches are chiefly based on software techniques and none of them leverages current enhancements made by system architecture vendors to the hardware, such as both Intel's and AMD's processor support for virtualization.

### 1.1 Contributions

We propose a *secure and reliable* architecture, called **iKernel**[1], which utilizes the latest hardware virtualization support to protect a kernel from both buggy and malicious device drivers. Our approach is based on the idea that a device driver isolated in a separate virtual machine will minimally

[1] iKernel stands for isolation Kernel

affect the host operating system or drivers running on other virtual machines. Even if a driver crashes, its effect will be isolated to the virtual machine on which it is running. The Virtual Machine Monitor can stop this virtual machine and restart it to transparently recover without affecting the host kernel or other virtual machines. Drivers running on virtual machines communicate with the host kernel using shared memory communication, which provides better performance in comparison to message-passing based communications. With strong virtual machine isolation guarantees, it is as safe and secure as message passing.

We are the first, to the best of our knowledge, to show that it is feasible to utilize commodity hardware virtualization support to allow device drivers in a Virtual Machine(VM) to communicate with devices directly, without incurring significant trap overhead. We have implemented direct communications between a Linux parallel port driver in a VM and an LED peripheral connected to the parallel port.

We believe that hardware-assisted virtualization can provide at least two advantages: (1) simplification of the isolation mechanisms; and (2) the potential of providing performance benefits over software-based virtualization techniques. We present the remainder of the paper with references to Intel's virtualization technology, but the same concepts apply to AMD's or any other similar hardware-assisted virtualization technologies.

## 2 Intel VT Background

Intel has recently launched Pentium-based processors supporting the new Intel Virtualization Technology (IVT). Formerly known as Vanderpool, this technology provides hardware support for virtualization. A Virtual Machine Monitor (VMM) is a very thin privileged "hypervisor" which resides above the physical hardware. Virtual machines (VM), running on top of the VMM, all run at a reduced privilege level; code running inside a VM including an operating system is said to be de-privileged. One or more of these VMs can be allowed to access physical resources and made responsible for I/O processing and sharing. In our evaluation, we demonstrate and analyze the performance of one VM running Linux with a virtual device driver which is used by the host operating system. With this approach, we can use the device drivers in their unmodified form and still achieve sufficient driver isolation.

## 3 Overall System Architecture

The iKernel system is designed to provide strong isolation mechanisms for device drivers using hardware-assisted virtual machine technology.

In the iKernel system architecture, most device drivers are designed to run in their own virtual machine. The host kernel acts as the primary OS environment of the computing system which will run all of the user processes. Lightweight communication stubs are placed in both host and guest OSes, which can communicate with each other using shared memory mechanisms. Shared memory communication mechanisms are chosen so that communication between stubs incurs the least overhead. The host stub provides the same interface to the operating system as the original driver. For example, a typical Linux driver provides the open, close, read, write and ioctl functions, but instead of executing the request, iKernel forwards the information to the stub driver which runs in the guest virtual machine. The guest stub driver will then relay the call to the real device driver functions, which also run in the guest virtual machine.

If a device driver fails, all damage is isolated to the driver's guest virtual machine environment and cannot corrupt the host kernel memory. Similarly, a malicious device driver cannot access the host kernel's data structures. Although there is a shared memory communication channel between the device driver virtual machine and the host kernel, the interface can be well-defined and therefore the chance of error propagation can be minimized.

Figure 1 depicts a schematic diagram of the iKernel system architecture. The kernel to be protected resides in the host. To prevent drivers from corrupting each other, each driver can have its own VM instance. To conserve system resources, a system designer may collocate several drivers into a single VM. The VM driver stubs will accept intercepted calls from the host driver stubs made by the host kernel and invoke the actual driver code in the VMs. The host driver stubs will process requests from the VM driver stubs and invoke host kernel utilities. These stubs allow the implementation of unmodified devices to be used in our system.

Our system uses the KVM virtual machine built into Linux 2.6.20 for the host OS and virtual machine monitor. KVM affords us a lightweight virtual machine that operates with the Intel VT-x technology. Our guest driver kernels are also based on Linux 2.6.20.

There are two key requirements to the design of the iKernel system:

- The ability to delegate direct control of a physical device to a device driver running in a guest virtual machine.

- An efficient communication channel between the host and the guest virtual machine.

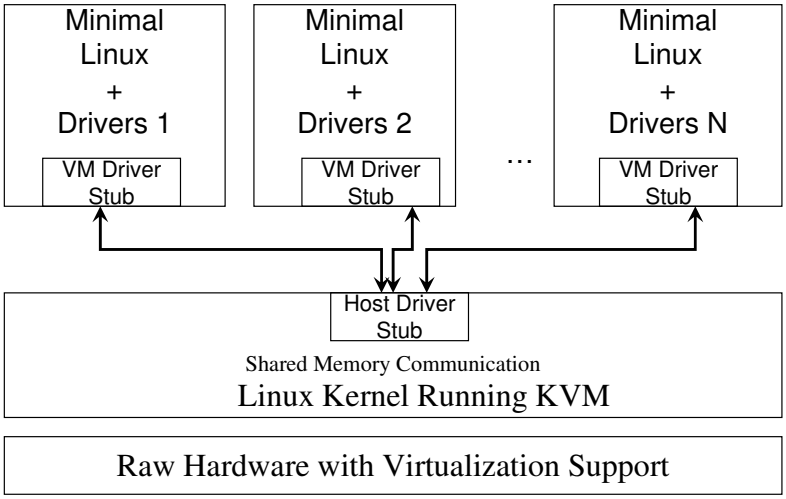


**Figure 1. The iKernel Architecture**

### 3.1 VM Direct Access to Devices

One feature of iKernel is the ability to assign a physical device to a specific virtual machine. When virtualizing an I/O device, it is necessary for the underlying virtualization software to service several types of operations for that device. Interactions between software and physical devices include the following:

- Device discovery: a mechanism for software to discover, query, and configure devices in the platform.
- Device control: a mechanism for software to communicate with the device and initiate I/O operations.
- Data transfers: a mechanism for the device to transfer data to and from system memory. Most devices support DMA in order to efficiently transfer data.
- I/O interrupts: a mechanism for hardware to be able to notify the software of events and state changes.

Current virtualization techniques can be divided into three broad categories with regard to how they manage the I/O devices of a system.

**Emulation:** The first category is device emulation, which provides a set of virtual I/O devices for any guest virtual machine conforming to a specific hardware configuration; this configuration may not necessarily be identical to that of the host. Device drivers running in the guest virtual machine access these virtual devices as if they were physically present. Any device accesses are translated to the corresponding resources on the host. For example, a virtual Ethernet device might forward packets to an actual Ethernet device on the host. The primary drawback of this approach is that it is unable to provide support for a variety of application-specific devices. Device models chosen for virtual machines that use this technique are generally based on a simple interface. However, an unoptimized approach which attempts to directly mimic real hardware would have to conform to the exact specifications of the device, including any hardware bugs. This can result in reduced performance, as each operation usually entails a trap to some software which emulates the specific behavior of the hardware device.

**Para-virtualization:** The second category is called paravirtualization. This is the method used by Xen [6]. In this model, the guest OS is aware that it is running in a virtualized environment, and therefore doesn't expect to interact with physical hardware on the system. The drivers in the guest OS are modified so that they communicate using higher level hypervisor abstractions. The VMM provides a simplified interface for the guest OS, for example, instead of trying to directly access the control registers and buffers of a network interface card, the guest OS will transmit high-level requests of network packets to the VMM, and the VMM subsequently directs the physical network interface of the system to perform this request.

The merit of this method is evident in the reduced overhead when compared to device emulation. However this technique is not perfect, since the drivers for a guest OS in a paravirtualized system may have to be partially re-written. With thousands of device drivers available today, this approach requires significant effort to convert existing drivers to use this technique.

**Hybrid:** In an effort to retain some of the security and reliability benefits of hypervisor-style VMM architectures, while simultaneously leveraging the facilities of an existing OS and its associated device drivers as in an OS-hosted VMM, some VMMs adopt a hybrid driver model. In a hybrid VMM architecture, a small hypervisor kernel controls core CPU and memory resources, but I/O resources are programmed by device drivers that run in a de-privileged service OS. The service OS functions in a manner similar to that of a host OS in that the VMM is able to leverage its existing device drivers.

While a hybrid VMM architecture offers the promise of retaining the best characteristics of hosted and hypervisor-style VMMs, it does introduce new challenges, including new performance overheads, due to frequent privilege-level transitions between the guest and service OS through the hypervisor. This technique is employed by LeVasseur et al. [14], in which the device driver hosted on its native OS directly controls its device via a pass-through enhancement to the virtual machine which permits it to access the device's register and ports and receive hardware interrupts. In order to prevent an attack using DMA mechanisms, this pass-through method needs to perform interpretation of DMA requests for the device, which can be a significant source of overhead.

#### 3.1.1 Our Approach

Our approach is different from existing approaches in that is not a pure software solution: we utilize hardware extensions, specifically Intel VT-x technology, to assign control of physical devices to drivers residing in virtual machines. This allows us to use hardware enforced isolation to protect the core OS running in the VMM from device drivers running on the virtual machines without the overhead of VMexits. Using VT-x device drivers running virtual machines can directly access the I/O ports and registers of a device.

While this works well for device drivers that only need access to I/O ports and memory-mapped registers, the current VT-x implementation does not currently support DMA mapping or interrupt remapping. This is an important drawback because it means that complete control of a hardware device cannot be completely delegated to a guest machine. It is left to the host itself to route interrupts to the appropriate VM and to handle DMA transactions. DMA transactions must be handled by the host so that a device driver cannot use DMA to access memory outside of the memory assigned to it in the virtual machine.

In order to achieve the desired performance and maintain the isolation, security, and reliability properties desired from virtual machines efficient hardware mechanisms for constrain all I/O operations, including DMA and interrupts, are required. These necessary mechanisms are provided by the recent Intel VT-d extensions; however, at the time of this writing, hardware equipped with these extensions was not available. Therefore, the experiments described in this paper use only the Intel VT-x technology. A more detailed discussion of this issue can be found at Section 4.2.

### 3.2 Shared Memory Communication between the Host and the VM

In the iKernel architecture, there are two roles: driver kernel and host kernel. The host kernel is a standard Linux environment where user applications may run. The driver kernel is a minimal Linux kernel isolated in a KVM virtual machine. The host kernel is protected from the driver kernel by the virtualization hardware.

The host and driver kernels communicate with each other to service driver requests. The communication channel we chose to use was shared memory, since other techniques such as message passing tend to have high overheads if messages are frequently exchanged. In our shared memory implementation, the host and driver kernels define the messaging format to be the most efficient representation for the task. This might include batching requests to minimize overhead costs.

The communication mechanism relies on a shared memory region between the host and driver kernels. This memory region is strictly isolated from the rest of the system, so the host kernel is not directly affected if the driver kernel fails. The negotiation for this shared memory area is performed by the driver and host kernels. Typically, the driver kernel will export a page within its memory space for the host kernel to access. To protect the confidentiality and integrity of the host kernel, the shared page must be allocated from the untrusted guest VM. This prevents an attack where a malicious guest may attempt to flood the host with shared page requests, which can exhaust the host of non-pageable kernel memory.

The VM and the host driver stubs agree on a common communication format to interoperate. For example, in the parallel port driver that we have implemented, the first three bytes of the shared region is used for the kernel to write the data to be outputted on the device LED. Using the common messaging protocol, the parallel port driver can then read the three bytes periodically to process any pending requests and perform the appropriate actions on the physical hardware device.

The details of our shared memory implementation will be discussed in Section 4.3.

| Entity | Size |
|---|---|
| Driver kernel size | 2 mb |
| Root file system size | 10 mb |
| Memory overhead of driver VM | 8 mb |
| QEMU memory overhead | 27 mb |

**Table 1. Size overhead of the driver VM**

# 4 System Implementation

In this section, we discuss the implementation details of iKernel. We demonstrate the feasibility of the iKernel approach by isolation of an unmodified Linux parallel port device driver operating in an unprivileged iKernel VM.

We run a minimal stripped down kernel with a small root file system to reduce memory and performance overhead of each driver VM. Table 1 shows the relative sizes of the driver kernel, file system image and memory overhead. This can be made very small to avoid the overhead of a traditional virtual machine which may weigh in at hundreds of megabytes of disk space, and dozens of megabytes of memory overhead. Based on the cumulative overhead of the QEMU emulator with 8 mb of guest memory, we observed the total overhead to be around 27 mb per driver VM instance. Given the size, we expect that this approach will be feasible to run several device drivers on a modern machine with an average of 1gb of memory without significant performance degradation.

Although we made an attempt to reduce the size of the guest VM, there is still room to improve the memory overhead of QEMU when used as a driver VM. For example, we can disable the network, audio, video and disk emulation as our system can run off a small kernel with a RAM disk.

## 4.1 System Setup

For our experiments, we used a Dell Precision 390 workstation with Intel VT-x virtualization technology. We have installed the most recent Ubuntu - version 7.04 built on Linux kernel 2.6.20 with KVM and QEMU integrated as a loadable driver module, which serves as our host OS running KVM.

KVM has two components:

- A device driver for managing the virtualization hardware; this driver is represented as a character device, /dev/kvm, which exposes a VM management interface to user space. Opening this device using the kvm client creates a new virtual machine which can then be manipulated with a set of ioctl() calls. Each virtual machine is represented as a process in the host OS.
- A user-space component based on QEMU for emulating PC hardware.

Effectively, the KVM driver adds a third execution mode, guest mode, in addition to the traditional kernel and user modes. Guest mode effectively acts as a separate isolation sandbox, which has a separate memory map, I/O permissions, and CPU state. Guest mode memory is accessible to the host kernel, but the guest kernel cannot directly access host memory in our architecture. By default, guest mode has no access to any I/O devices; any such access is intercepted and directed to user mode for emulation by the helper QEMU process. We have modified guest mode for iKernel so that hardware accesses are passed through to the hardware directly.

## 4.2 VM Direct I/O Access

The Intel VT-x technology provides VMs with an optional mechanism to communicate with the hardware devices directly without trapping to the host kernel. However, this feature is not enabled by default in KVM.

In the default configuration of KVM, the hardware generates a trap to the host kernel (a VMexit) when a VM tries to perform an I/O operation, so the drivers in a VM will not be able to communicate with the devices directly. This is the case because the KVM initialization code sets the VMX **Unconditional I/O** hardware bit to 0 for unconditional VMexits on VM I/O operations.

To enable direct communication between a VM and a device, we need to set up two parameters by modifying the Linux KVM implementation. These parameters, which provide access control to I/O ports, are located in the hardware VM control structure. Since Linux KVM by default does not initialize these structures, we have modified the kvm driver to do so.

The first parameter we need to set is the **activate I/O bitmaps** bit, which controls whether the hardware should consult I/O bitmaps to restrict executions of VM I/O instructions. If this bit is set to 0, it will not use the I/O bitmaps, but instead uses the Unconditional I/O bit to allow or disable VM I/O accesses to *all* I/O ports. This is not desirable because we only want to enable direct VM I/O accesses for certain I/O ports. Therefore, we set the activate I/O bitmaps bit to 1 so that the hardware will use the I/O bitmaps for fine-grained access control.

The I/O bitmaps indicate which hardware accesses to I/O ports should cause VMexits. There are two I/O bitmaps, A and B. Each of them is 4KB in size. I/O bitmap A contains one bit for each I/O port in the range 0x0000 to 0x7FFF, and I/O bitmap B contains one bit for each I/O port in the range 0x8000 to 0xFFFF. If any of these bits are set to one, a VM I/O access to that port will cause a VMexit to the monitor.

To set up the I/O bitmaps, we allocate two pages of free memory (4KB each), and then fill these pages with all bits set, and then clear bit 0x378, which is the ad-

dress of the parallel port. Then, we obtain the physical addresses of these two bitmap pages, and send the physical addresses to the KVM function `vmcs_write64()`, which will set up the hardware bits. The core instructions are shown in Figure 2. The `vmx_io_bitmap` contains the host virtual address of the first bitmap page. The `IO_BITMAP_A` and `IO_BITMAP_B` values are `0x00002000` and `0x00002002` respectively, which are the indices into the hardware VMCS field for the two bitmap addresses.

While the VT-d techniques provide hardware protection against attacks exploiting DMA capable devices, we did not experiment with VT-d in our experiments, because VT-d hardware was not available to us at the time this paper was written. Without direct hardware support for DMA and interrupt remapping (e.g. VT-d), it is conceivable to use pure software protection mechanisms as in previous work [12] to protect iKernel from DMA-based attacks. Additionally, incorporating VT-d into iKernel remains as future work.

### 4.3 Shared Memory Communication

We chose to allocate the shared memory region as a statically allocated memory region into the guest VM driver stub code. The shared memory area is loaded into memory when the driver is installed.

Whenever the host requests to send data to the parallel port, the host driver stub will write the data to the shared structure with the given messaging format. The VM driver stub will process the requests periodically, and when it detects a new message, it will send the data directly to the device without interrupting the host.

## 5 Observations and Discussions

**Performance of Hardware Virtualization Support:** We found that KVM with hardware virtualization support runs fairly well. However, there are still known bottlenecks in the KVM implementation. A recently released patch [8] claims that it can improve KVM performance by reducing VMM trapping due to the 0x80 I/O delay operation. Another source of overhead is that the Intel Virtualization techniques have not been perfectly tuned, and in some cases can trap more frequently than desired, which may cause performance problems as illustrated by the recent work from VMWare [2].

We measured the performance of our approach and found that the average latency of sending a request from the host to the driver VM was 12 ms. This number is very close to the time slice granularity of a Linux kernel. We suspect that the host sends a message, then within a single time slice (10 ms), the driver VM becomes scheduled, and processes the request. Also, the Intel Core2Duo machine we were using has multithreading support, so the other core may have processed a request initiated from the first core.

**Latency:** We performed some rudimentary experimentation with lowering the latency of the request. One experiment we performed was to raise the priority of the KVM process into the real-time FIFO class. Given this priority, we were hoping that the KVM process would receive more CPU cycles. In order for this strategy to be successful, the host must allow the KVM process to be scheduled as close to the submission of the request as possible, and the guest kernel running in the KVM virtual machine must schedule the driver stub at the earliest opportunity. In practice, we did not perceive any large performance gain. We suspect that the real-time FIFO scheduling class on our setup was still subject to the 10 ms scheduler timeslice. Another way to decrease latency would be to change the system's timeslice to be smaller so that the driver VM can be scheduled earlier. We did not try this experiment, as changing the system timeslice can have adverse effects on other time-critical processes. Finally, we could optimize the scheduler to favor the KVM process when a request is pending. We did not get the opportunity to try this approach either.

Ideally, the host kernel should give priority to the driver VM when a request is pending. In order to efficiently do this, the host must know which VM contains the appropriate driver. This can be done in the host stub driver, which has a linkage to the associated driver VM. The driver VM must also be aware of the request and schedule it as early as possible. When the guest VM is done, it must relinquish the processor as early as possible to be able to service any pending requests quickly. Currently, the guest relies on running out of timeslice, or scheduling of the idle loop to pass control back to the host. It would be better if the guest stub driver used VM hints to tell the KVM system to yield. Alternatively, gang scheduling of the driver VM and host driver stub on a multicore system may also be a good approach to reduce latency without too many changes to the host and guest scheduling strategies.

## 6 Related Work

Research in system dependability is a relatively well-studied topic. In this section, we summarize several popular dependability approaches such as: sandboxing, system design, safe languages, and static analysis.

**Microkernel and Virtual Machine Based Approaches:** LeVasseur et al. [12] proposed executing a device driver on its native OS within a virtual machine in order to isolate the device driver without modifying it. Their solution is aimed at device driver reuse and improved system dependability

```
vmx_io_bitmap = (char *) __get_free_pages(GFP_KERNEL, 1);
// some error checking code
...
memset(vmx_io_bitmap, ~0, 2 * PAGE_SIZE);
clear_bit(0x378, vmx_io_bitmap);
...
vmcs_write64( IO_BITMAP_A, (unsigned long) __pa( vmx_io_bitmap) );
vmcs_write64( IO_BITMAP_B, (unsigned long) __pa( vmx_io_bitmap + PAGE_SIZE) );
```

**Figure 2. Core code segment to set up the two I/O bitmaps.**

by driver isolation. The architecture consists of a hypervisor (privileged kernel) which multiplexes the processor and provides protection for memory and I/O ports. The VMM allocates and manages the resources (devices) and implements the virtualization layer. The device driver hosted on its native OS directly controls its device via a pass-through enhancement to the virtual machine which permits it to access the device's register and ports and receive hardware interrupts. To isolate device drivers from each other, the drivers are executed in separate and co-existing virtual machines. Any client running on top of the virtual machine can interface with a device driver via a translation module added to the device drivers' OS. This module maps client requests into sequences of device driver/native OS primitives for accessing the device, and converts completed requests into responses to the client. Communication between the client running on one virtual machine and device driver running on another virtual machine is in the form of microkernel message passing. The device driver isolation helps to improve reliability by preventing fault propagation. This approach does not leverage the hardware support for virtualization provided by Intel and AMD.

The Microkernel [13] approach appears promising, but it cannot be applied to existing legacy operating systems (Windows, Linux) which do not have a microkernel-based architecture. However, our approach can be applied to these operating systems with no modifications needed for the device driver and guest operating systems and very slight modification to the Kernel Virtual Machine (KVM).

**New Operating System Design:** Choices and Singularity [3, 10, 5] aim to build a secure and reliable operating system from scratch. Although the new OSs have many security, maintenance and performance advantages, they can not improve the reliability and security of existing operating systems, such as Linux.

**Other Approaches:** Nooks [16] is a reliability subsystem that greatly improves OS reliability by isolating the OS from driver failures. It achieves this by executing each driver in a lightweight kernel protection domain, which is a privileged kernel mode environment with restricted write access to kernel memory. It creates a new OS reliability layer that is inserted between the drivers and the OS kernel. Nooks is a departure from the virtualization approach for improving OS reliability and fault isolation. It can be perceived as virtualizing only the interface between the kernel and the driver (instead of virtualizing the underlying hardware). The design tenet of Nooks was to provide resistance against buggy drivers, not necessarily malicious drivers. Since Nooks still operated in privileged mode, a driver was still capable of defeating the memory protection by changing page tables to manipulate memory mappings. The extra permissions were granted to retain compatibility with other privileged operations such as enabling/disabling interrupts. Our approach runs in privileged guest mode, which allows drivers to operate with the same semantics, but VMX protects the host kernel memory. Experiments reported in the paper show that in a large number of fault-injection tests, Nooks recovered automatically from 98% of the faults that caused the OS (Linux) to crash. However, it does not provide complete isolation or fault tolerance against all possible driver misbehaviors (especially malicious ones). Since Nooks runs extensions in kernel mode, if an extension (device driver) is deliberately designed to corrupt the system, Nooks will not be able to prevent it.

SafeDrive [17] is language-based approach to protect extensions from corrupting or crashing the kernel. The key idea of SafeDrive is to prevent errors in driver code from causing any damage outside the driver. The kernel and the driver coexist in the same protection domain and isolation is enforced through language mechanisms. SafeDrive can detect type safety and memory related bugs. It works by having developers add annotations to source code and automatically translating them into runtime checks to guarantee type safety and detect many memory related bugs, such as buffer overflows and null pointer dereferences. All this is done at the source code level and so the costs are reduced when compared with other approaches [16, 12]. However, it has its own limitations: the errors it can detect and recover from are limited to memory related bugs and type safety violations; there is no special handling for memory leak bugs;

and more importantly, it is not designed to handle malicious driver code.

XFI [7] is based on controlling the flow of the program by placing software guards at strategic control points of a program and having a predetermined control flow graph for the program devised by static analysis of the code. The guards work by ensuring that control can only transfer to one of the predetermined locations in code. Any deviation from this generates an error. But placing these guards in an already existing legacy operating system with millions of lines of code is an extremely time-consuming task and the process of placing the guards might miss some critical locations.

In a hardware-based approach [14] for driver isolation on x86 under Linux, runtime loadable modules containing device drivers code are run in lower privilege x86 rings (rings 1 and 2), and memory isolation prevents drivers from accessing kernel data directly. If a driver attempts to access kernel memory, this violation is detected in hardware and the offending driver is unloaded by the kernel. A new API wrapper layer facilitates control transfers between drivers and the kernel using x86 call gates. This approach requires existing drivers to be recompiled (without changes) to work with the modified kernel. Also, similar to Nooks, it does not completely solve the problem. Drivers can corrupt themselves or anything else running in their ring or any lower ring. Also, because drivers are trusted with the interrupt enable flag, deadlocks can occur. Finally, malicious drivers can still cause system crashes or open security holes.

None of the four approaches above can completely protect the kernel from malicious device drivers.

## 7 Conclusions and Future Work

In this paper we have described the design and implementation of iKernel, a framework which provides isolation for device drivers to increase the reliability and security of a commodity operating system. The iKernel approach allows device drivers to run in a virtualized environment without incurring significant trap and messaging overhead present in other systems. To demonstrate the feasibility of our approach, we have implemented and analyzed a prototype parallel port driver residing in an isolated guest driver virtual machine utilizing Intel VT-x technology. In the future, we plan to extend the iKernel framework to support Intel VT-d technology and extend the iKernel to other I/O devices such as graphics cards. Another direction for future work is to improve driver latencies.

## Acknowledgments

We thank the anonymous reviewers and Robin Snader for their invaluable comments and help. This project was sponsored in part by NSF CNS-0347854 (A64421), grants from the Motorola Communications Center (Project 34) and Microsoft (434U5L).

## References

[1] D. Abramson et. al., "Intel Virtualization Technology for Directed I/O." Intel technology journal, Volume 10, Issue 3, 2006.

[2] Keith Adams and Ole Agesen. "A comparison of software and hardware techniques for x86 virtualization." In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[3] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. "Choices, Frameworks and Refinement." In International Workshop on Object Orientation in Operating Systems, 1991.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An Empirical Study of Operating Systems Errors." In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP), 2001.

[5] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, Philip A. Reames, Roy H. Campbell, "Improving Dependability by Revisiting Operating System Design." In Workshop on Hot Topics in Dependability, 2007.

[6] B. Dragovic et. al. "Xen and the Art of Virtualization." In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP), 2003.

[7] U. Erlingsson, M. Abadi, M. Vrable "XFI: Software Guards for System Address Spaces." In Proceedings of the 7th Operating System Design and Implementation (OSDI) 2006.

[8] Qing He. "VMX: enable io bitmaps to avoid IO port 0x80 VMEXITs." http://article.gmane.org/gmane.comp.emulators.kvm.devel/2621, 2007.

[9] J.N. Herder, H. Bos, P. Homburg, A.S. Tanenbaum. "MINIX 3: A Highly Reliable, Self-Repairing Operating System" In Proceedings of the ACM SIGOPS Operating Systems Review, 2006.

[10] G. Hunt et. al. "An Overview of Singularity Project." Microsoft Research Technical Report 2005.

[11] A. Johansson, N. Suri, "Error propagation profiling of operating systems." In Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2005.

[12] J. LeVasseur, V. Uhlig, J. Stoess, S. Gotz. "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines." In Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI), 2004.

[13] J. Liedtke, "On Microkernel Construction." In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP), 1995.

[14] D.A. Kaplan. "RingCycle: A Hardware based approach to driver isolation." http://www.acm.uiuc.edu/projects/RingCycle/browser/RingCycle.pdf, 2006.

[15] M. Sullivan and R. Chillarege. "Software Defects and their Impact on System Availability A Study of Field Failures in Operating Systems." In Proceedings of the 21st International Symposium on Fault Tolerant Computing (FTCS-21), 1991.

[16] M. M. Swift, B.N. Bershad, H.M. Levy. "Improving the Reliability of Commodity Operating Systems." In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP), 2003.

[17] F. Zhou, J. Condit, Z. Anderson, I. Bagrak "SafeDrive: Safe and Recoverable Extensions Using Language Based Techniques." In Proceedings of the 7th Operating System Design and Implementation (OSDI), 2006.